An ECA Rule-Matching Service for Simpler Development of Reactive Applications

Diego López de Ipiña

Laboratory for Communications Engineering
Cambridge University Engineering Department
Cambridge, UK
Tel: +44 (0)1223 766513
Fax: +44 (0)1223 766517
dl231@eng.cam.ac.uk

Abstract. This paper describes the implementation of a CORBA-based Event-Condition-Action (ECA) Rule Matching Service. This service complements the standard CORBA Notification Service with a Composite Event Matching Engine. The originality of this work resides on having leveraged on the pattern matching capabilities of a Production Systems Programming Language, i.e. CLIPS, for the implementation of the event correlation and aggregation process. A powerful ECA Rule Specification Language, easily mapped to CLIPS, has been designed that hides the inconvenient syntax of the underlying programming language and extends it with suitable features for event correlation. An ECA Server receives ECA rule subscriptions from clients, maps them to CLIPS, and delegates the rule-based event correlation process to an embedded CLIPS inference engine. CORBA Structured Events received by the ECA Server are mapped into CLIPS facts. The ECA Service main capabilities are to enable composite event matching and event aggregation and to trigger a set of pre-defined and arbitrary actions as a consequence of a matched situation. This service highly simplifies the development of reactive applications by alleviating the programmer from the implementation of complex composite event handling mechanisms. An example is given illustrating the potential of this service for the implementation of a sentient application reactive to the stimuli received from environmental sensors.

1. Introduction

A reactive system performs actions in response to events. Our research is focused on an interesting type of reactive systems, namely sentient systems [Hopper00]. These systems respond to the stimuli provided by sensors distributed through the environment by triggering actions that are adequate to the changing context of the user, e.g. his identity, location or current activity.

The development of even a conceptually simple sentient application is non-trivial because it encompasses the cooperation of several distributed elements, such as sensors, databases or effectors. Usually, a sentient application needs to register with two or more distributed event sources and to monitor when a pre-defined combination

of events occurs. Consider we want to automatically initiate the playback of a user's favourite music whenever he starts working. For that, the application would have to register interest with a location system event source to receive presence events and with a keyboard activity monitor event source to be notified when the keyboard activity of the user's computer changes significantly. When a user is both in his office and typing regularly then the application would initiate the music playback.

As observed by this example, the *modus operandi* of sentient applications, and reactive applications in general, follows a common pattern, i.e. they wait until a pre-defined *situation* (a composite event pattern) is matched to trigger an *action*. In other words, reactive applications respond to an *Event-Condition-Action* loop. Consequently, it is ineffective to enforce each reactive application to handle this behaviour separately. A generic middleware capable of undertaking the composite event monitoring process, i.e. the correlation of multiple simple or composite events, on behalf of applications is hence necessary.

This paper describes the implementation of an Event-Condition-Action (ECA) Rule Matching Service. The ECA concept is inspired by past research in Active Databases [Paton+99]. Application developers specify with the help of a powerful ECA Rule Specification Language sophisticated conditions over events supplied by diverse event sources and associate to them actions, e.g. notification of an aggregated event summarising a set of matching events. A key factor that distinguishes our work from previous research in Composite Event Management [Bacon+00][Gruber+99] is that our Rule Matching Engine leverages on the pattern matching capabilities of the inference engine of a Production Systems Programming Language, i.e. CLIPS, to undertake the complex rule-based reasoning. Other approaches required a whole implementation of the matching engine from scratch. The core of our work has been to establish a smart mapping between our proposed rule specification language and CLIPS rules. A mapping is also performed from the events provided by our chosen notification service, namely the CORBA Notification Service, and CLIPS facts.

Section 2 gives some details about the CORBA Notification Service upon which our middleware service is built. Section 3 describes the operation, architecture and rule specification language defined by the ECA Rule Matching Service. Next, section 4 gives an example of how much easier is to build a sentient application with our middleware service. Finally section 5 states the direction of our current research and draws some conclusions.

2. The CORBA Notification Service

The CORBA Notification Service [OMG00] decouples the communication between CORBA objects and permits their interchange of events in an *asynchronous* form through Notification Channels. A *Notification Channel* is both a supplier and a consumer of events. This object allows multiple suppliers to communicate with multiple consumers asynchronously and without knowing about each other. It is responsible for supplier and consumer registration, timely and reliable event delivery to registered consumers, and the handling of errors associated with unresponsive consumers. This service also provides Structured Events support, event type

discovery, event filtering, sharing of filters among several consumers, QoS properties, and an optional Event Type Repository, where event type descriptions are stored.

The main limitation of the Notification Service is that it enables filtering only on atomic events, not providing support neither for compound event matching nor for event aggregation. By *composite event* is understood an asynchronous occurrence triggered by the correlation of multiple atomic or composite events. *Event aggregation* is defined as the process by which a set of matching events is summarised by a single new event.

This work addresses the event composition and aggregation aspects not addressed by the Notification Service by defining an ECA Rule Matching Service. An important assumption of this service's implementation is that events supplied to the ECA Service comply with the structured event format specified by the OMG's Notification Service. Nevertheless, the modular architecture of the devised Rule Matching Service allows for the simple integration of other types of event notification services to the system.

OMG's *Structured Events* consist of a header, a filterable body, and a remainder of body. The header consists of event type, and event name, and an optional variable part that may contain QoS-specific attributes, timestamps, access rights and so on. Event types contain a domain name (D) and a type name (T). The domain name is used as a namespace in C++. The filterable body part contains attribute-value pairs, while the remainder of the body is an opaque value. This work assumes that every event supplied contains in the filterable body two event-independent attributes, namely `sourceID` and `timestamp`, apart from the set of event type dependent attributes. The `sourceID` depicts the issuer of the event. The `timestamp` denotes when the event was issued. This event timestamp allows temporal comparison of events. This is an important requirement as the delivery of events within a distributed system may be subject to delay. To guarantee the clock synchronisation of every machine within a LAN, the Network Time Protocol (NTP) is used.

3. A Middleware Service for ECA Rule Matching

Declarative languages such as Prolog or CLIPS, used in the context of Logic and Expert Systems Programming respectively, are specialised in the description and analysis of relationships. Given a set of *facts* (true propositions about entities) and *rules* applied to them, an *inference engine* built directly into the language can decide what rule to fire. In order to solve a problem, programmers only need to specify a set of *situation-action* (or production) rules and facts. The *situation* (IF) component takes the form of predicates applied to values of attributes of a specified object. The *action* component (THEN) of the rule produces new knowledge or triggers an action. Surprisingly, very few researchers have exploited the inherent benefits of these languages in the specification of reactive systems' behaviour rules.

This work, based on the above observation, leverages on the built-in pattern matching capabilities of CLIPS (C Language Integrated Production System) [NASA99] to provide a middleware service, termed ECA Rule Matching Service, that undertakes the common event composition and aggregation tasks required in the

implementation of reactive systems. CLIPS suitability for this task is explained by three reasons: (1) the expressive power of this declarative language permits the specification of very complex relations of event patterns; (2) its built-in inference engine implements the RETE [Forgy82] algorithm, a very efficient mechanism to solve the difficult many-to-many matching problem, that is very suitable for our task; and (3) CLIPS is designed for full integration and extensibility with procedural languages such as C++ and Java. In addition to being used as a stand-alone tool, CLIPS can be called from a procedural language, perform its function, and then return control back to the calling program. Likewise, procedural code can be defined as external functions and called from CLIPS. When the external code completes execution, control returns to CLIPS. However, CLIPS obscure syntax based on LISP, such as the overuse of parenthesis and the need to use inverse polish notation for building arithmetic and conditional expressions, make it an inconvenient language for the programmer. This work proposes a new language that adds some syntactic sugar to CLIPS, keeps its rich expressive power and defines a useful set of extensions for event composition and correlation (see section 3.3) that hide the low-level intricacies of such a process.

3.1. ECA Service Operation

The main function of an instance of an ECA Service, or ECA Server, is to accept, match and manage Event-Condition-Action rule specifications on behalf of a client application. A specification describes a set of event patterns and conditions applied upon them, as well as the actions to be triggered when the given situation is matched. An ECA Server receives as input both ECA rule registrations and CORBA Structured Events. As result of a situation match an ECA Server triggers one the following three actions:

1. *Event Composition*: the ECA Server notifies a consumer the batch of event instances corresponding to a composite event pattern match.
2. *Event Aggregation*: the ECA Server notifies a consumer with a new event summarising an event correlation.
3. *Action Execution*: the ECA Server executes the action triggered as result of a matching situation. This action may correspond to a set of common actions provided by the service, e.g. sending an email, playing a sound or issuing a verbal notification through a text-to-speech processor, or an arbitrary script specified by the user may also be run.

3.2. ECA Server Architecture

Figure 1 shows the building blocks of an ECA Server. The heart of an ECA Server is an embedded CLIPS inference engine. This reasoning engine determines when the left-hand-side (LHS) of user specified rules are matched by the current set of facts stored in the knowledge base of the engine. As a result of this process, the engine initiates the actions specified in the right-hand-side (RHS) of rules.

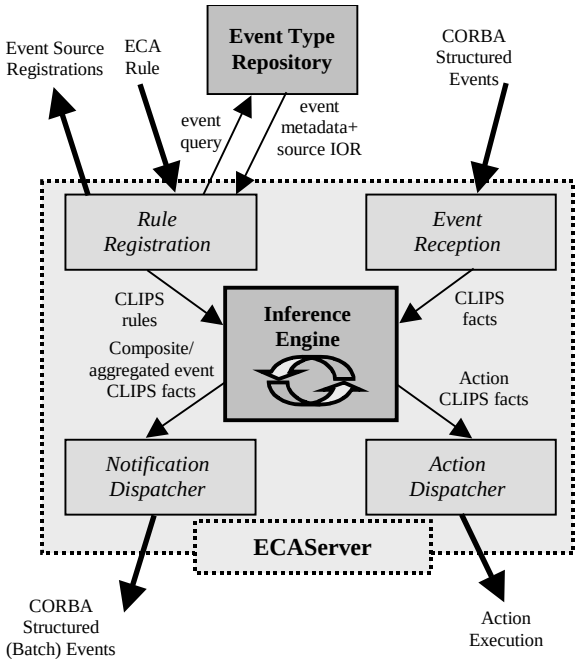

Figure 1. ECA Server Architecture

The Event Reception Module populates the knowledge base of the embedded CLIPS engine with facts representing the events coming from distributed event sources. This module understands events supplied by a given event notification platform, in our implementation OMG's Notification Service's Structured Events, and maps them into facts representing events that are understood by CLIPS. In the Structured Event decomposition process, required to undertake their mapping into CLIPS facts, the data type introspection capabilities of CORBA [OMG99], provided through the TypeCode and DynAny interfaces, are used. A similar process is used by the Notification Dispatcher to generate on-the-fly aggregated events whose contents are unknown at compile time.

The Rule Registration Module maps client issued rules, expressed in our ECA Rule Specification Language (see section 3.3), into CLIPS rules, and inserts them into the embedded engine. In order to undertake run-time checking of the validity of the event templates passed within a rule specification, this module needs to query an external Event Type Repository (ETR). This component is populated with meta-data describing the structure of all known event types that may be supplied to an instance of a Notification Channel. Event sources announce to this server the meta-data of the events they generate and the object references of the Notification Channels where those events are supplied. The Rule Registration Module queries the ETR to retrieve both the metadata and the object references of Notification Channels where events

types expressed within a given ECA Rule are supplied. Then, it registers to such channels as a consumer.

The Notification Dispatcher maps engine supplied CLIPS facts representing aggregated events into CORBA Structured Events and facts representing a composite event match into a batch of CORBA Structured Events. These events are then communicated to the clients that issued the firing rules. Aggregated Structured Events are also supplied to a Notification Channel associated with each ECA Server. Thus, applications may be developed that just connect to an ECA Server's channel passing a filter constraint expressing their interest on a given type of aggregated events, without having to explicitly register with the ECA Server.

The Action Dispatcher processes the facts representing actions sent by the engine and, as a consequence, either triggers some of its built-in common actions or initiates a user specified script.

The modular architecture of the ECA Rule Matching Server permits the replacement of the modules specific to the event format of a given notification middleware for others suitable for another notification technology.

3.3. A Language for Specifying ECA Rules

A convenient way for reactive applications to use event facilities and express rules applied to them is to use a declarative language, as it was remarked at the beginning of section 3. This section details the basic elements of a language that has been devised for this purpose. An ECA rule specification is composed of a set patterns applied to the contents of events together with the actions that are triggered by successful matches:

```
<rule> ::= <pattern_list> => <action_list>
```

A rule may be applicable within an ECA Server permanently or be fired just a fixed number of times. Moreover, the evaluation of a rule may wished to be active a maximum period of time, i.e. a rule may need to enforce that correlated events within it occur within a given time span. The times a rule may be fired and its default detection time span are specified on rule subscription.

The most common pattern within a rule corresponds to an event template. An event template contains the domain name and type name of the event and a set of name value pairs corresponding to the attributes of the event that want to be used for matching. For instance, `ActiveBadge$presence(user "Peter", room ?roomID)` matches the first event belonging to domain `ActiveBadge` and of type `presence`, where the attribute `user` is equal to the constant `"Peter"` and the contents of the attribute `room` can have any value and are assigned to the variable `?roomID`. Attribute name identifiers may be separated by dots to denote nested attributes, e.g. `location.Xpos`. Note that variable declarations are preceded by a question mark. Optionally, event patterns may also be preceded by a variable. Event representing variables are used when the set of events conforming a composite event is communicated to a client as an Event Batch. Thus, the generic form of an event pattern is expressed as:

```
[<event_var>:]<domain>$<event_type>(<name_value_list>)
```

Simple event matching expressions can be combined to form compound matching expressions that describe a combination of events using the operators `and`, `or`, `not` and `then`. The semantics of the first three operators coincides with the semantics of their analogous operators in standard high-level languages. Nevertheless, a `not` operator preceding an event template signifies that no occurrence of such event should take place. The operator `then` indicates a temporal sequence, i.e. it does not attempt to match patterns on its RHS until its LHS ones have been matched.

A `test` expression can be applied to variables corresponding to the fields of event templates, to further constraint the set of possible matches. This expression must return a Boolean value. This construction is useful for expressing both intra-event and inter-event constraints. All the conventional relational and arithmetic operators can be used to build an expression, e.g. +, *, < or =.

The action part of a rule is executed only after the antecedent of the rule has successfully matched. The supported action statements are: (1) variable assignment, (2) event notification and (3) command execution.

```
<action_stmt> ::= notifyEvent(<event>); |
                  fireAction(<action-name>, <params_list>); |
                  <VARIABLE> := <expr>;
```

An assignment action binds a variable to an expression. This is most often used to create the fields composing a new aggregated event or the parameters to be passed when firing an action. An event notification action may be triggered whenever either the batch of events corresponding to a matching composite pattern is communicated or when a new aggregated event is generated. The parameters passed within an aggregated event correspond either to constant values or to variables declared before the statement. The `eventBatch` statement must pass as parameters only variables that have previously been bound to event templates.

```
<event> ::= <EVENT_ID>(<event_params_list>) |
            eventBatch(<event_var_list>)
```

An action may be fired as response of a matched situation. The action name may correspond to one of the common actions available with the implementation of the ECA Rule Matching Service or correspond to the file path of an arbitrary script created by the application designer.

Finally, some commonly used variables to be used in expressing temporal constraints such as `dayofweek`, `date` or `curtime` have been incorporated.

3.4. Event-representing Facts Storage and Cleanup

Events are discrete temporal occurrences. Some event sources produce events at regular intervals, others irregularly. Furthermore the average frequency of event reporting differs from one event source to another. On the other hand, the CLIPS-based implementation of the ECA Rule Matching Services implies that Structured Event representing facts may be added indefinitely to the Knowledge Base of the embedded CLIPS inference engine. For all these reasons, it is necessary to cache in the inference engine some of the events coming from event sources, so that the matching engine can use the last event reported by an infrequent source of events.

Still the number of events cached should be minimised in order to reduce the memory needs of the engine.

Our approach to tackle these issues is to make one or several attributes of an event type *primary*. Then, for each differing value of this primary attribute(s) the most recent corresponding event is cached. For example, in the case of the `presence` event above mentioned the attribute `user` would be considered primary so that the Rule Matching Engine would cache only the last occurrence of a user's presence event. In order to indicate the rule engine to match an event template with the last occurrence of an event rather than with a new instance of it to come, an event template pattern expression in the ECA language must be preceded by the keyword `query`.

For the case of event or action representing facts added to the Knowledge Base of the engine as result of the action part (RHS) of a rule, the ECA Rule Specification Language compiler will generate rules to retract (or garbage collect) them. The rules retracting those facts are triggered as soon as the rules inserting them finish execution.

3.5. Implementation Details

An implementation of the ECA Rule Matching Service has been completed in Java using the default CORBA ORB included in Sun's JDK 1.3 distribution. A Jess (Java Expert System Shell) [Friedman-Hill01] inference engine, embedded in an ECA server, undertakes the rule-based reasoning. Jess is a Java written clone of CLIPS that provides a neat integration with Java. The JLex [Berk+00] lexical analyser and CUP [Hudson99] parser generator tools were used to implement the mapping from ECA rules into CLIPS rules. The CORBA Notification Service implementation used for our experiments is omniNotify [AT&T01].

4. Building Applications with the ECA Rule Matching Service

Consider that a sentient jukebox application wants to be built that determines whether it is suitable to initiate music playback for a user, and if so, that decides the kind of music to play, according to the user's contextual conditions, e.g. day of the week, his activity and location, etc. In order to build such application a number of rules describing the behaviour expected by users upon different set of contextual conditions would have to be specified. An example rule could be:

"If it is Monday, a user is working and it is raining, then play some cheerful music to raise the user's spirits".

Without the support of a middleware providing composite event detection, the reactive application would need to register with the event sources providing the contextual information of interest, e.g. people location events, people log in events, keyboard activity events or weather condition events. Then the application would need to correlate all these atomic events to determine when a composite event fulfilling a complex condition would be matched. The implementation of only the rule

above mentioned would be rather complex. If several rules of this kind would also need to be implemented the application development would become even harder. Moreover, if the rules associated to an application would need to be changed later on, the hard-coded composite event monitoring process would have to be modified. This is a result of not separating the application logic from the functionality implementation.

Jukebox$play_music		
Name	*Type*	*Content*
user	String	Username of a user, e.g. *dl231*
host	String	Domain name of host, e.g. *heineken*
kind	String	Type of music to play, e.g. *ROCK*
Timestamp	TimeT structure	Secs (`high`) and μsecs (`low`) since 1/1/ 1970

Figure 2. Jukebox$play_music as a CORBA Structured Event

```
(deftemplate Jukebox$play_music
   (slot eventID)
   (slot *STRUCT_timestamp)
   (slot user)
   (slot host)
   (slot kind))
```

```
(deftemplate Time_T
   (slot eventID)
   (slot high)
   (slot low))
```

```
(Time_T (eventID event0) (high 946080000) (low 34456))
(Jukebox$play_music (eventID event0) (user "dl231")
  (host "heineken") (kind “ROCK”) (*STRUCT_timestamp factID))
```

Figure 3. Jukebox$play_music Structured Event mapped to CLIPS

Through the use of our devised ECA Rule Matching Service, the development of the sentient jukebox application would be much easier. First, the developer would define a high-level event type that would contain the information necessary to trigger the playback of music in a given machine. For example, the application could define a `play_music` event, see Figure 2, containing as attributes: the `user` for which music would have to be played, the `host` where the music should be played and the `kind` of music to play. Secondly, the application would register the metadata associated to this new type of event, belonging, for instance, to the domain `Jukebox`, with the Event Type Repository. Finally, the developer would create rules, containing as LHS, a set of conditions over events that want to be monitored and as RHS a notification action reporting the `play_music` aggregated event. Figure 4 shows the definition of such a rule in the ECA Language, whereas Figure 5 depicts how this expression is mapped to CLIPS by our language compiler. An analogous process should be carried out to define rules triggering a `stop_music` aggregated event when the set of changing contextual conditions so indicate, e.g. our boss is approaching to our room.

The ECA Server upon reception of an ECA Rule will delegate to the Rule Registration module the parsing and mapping of the rule to CLIPS. If an unsuccessful parsing of the rule occurs, the client is reported about the errors of its specification. Otherwise, the Rule Registration Module undertakes a run-time event type checking on the contents of the specification. If it does not know about certain types of events,

it will contact the ETR to retrieve the metadata associated to the unknown event types. The ECA Server caches internally the retrieved event type descriptions for later use. Next, the Rule Registration Module registers with all the event sources supplying the requested event types in the rule. Finally, this module inserts the ECA rule mapped to CLIPS into the embedded CLIPS inference engine. Observe that in the ECA to CLIPS mapping, the ECA language compiler generates all the temporal constraints implicit in an ECA rule specification.

```
query PCMonitor$login(user ?userID,host ?hostID) then
(ActiveBadge$presence(user ?userID) and
 PCMonitor$keyboard_activity(host ?hostID, level ?m) and
 test(?m > 0.3) and
 query WeatherMonitor$report(raining ?rainIntensity) and
 test(?rainIntensity > 0.2) and
 test(dayofweek = "Monday"))
=>
notifyEvent(Jukebox$play_music(?userID, ?hostID, "ROCK"));
```

Figure 4. PlayCheerfulMusic Rule in the ECA Rule Specification Language.

```
(defrule PlayCheerfulMusicRule
  (rule (ruleID 1234)(*STRUCT_timestamp ?ruleRegTime))
  (PCMonitor$login (user ?userID) (host ?hostID)
                   (*STRUCT_timestamp ?time0))
  (ActiveBadge$presence (user ?userID)
                        (*STRUCT_timestamp ?time1))
  (test (moreRecentTS ?time1 ?ruleRegTime))
  (test (moreRecentTS ?time1 ?time0))
  (PCMonitor$keyboard_activity (host ?hostID) (level ?m)
                               (*STRUCT_timestamp ?time2))
  (test (moreRecentTS ?time2 ?ruleRegTime))
  (test (moreRecentTS ?time2 ?time0))
  (test (> ?m 0.3))
  (WeatherMonitor$report(raining ?rainIntensity))
  (test(> ?rainIntensity 0.2))
  (test(str-compare (dayofweek) "Monday"))
=>
  (bind ?newEventID (getNewEventID))
  (bind ?currentTime (time))
  (bind ?time3 (assert (Time_T (eventID ?newEventID)
    (high (getHighPart ?currentTime))
    (low (getLowPart ?currentTime)))))
  (bind ?factID
    (assert (Jukebox$play_music (eventID ?newEventID)
       (user ?userID) (host ?hostID)
       (kind "ROCK")(*STRUCT_timestamp ?time3))))
  (notifyEvent ?factID))
```

Figure 5. PlayCheerfulMusic Rule mapped to CLIPS

Arriving CORBA Structured Events, see Figure 2, are mapped by the ECA Server's Event Reception Module into CLIPS facts, see Figure 3. This module decomposes the contents of arriving events based on the event content descriptions stored in the event type cache kept by the ECA Server. Basically, this process consists on flattening the contents of the Structured Event into a string representing a CLIPS fact.

On successful matching of the situation expressed by the LHS of a rule, the engine will issue requests into either the Notification or the Action Dispatching Modules in order to trigger the pertinent action. In Jess, new functions to the Jess language can be added by simply writing a Java class that implements the `jess.Userfunction` interface, creating a single instance of this class and installing it into the Jess engine. Both the Notification and Action dispatching modules are implementations of this interface. They represent gateways into a complex Java subsystem from the internals of the Jess inference engine.

5. Future Work and Conclusion

Presently, we are working on the idea of making the creation of reactive sentient applications feasible even for computer illiterate users. We believe it is paramount the involvement of end-users in the definition of rules that delimit their expectations from a sentient system. Only in this way, sentient systems will respond to users with the 'right' actions at the right time, satisfying end-user individual needs and making interactions more satisfactory and undisruptive.

This ongoing research proposes the association of every user in a sentient environment with an Event-Condition-Action Agent, ECAgent in short. ECAgents are small scale ECA Servers that embody a set of *situation-action* rules governing the interactions of the user with his sentient living space. We are aiming to create a sophisticated GUI for ECAgents that will enable users to define generic ECA rules. It seems feasible to be able to specify a complex situation with a GUI that will be mapped to our ECA language. However, the specification through a GUI of generic actions appears more challenging. So far, our system has been constrained to assign to a situation a set of pre-defined actions, or to run an arbitrary script specified by the user. The CORBA Dynamic Invocation Interface [OMG99] enables the creation of requests on objects without having compile-time knowledge of their interfaces. Middleware solutions as LocALE [Ipiña+01] permit the dynamic activation, migration and deactivation of services on demand. These tools provide the infrastructure support for the creation of generic actions. The problem, however, is to design a GUI that in an intuitive way permits the user to assign a generic action to a situation. We believe this is still an unresolved research issue.

An ECA Rule Matching Service has been created that makes the development of reactive application a less cumbersome process. The ECA Language defined enables the creation of sophisticated rules expressing complex conditions upon the atomic events that may constitute a high level composite event and the actions to be taken when those conditions are met. In the action side of a rule, a programmer may specify a high level aggregated event or the batch of events corresponding to the matched

situation to be notified, or, alternatively, a set of pre-defined or user specified actions to be fired. Our approach relieves the programmer from the tedious and difficult process of composite event handling. Programmers simply concentrate on the specification of the situation wished and delegate its matching to our service. They only need to implement the reaction desired when a notification of the matched situation is received from the ECA Rule Matching Service. The middleware service devised represents our solution to the very difficult task of making computers understand the current contextual situation surrounding them.

References

[AT&T01] AT&T Research, omniNotify Home Page, 2001, http://www.research.att.com/~ready/omniNotify/index.html

[Bacon+00] Bacon J., Moody K., Bates J., Hayton R., Ma C., McNeil A., Seidel O. and Spiteri M.. "Generic Support for Distributed Applications". IEEE Computer, pages 68-76, March 2000.

[Berk+00] Berk E. and Ananian C.S. "JLex: A Lexical Analyzer Generator for Java", Princeton University, 2000 http://www.cs.princeton.edu/~appel/modern/java/JLex/

[Forgy82] Forgy C.L. "Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Artificial Intelligence 19, pp. 17-37, 1982

[Friedman-Hill01] Friedman-Hill E. "Jess: the Java Expert System Shell", Sandia National Laboratories, http://herzberg.ca.sandia.gov/jess/, 2001

[Gruber+99] Gruber R. E., Krishnamurthy B. and Panagos E. "The Architecture of the READY Event Notification Service". Proceedings of the 19th IEEE Internation Conference on Distributed Computing Systems Middleware, May 1999.

[Hopper00] Hopper, A., "The Clifford Paterson Lecture, 1999 Sentient computing", Philosophical Transactions of the Royal Society London. A (2000), 358, 2349-2358.

[Hudson99] Hudson S. "CUP Parser Generator for Java", Princeton University, 1999 http://www.cs.princeton.edu/~appel/modern/java/CUP/

[Ipiña+01] López de Ipiña D. and Lo S.. "LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing", ICOIN-15, Beppu, Japan, February 2001.

[NASA99] NASA, "CLIPS: A Tool for Building Expert Systems", http://www.ghg.net/clips/CLIPS.html, August 99

[OMG00] Object Management Group (OMG). "Notification Service Specification", June 2000.

[OMG99] Object Management Group (OMG). "The Common Object Request Broker Architecture: Architecture and Specification", October 1999.

[Paton+99] Paton N.W. and Díaz O. "Active Databases Survey", ACM Computing Surveys, Vol. 31 No.1, pp. 63-103, March 1999