EXTENDING AN ADVANCED OBJECT MODELLING ENVIRONMENT WITH VERSATILE RULE SHARING & REACTIVE CAPABILITIES

BY

LEUNG CHI CHAN

A Thesis Presented to
The Hong Kong University of Science and Technology
In Partial Fulfilment
of the Requirements for
the Degree of Master of Philosophy
in Computer Science

Hong Kong, August 1996

Copyright© by Leung-Chi Chan 1996
Authorization

I hereby declare that I am the sole author of the thesis.

I authorise The Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

[Signature]

I further authorise The Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

[Signature]
EXTENDING AN ADVANCED OBJECT MODELLING ENVIRONMENT WITH VERSATILE RULE SHARING & REACTIVE CAPABILITIES

BY

LEUNG CHI CHAN

APPROVED:

[Signature]
DR. QING LI, Supervisor

[Signature]
PROF. SAMUEL CHANSON, Associate Head of Department

Department of Computer Science
19 August 1996
For my parents, brother and sister
ACKNOWLEDGEMENTS

This thesis would not be possible without the support, advice and guidance of my supervisor Dr. Qing Li. His intellectual profundity, patience, persistence and encouragement have constantly assisted me in my research endeavour. For this, I pay my sincere gratitude to the distinguished supervisor.

I would also like to thank Dr. Kamalakar Karlapalem and Prof. Frederick H. Lochovsky for generously served on my thesis examination committee. Their effort and time spent on reading my thesis, and their valuable feedback are greatly appreciated.

To Dickson K.W. Chiu I express my gratitude for his enlightened discussions at the early stage of the ADOME project.

I would also like to thank Wilson Tang who have been very helpful in liaising with ITASCA vendors and solving database software related problems.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE PAGE</td>
<td>i</td>
</tr>
<tr>
<td>AUTHORIZATION PAGE</td>
<td>ii</td>
</tr>
<tr>
<td>SIGNATURE PAGE</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>v</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1  INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Need for Knowledge Management</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Background of Research</td>
<td>2</td>
</tr>
<tr>
<td>1.3 ADOME-II: Extending ADOME with Versatile Rule Sharing and Reactive Capabilities</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Definitions &amp; Terminology</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Organization of the Thesis</td>
<td>5</td>
</tr>
</tbody>
</table>

| vi |

2 RELATED WORK

2.1 Role Models ........................................... 7

2.2 Historical Review of Database Integrated Rule Management ........ 8
  2.2.1 Active Databases .................................. 8
  2.2.2 Rules in A.I. ....................................... 8
  2.2.3 Rules in the Object-Oriented Paradigm .................... 9

2.3 Active Object-Oriented Database Systems ....................... 9
  2.3.1 Overall Comparison with Representative Systems .......... 10
  2.3.2 Extensibility ....................................... 12
  2.3.3 Architecture ....................................... 12
  2.3.4 Accommodating Application Dynamics .................... 13
3 THE DATA/KNOWLEDGE MODEL

3.1 Role and Role Dependency .................................. 14
3.2 Rule Sharing Without Inheritance .......................... 15
  3.2.1 Problems of Rule Inheritance .......................... 16
  3.2.2 An Approach to Rule Sharing ......................... 17
3.3 A Rule Model .................................................. 18
  3.3.1 The Rule Semantics ...................................... 18
  3.3.2 Relationship among Events, Rules & Objects .......... 19
3.4 An Event Model ............................................... 21
  3.4.1 Overview of Event Types ............................... 21
  3.4.2 Composite Events ...................................... 23
  3.4.3 Modeling Events as Objects ............................ 25
  3.4.4 Layering of Event Types ............................... 26
  3.4.5 Event Propagation ...................................... 27
3.5 Rule and Event Language .................................... 29
  3.5.1 Rule Conditions & Actions ............................ 30
  3.5.2 Event Specification .................................... 32
  3.5.3 Event-Rule Association ............................... 33
  3.5.4 Rule-Object Association .............................. 34
3.6 Execution Semantics ......................................... 35
4 EXTENSIBLE APPROACH TO REACTIVE PROCESSING

4.1 Reactive Functionalities .................................................. 38
4.2 Framework for Supporting Reactive Processing ....................... 40
   4.2.1 Production Rule for Supporting Reactive Processing .......... 40
   4.2.2 Supporting Rule Triggering, Activation & Execution .......... 42
   4.2.3 Supporting Composite Event Detection ......................... 44
4.3 Pre-Defined Composite Event Expressions ............................. 44
4.4 Incremental Composite Event Detection with Production Rules ........ 46
   4.4.1 Event Detection Rules .......................................... 47
   4.4.2 Logical and Sequence Events Detection Rules ................. 49
   4.4.3 Temporal Event Detection Rules ............................... 50
   4.4.4 Cumulative Event Detection Rules ............................. 53
   4.4.5 Feasibility of Evaluating Event Detection Rules .............. 54
4.5 Extending Reactive Functionality ...................................... 55
   4.5.1 Handling Other Parameter Contexts ............................ 55
   4.5.2 Some Extra Event Operators ................................. 60
4.6 Comparison with Other Approaches .................................. 61
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>A class hierarchy of projects funded by various means</td>
<td>16</td>
</tr>
<tr>
<td>3-2</td>
<td>Entity-Relationship Diagrams Depicting the Relationships of Rules, Events</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>and Objects: (a) typical ECA rule systems, (b) ADOME-II</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>An Event Hierarchy Instance</td>
<td>26</td>
</tr>
<tr>
<td>3-4</td>
<td>Interleaving of Database Operations and Rule &amp; Event Processing Cycle</td>
<td>35</td>
</tr>
<tr>
<td>4-1</td>
<td>Event &amp; Rule Processing Cycle</td>
<td>42</td>
</tr>
<tr>
<td>4-2</td>
<td>A finite automaton for detecting events in the cumulative context</td>
<td>59</td>
</tr>
<tr>
<td>5-1</td>
<td>Functional Modules of the System Kernel</td>
<td>65</td>
</tr>
<tr>
<td>5-2</td>
<td>An Object Composition Diagram Illustrating the Implementation of Role</td>
<td>67</td>
</tr>
<tr>
<td>5-3</td>
<td>Association of Event, Rule and Objects (Showing only necessary structure)</td>
<td>68</td>
</tr>
<tr>
<td>5-4</td>
<td>Rule Agenda for ECA Rules</td>
<td>69</td>
</tr>
<tr>
<td>5-5</td>
<td>Mappings of Object in Cache and Database</td>
<td>70</td>
</tr>
<tr>
<td>5-6</td>
<td>Representing Database Objects in C++</td>
<td>70</td>
</tr>
<tr>
<td>5-7</td>
<td>Representing Database Objects in the Production Rule System</td>
<td>70</td>
</tr>
<tr>
<td>5-8</td>
<td>Query Language Modules</td>
<td>73</td>
</tr>
</tbody>
</table>
**LIST OF TABLES**

1-1 Summary of ADOME-II Reactive Features. ................................................. 4
1-2 Pre-defined Composite Event Expressions ................................................. 4
3-1 System Events and Transition Values ....................................................... 22
3-2 Summary of execution semantics .............................................................. 37
5-1 Translation of the Query Language into the Native Procedural Language .... 73
EXTENDING AN ADVANCED OBJECT MODELLING ENVIRONMENT WITH VERSATILE RULE SHARING & REACTIVE CAPABILITIES

BY

LEUNG CHI CHAN

A Thesis Presented to
The Hong Kong University of Science and Technology
In Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science

Hong Kong, August 1996

ABSTRACT

ADOME, an ADvanced Object Modeling Environment, was developed to better understand, and to more adequately accommodate, the requirements of next-generation information systems (NGISs) and their applications. While ADOME has been very effective in capturing declarative knowledge semantics, it still falls short of capturing active aspects of applications. Such active capabilities are particularly important for such NGIS applications as CSCW where the form of cooperation/collaboration is often realized as activities.

Thus, to extend the scope of ADOME so as to better support those active aspects, reactive capabilities (rules and events) are introduced into it. In this thesis, an exten-
sion to the data/knowledge model of ADOME is introduced which incorporates reactive capabilities without sacrificing the flexibility of rule sharing in ADOME. The resultant system is called ADOME-II for obvious reasons. Rules and events are modelled as first-class objects. ADOME-II has a rich set of pre-defined composite event expressions and a well-defined execution model. An approach to reactive processing is also introduced so that the reactive capabilities can be integrated into the loosely-coupled architecture of ADOME. In this approach, production rules combined with methods are used as a unifying mechanism to process rules, to enable composite events to be detected incrementally, and to allow new composite event expressions to be introduced into the system in a declarative manner. Methods for supporting new composite event expressions and handling other parameter contexts are outlined to illustrate the extensibility of the approach. Other relevant approaches are compared and it is concluded that the approach of ADOME-II is more declarative and extensible. Such extensibility is critical for the system to be able to evolve in accommodating dynamic NGIS applications’ requirements.
Chapter 1

Introduction

1.1 The Need for Knowledge Management

Relational database systems have predominated the use of database systems in the commercial community for many years. Recently, more database systems with complex data models, such as semantic data models and particularly object-oriented data models have appeared and have been gaining more and more attention in both commercial and research community. The emergence of these database systems are driven by the requirements of non-traditional applications, such as computer aided design/manufacturing (CAD/CAM), organizational information systems (OISs), decision support systems (DSSs), computer supported cooperative work (CSCW) and spatial/temporal information systems. While object-oriented data models accommodate both their structural modeling and behavioral modeling requirements adequately to a certain extent, the knowledge-level semantics, ranging from specific integrity constraints to more general problem-solving rules, are less adequately captured. Thus, to overcome such deficiencies, issues related to developing integrated/hybrid data and knowledge base systems need to be addressed as they are vital to the development of next-generation information systems (NGIS) and their applications [38].

To better understand, and to more adequately accommodate, the requirements of NGISs and their applications, an ADvanced Object Modeling Environment (abbreviated as ADOME) was proposed and developed [13]. While ADOME has been very effective in capturing declarative knowledge, it still falls short in capturing active aspects of applications. Such capabilities are particularly important for such NGIS applications as CSCW, where cooperation is often realized through activities.

Thus, to extend the scope of ADOME so as to better support these active aspects, reactive capabilities are introduced into it, resulting in the successor
ADOME-II. In ADOME-II, the data/knowledge model of ADOME is extended to incorporate reactive capabilities without sacrificing the flexibility of rule sharing in ADOME. Furthermore, the thesis also shows how the reactive capabilities can be integrated into the architecture of ADOME in an extensible manner, which is crucial for the system to be able to evolve in accommodating dynamic/unanticipated NGIS applications’ requirements.

1.2 Background of Research

The underlying approach of ADOME is based on integrating a general-purpose rule base system into an object-oriented database (OODB) system, in order to enable the integrated system to handle a wide variety of data and knowledge management requirements. The rationale behind such an approach is to allow us to investigate the feasibility and practicality for an organization to upgrade its existing OODB system by simply integrating it with an available rule base system. Such an approach should be highly cost-effective, as it has the advantages of simplicity in design and readiness for experimentation, and allows the resultant system to best utilize the individual component’s strengths in handling different types of knowledge and their associated operations.

In ADOME, the integration of the rule base with the OODB system is based on a generalized notion of roles [37, 36]. The concept of roles has been employed as an effective means for gracefully accommodating “object migration” and/or facilitating multi-perspective modelling of objects. More specifically, roles have been intended to naturally partition messages for objects so that they can receive and send different messages at different stages of their evolution/life-cycle, complementing the static classification-based approach of conventional OODB models and systems. In general, a role exhibits some features similar to a class, yet at the same time, also has several different ones. Like a class, a role has a set of attributes and methods (operations) which defines the properties and behaviors applicable to the objects playing this role. But unlike a class, it does not create or delete objects, but only includes-in or excludes-out objects from existing database classes (these objects are called role play-
ers. Further, the players of a role are not required to be of the same type (i.e., the "extent" of a role can be composed of heterogeneous objects).

In ADOME, roles have been further generalized to serve as virtual classes which can capture the dynamic interactions and integration of objects, procedures, and rules, by serving as dynamic "mediators/binders" among such conceptually distinct constructs. In particular, a role can have a set of IF-THEN production rules which are created/defined within the role and are applicable to any of the role players [37]. This extension allows the role to capture a wide range of data/knowledge semantics and integrity constraints, thereby facilitating the desired integration of data and knowledge management for NGIS applications [38].

1.3 ADOME-II: Extending ADOME with Versatile Rule Sharing and Reactive Capabilities

Apart from that of ADOME, the specific salient features of ADOME-II are as follows:

- **Versatile rule sharing.** Rules can be shared across a class hierarchy, without the knowledge of their triggering events. Rules can be clustered through their association with roles; the cluster of rules can then be inherited by players of the roles.

- **Expressive rule language.** Conditions and actions of a rule can be arbitrary database queries, thus facilitating complex knowledge to be expressed.

- **Extensible reactive processing.** The thesis shows how a rich set of reactive capabilities can be supported in an extensible manner and in a loosely-coupled architecture that has been used to support production rules in ADOME.

To provide readers a preview of the reactive capabilities in ADOME-II, a summary of them is given in Table 1-1 and Table 1-2.
Table 1-1 Summary of ADOME-II Reactive Features.

<table>
<thead>
<tr>
<th>Category</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules</td>
<td>Rules are in the form of if...then...else. They are triggered by events and can be shared among entities in the database.</td>
</tr>
<tr>
<td>Events</td>
<td>System events, abstract events, logical events, temporal events, and cumulative events are supported. Composite events are detected in the most recent parameter context.</td>
</tr>
<tr>
<td>Conditions</td>
<td>Arbitrary database queries are allowed and event parameters can be referenced in a condition.</td>
</tr>
<tr>
<td>Actions</td>
<td>Actions can be specified for both true or false evaluation result of a condition. Arbitrary database operations are allowed and they may generate events.</td>
</tr>
<tr>
<td>Coupling Modes</td>
<td>Immediate and deferred event-condition coupling modes are supported.</td>
</tr>
<tr>
<td>Rule Priority</td>
<td>The priority of a rule is a numerical value which can be the result of a method execution.</td>
</tr>
</tbody>
</table>

Table 1-2 Pre-defined Composite Event Expressions

<table>
<thead>
<tr>
<th>Event Expressions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{And}(E1, \ldots, En)$</td>
<td>Conjunction of $E1$ through $En$.</td>
</tr>
<tr>
<td>$\text{Or}(E1, \ldots, En)$</td>
<td>Disjunction of $E1$ through $En$.</td>
</tr>
<tr>
<td>$\text{Seq}(E1, \ldots, En)$</td>
<td>Occurs when a sequence of events $E1$ through $En$ is found.</td>
</tr>
<tr>
<td>$\text{Not}(E1, E2, E3)$</td>
<td>Non-occurrence of $E2$ within the closed time interval defined by $E1$ and $E3$. It is signaled at the occurrence of $E3$.</td>
</tr>
<tr>
<td>$\text{P}(E1, T, E3)$</td>
<td>Occurs in the frequency of $T$ between the closed time interval defined by $E1$ and $E3$.</td>
</tr>
<tr>
<td>$\text{A}(E1, E2, E3)$</td>
<td>Occurs for each $E2$ occurrence within the closed time interval defined by $E1$ and $E3$.</td>
</tr>
<tr>
<td>$\text{A*}(E1, E2, E3)$</td>
<td>Occurs zero or one times for all $E2$ occurrences within the closed time interval defined by $E1$ and $E3$.</td>
</tr>
<tr>
<td>$\text{Every}(E1, TP, E3)$</td>
<td>Occurs every time when the time pattern is matched within the closed time interval defined by $E1$ and $E3$.</td>
</tr>
<tr>
<td>$\text{At}(TP)$</td>
<td>Occurs at the absolute time specification.</td>
</tr>
<tr>
<td>$\text{Disjoint_Seq}(E1, \ldots, En)$</td>
<td>Non-overlapping version of $\text{Seq}(E1, \ldots, En)$.</td>
</tr>
<tr>
<td>$\text{After}(E1, T)$</td>
<td>Occurs at the time $T$ after the occurrence of $E1$.</td>
</tr>
<tr>
<td>$\text{Within}(E1, E2, T)$</td>
<td>$E1$ and $E2$ must occur within the time period of $T$.</td>
</tr>
</tbody>
</table>

a. Here $E1, \ldots, En$ are events, $T$ is a string of time period, $TP$ is a string of time pattern.

1.4 Definitions & Terminology

To facilitate reading later chapters of the thesis, there are some definitions and termi-
nology that need to be presented. Without additional qualification, the term object always refers to a database object in ADOME-II. In contexts where it is required to refer to an object not particular to ADOME-II's data/knowledge model, the terms class instance or ordinary object will be used instead. An object is, therefore, always more general than a class instance or ordinary object. The meta-data of a role are grouped into a database class and referred to as role class. A role instance is an instance of the role class. The term role, when appears individually, denotes either a role class or a role instance. Its actual meaning can be determined from the context. A role player is an object playing a role. The term entity may refer to an object, a role class, or an ordinary class, and it may be qualified by the word database without changing its meaning.

In traditional ECA rule models, a rule is an event-condition-action triple. In ADOME-II, a rule, unless it is preceded by the word ECA, refers to a condition-action pair which can be triggered by events. To distinguish from rules in ADOME-II, the term production rule refers to a data-driven condition-action rule found in forward-chaining rule systems. Every event has a type, which is referred to as event type. The class that implements an event type is called an event class. In most cases, they are inter-changeable. The term event occurrence denotes an instance of an event class. The term event, when used individually, means either an event type or an event occurrence. Its actual meaning can be determined from the context. When an event occurs, it may trigger rules. A triggered rule becomes activated when it is ready to be fired or executed but has not yet been so. Finally, the process of rule execution consists of condition evaluation and action execution.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews relevant work on object-oriented models and active databases, where their differences with ADOME-II are highlighted and discussed. Chapter 3 introduces the data/knowledge model of ADOME-II; in particular, the role model, rule and event model, the rule and event language, and execution semantics of rules and events are described. In chapter 4, the
operational aspect is addressed where an approach to supporting rule execution and composite event detection is described; it also illustrates how extensible the approach is. Chapter 5 reveals the architecture and implementation details of ADOME-II. Finally, chapter 6 concludes the thesis and explores several worth-while directions for future work.
Chapter 2

Related Work

ADOME-II draws mainly on two broad fields in database research: object-oriented models and rule processing. This chapter compares some of the relevant topics in the two fields with that of ADOME-II. In the first section, some comparison on role models is discussed. Section 2.2 briefly describes the history of rules in active database and in expert systems as well as the use of the object-oriented model in modeling rules. Section 2.3 compares some representative active object-oriented database systems with ADOME-II in specific aspects.

2.1 Role Models

The notion of roles is actually not new in database models and systems. There are a number of similar works on roles, e.g. [1, 2]. In the context of object-oriented databases, roles have been used in supporting object evolution and migration, as well as conceptual clustering modeling [39] and multi-perspectiveness. Some comparisons can be found in [36], which are not repeated here.

A more recent work on roles not mentioned in [36] can be found in [30], where extension to the object-oriented programming language Smalltalk with roles is introduced. Objects can acquire/abandon roles at the instance level dynamically. Role types are organized in a hierarchy similar to a class hierarchy but a subtype in a role hierarchy does not inherit properties from the supertype. The difference of this model with the role model in this thesis is that [30] imposes the restriction that role types are modeled as a subclass of their applicable classes. This restriction effectively fixes the roles that can be played by objects of a class statically, which lowers the flexibility gained from the use of roles. Also the notion of rule is not present their model.
2.2 Historical Review of Database Integrated Rule Management

Rule management has been one of the most important extensions in recent years to mainstream database technologies. Rules not only can specify active capabilities but also can capture general knowledge in a declarative way. Expert databases, deductive databases and active databases all deal with rules in addition to the basic relational model and more recently the object-oriented models. These are enabling technologies for building advanced next-generation information systems (NGISs). Thus, their histories are briefly reviewed.

2.2.1 Active Databases

Active databases refer to databases that perform certain operations automatically in response to certain events occurring or certain conditions being met. The concept of activeness can be traced back to triggers in CODASYL and relational databases where they are specified in the form of ON <command> ... . Similar active capabilities can be found in AI systems in the form of actors, daemons, active objects and procedural attachment to slots of frames. These earlier forms of active capabilities are very simple in the sense that the conditions to respond to are often a single command or operation. More powerful and general specification of such active capabilities are realized later through the use of Event-Condition-Action (ECA) rules introduced in a pioneering project HiPAC, which dominates current active database systems. An ECA rule essentially specifies that when the Event occurs, the Condition is evaluated and if the result is true, the Action is executed.

2.2.2 Rules in A.I.

Historically, rule management has been the domain of expert systems (either forward chaining or deductive) [42], which are stand-alone main memory programs with no interaction with database systems. Rules in these systems do not have the explicit control over the triggering of rules as do ECA rules, and are more declarative than ECA rules. On the other hand, modelling such active capabilities as activities, which
can be readily done in ECA rules, cannot be easily done in these expert systems. Apart from the differences in the utility of rules in expert systems and active databases, there is a problem with expert systems. The problem is that there is no means to share objects (facts) and to store objects persistently. Consequently, there have been efforts to extend these systems with database interfaces so that objects can be managed by the DBMS. However, the consistency of rules (which are managed by expert systems) and data (managed by a DBMS) is still not guaranteed since they are managed by different systems.

2.2.3 Rules in the Object-Oriented Paradigm

Recently, object-oriented models have received much attention and have been utilized in various fields, such as conventional programming languages, knowledge representation in A.I. and databases (for an introduction to object-oriented databases, see [34, 35, 17, 32]). Compared with relational database systems, OODB systems support richer data structuring mechanisms, hierarchical and general relationships between objects, and better encapsulation. The use of object-oriented data models thus allows more application logic to be captured in databases, which not only reduces duplication and encourages code reuse, but also provides a promising tool for incorporating rules into the system. In particular, rules and their actions may be stored and managed as objects and their methods [4, 16]. Thus, an OODB system is an excellent enabling technology for the unified management of rules, data and procedures.

2.3 Active Object-Oriented Database Systems

Research in active database systems has greatly enhanced the understanding of active functionalities. Many prototypes have been built and some active functionalities are incorporated into many commercial database systems. More recently, active object-oriented database systems have received much attention and quite a number of prototypes are built/being built which incorporate very rich active functionalities. Due to the number of active object-oriented database systems, an exhaustive comparison would be beyond the scope of this thesis. Therefore, a selection of representative
active object-oriented database systems are compared in specific aspects. A small portion of the comparison below can be found in [40], which contains comparisons on other aspects as well.

2.3.1 Overall Comparison with Representative Systems

ODE [26, 28, 29, 27] is an object-oriented active DBMS based on the C++ object paradigm. Rules (called triggers in ODE) are Event-Action rules where event specifications can contain predicates serving the purpose analogous to conditions in ECA rules. The rules are defined in and tightly integrated with classes. Since rules are not first-class objects, they are susceptible to several shortcomings. Firstly, their existence depends on the class. In other words, rules can only be written when their applicable classes are defined and they vanish when the classes not longer exist. Secondly, rules cannot be generally shared among classes; duplication of rules is required to do so. Although rules can be inherited, such inheritance is not possible for classes not on the same inheritance path. Moreover, such inheritance is also not desirable as argued in more detail in section 3.2 of chapter 3. Together, these two shortcomings discourage the reuse of knowledge expressed in the form of rules, which is an important feature for knowledge intensive applications. Lastly, modification to rule definition would not be possible without some schema evolution facilities, which are considered as expensive. In contrast, the rules in ADOME-II are first-class objects, thus avoiding the shortcomings mentioned above.

Sentinel [3, 10, 11] is a follow-on project to HiPAC, the distinct features of which are its advocacy of external monitoring viewpoint and a rich set of composite event expressions. The notion of external monitoring viewpoint is essentially to adopt a loose association of rule (condition-action), its triggering events and its applicable objects. Thus, independence of the three components (i.e. rules, events and objects) is enhanced, which results in higher expressiveness and stronger extensibility of the ECA rules. The rule and event model proposed in this thesis not only achieves such independence of rules, events and objects, but the issue of rule sharing and event sub-typing are also addressed. Also, the set of composite expressions here is not only
comparable to that of Sentinel, but can be extensible (cf. chapter 4) to accommodate unanticipated application requirements.

REACH [5, 6, 7] is also a follow-on project of HiPAC focusing on integration of heterogeneous repositories around an active-object mediator system to support time constrained, complex applications. Since their ECA rule definition, including triggering events, is done at compile time, it severely limits the usability of the system for dynamic, ad-hoc interactions, which are necessary for many NGIS applications. Other issues such as composite events and relationship of rules with objects are not fully-addressed.

NAOS [15] is another active object-oriented DBMS built using the O₂ database system. It is similar to the event model proposed in this thesis in the treatment of event subtyping. In NAOS, an event on a class 𝐶, would trigger rules associated with events on the superclasses of 𝐶. Thus, in NAOS's term, the event type is inherited along the class hierarchy. In comparison, the event model proposed in this thesis is a generalized model of that of NAOS. Specifically, not only is event inheritance supported, but general event subtyping according to event semantics is supported. For instance, the event “update on employee’s salary” is made a subtype of the event “update”. Thus, rules associated with “update” can be triggered by the event “update on employee’s salary”. Furthermore, the notion of dynamically-bound event types is uniquely introduced to the event model of ADOME-II. More details on the event model can be found in section 3.4 of chapter 3. A final note on NAOS is that at the time of writing the thesis, issues on composite events are not addressed by NAOS.

SAMOS [24, 25, 23, 22] sets itself apart from other active object-oriented DBMS’s with its rich composite event expressions and deep integration of active concepts with a canonical object-oriented data model. Rule inheritance is supported by using event inheritance similar to that of NAOS. Thus, the arguments made about event inheritance in NAOS also apply to SAMOS. However, rules in SAMOS can only be defined or modified at compile time and before the database has been populated, which is not acceptable in dynamic environments.
In ADAM [20] and later EXACT (briefly described in [19]), rules are implemented as first-class objects. They can be defined and modified at run time. However, sharing of rules across the class hierarchy is not possible. Also for composite events, only disjunctive and conjunctive events are supported.

2.3.2 Extensibility

Although in [18], the "essential features" of active database systems for supporting different classes of applications are discussed, the details of reactive capabilities cannot be enumerated since applications within a class still differ. Thus, it is still not clear what exactly the features of an active database system should be. This uncertainty can be alleviated if an active database system is made extensible. By employing an object-oriented data model, a database system benefits from the extensibility inherent in object-oriented models. However, the extensibility of reactive components of active database systems are not well catered to by using object-oriented models alone. All the active object-oriented database systems mentioned above have not emphasized the extensibility of their reactive capabilities. In contrast, the work in this thesis introduces an approach to reactive processing so that reactive capabilities can be made more extensible. More details on the approach can be found in chapter 4 and comparison with other approaches can be found in section 4.6.

2.3.3 Architecture

All the active database systems mentioned above are built either from scratch (ADAM, EXACT), by employing some extensible database engines/toolkits (ODE, REACH, Sentinel, SAMOS), or by using a passive OODBMS (NAOS). The first two approaches require significant development effort and time, and are system-specific. The last approach, although it reduces duplication of DBMS functionalities, still requires significant efforts in developing reactive capabilities. In contrast, the architecture of ADOME-II is based on a loose coupling of a production rule system and a passive OODBMS. Instead of duplicating their functionalities, they are re-used as
much as possible, thus minimizing duplication of effort and enhancing the portability of the resulting system. The architecture design can be found in chapter 5.

2.3.4 Accommodating Application Dynamics

All the aforementioned active object-oriented database systems are focused on enabling reactive capabilities and integration with object-oriented concepts. Very little has been done on accommodating application dynamics. Specifically, several active database systems mentioned above do not allow rules to be dynamically defined and modified. Those that do allow dynamic definition and modification of rules are inflexible in supporting association of rules, triggering events and objects. Their inflexibility stems from the straight adoption of the ECA rule paradigm in which rules are triggered by some events on some objects. Such a paradigm does not have the explicit notion of rule ownership, which describes which rules are applicable to which objects. Consequently, knowledge sharing (in the form of rules) among objects is not supported at a high abstraction level. In other words, those ECA rules are considered at the operational level.

In contrast, the rule and event model proposed in this thesis allows flexible and explicit association of events, rules and objects. Such an association also enables rules to be shared without the knowledge of their triggering events, thus separating low-level knowledge (events) from high-level problem-solving knowledge (rules). Furthermore, through the use of roles, rules can be organized into modules and shared as a single unit without restrictions imposed by object classification.
Chapter 3

The Data/Knowledge Model

Rules in ADOME are data-driven IF-THEN rules. To better handle active aspects of NGISs and their applications, reactive capabilities are introduced into ADOME [14]. In this chapter, we introduce a rule model that incorporates such reactive capabilities. The rule model enables the relationships between rules and objects to be explicitly expressed in the model. As a result, rules can be shared among objects more directly than in traditional ECA rule models, and operations based on such relationships can be readily performed. Moreover, the redundancy of specifying explicit control by providing events is eliminated whenever possible.

3.1 Role and Role Dependency

The role model described in [36] circumvented some undesirable restrictions of canonical object-oriented data models, namely the static nature of class-based inheritance and monolithic classification of objects. In particular, by introducing the notion of role, it allows objects to acquire additional properties and behaviour outside the class inheritance hierarchy. Moreover, a role can have a set of rules associated with it. A rule can be associated to either all role instances of a role or to a subset of the role instances. Furthermore, to enhance the flexibility of roles, role type inheritance is introduced, which is differentiated from class inheritance through the notion of noble/base roles (as opposed to sub/super-class). Consequently, a player of a role does not automatically become a player of the corresponding base role; instead, an object acquires necessary roles explicitly. This avoids the rigidity imposed by the traditional class inheritance (is-a) hierarchy. On the other hand, without the is-a relationship between roles, conceptually dependent role instances cannot be modelled. Therefore,
we introduce the notion of role dependency to model the dependency relationships among dependent roles of an object.

In ADOME-II, the existence of an object's role can be made dependent on another role of the object. The term dependent role is used to denote the role whose existence depends on another role of the same object. Thus, to model the requirement that a person can be a research assistant if he/she is a postgraduate student of the university, the role research assistant can be made dependent on the role postgraduate student of the same person. Corresponding to dependent role, an independent role is a role which exists independently of other roles. Naturally, independent roles of an object vanish when the object is destroyed.

Access to a particular role's properties can be achieved by specifying a role path expression, which is an ordered sequence of role names, in a query. To query an independent role, the role path expression consists of only the name of the independent role itself. The use of role path expressions forces the current roles of an object to be exposed, reducing the chance of errors that are due to incorrect perception of current roles of an object.  

3.2 Rule Sharing Without Inheritance

The need for rule sharing can be observed from many real world applications. For instance, different types of students (e.g. undergraduate and graduate) are subject to university rules on studying and conducting. Employees are subject to rules on promotion and employment. For inventory applications, different active rules may be employed to track the level of inventory of different products. For DBMS internal applications, active rules may be used to gather statistics on the access of different objects. In this case, statistical rules may be shared among objects (possibly of different classes).

---

1. We note that, in the environment where objects take-up/relinquish roles dynamically and the dependencies among roles may change dynamically as well, such errors can occur frequently.
An important characteristic of rule sharing in active object-oriented databases is that objects that share the same rule do not always belong to the classes that are related by is-a relationships. Rule inheritance cannot solve the problem of rule sharing entirely. For some cases, the common rule has to be duplicated, one for each class of objects, which is undesirable. Therefore, a more flexible rule sharing method that transcends the limitation of inheritance is required and this is the subject of this section. We illustrate the problem of rule inheritance, and then introduce a rule sharing method that is more general and flexible than inheritance.

3.2.1 Problems of Rule Inheritance

Because of their similarity with methods, rules defined on a class are sometimes shared by derived classes by inheritance like other class properties. However, there are a number of problems with class-based inheritance of rules. These problems mainly stem from unique identity of rules and dynamic association of rules to classes. A situation where class-based inheritance is particularly problematic is when a rule is to be shared by some, but not all, siblings of a parent class. A straightforward approach is to duplicate for each applicable class, the rule to be shared. This approach, however, introduces difficulties in maintenance.

Another possible approach, based on class-based inheritance, is to define all potentially applicable rules in the parent class and let individual derived class inhibit non-applicable rules. Using Figure 3-1 as an example, a rule completion-deadline,

![Diagram](image)

**Figure 3-1** A class hierarchy of projects funded by various means

which may specify, for instance, the length of a project, or remedial actions to be carried out when a project deadline is missed, is applicable only to UGC (University Grant Council) or government funded projects. With this approach, the rule would be defined in the class R&D-Project, and those derived classes (Self-Sustained-Project, Industrial-Funded, Government-Funded) could inherit it.
and Industrial-Funded) not subject to the rule inhibit it. The problem with this approach is that encapsulation of a class is violated since whenever a rule is applied to some classes (UGC-Funded and Government-Funded), those non-applicable sibling classes (Self-Sustained-Project and Industrial-Funded) are forced to acknowledge the new rule.

Yet another possible approach, based on class-based inheritance, is to create a parent class only for classes UGC-Funded and Government-Funded, which captures their common properties. Then, both classes can share the rule completion-deadline by defining it in their immediate parent class. This approach is not feasible for two reasons:

1. Creating a non-leaf class dynamically is not regarded as a standard functionality of ODBMS, thus limiting the portability of the model. Furthermore, this kind of schema evolution is likely to be costly, not justifiable by merely the need to enable more flexible rule sharing.

2. It may lead to proliferation of such thin classes which only serve the purpose of a placeholder for common rules.

3.2.2 An Approach to Rule Sharing

Consequently, an alternative way of rule sharing; which is termed as by association (the query language for this operation is given in section 3.5.4), is adopted in ADOME-II. By association, each class individually associates with its desirable rules. With this approach, no duplication of rules results and encapsulation of a class is preserved. The use of association is not to invalidate some of the benefits of class-based rule inheritance, but rather to enable more flexible rule sharing in a dynamic environment. In conclusion, there are many viable ways of designing and applying rules but the extent of their side-effects varies. In general, it is claimed in this thesis that rule sharing by class-based inheritance is better suited for stable environments with a well-understood schema, whereas rule sharing by association is more suitable for prototyping and dynamic environments.
3.3 A Rule Model

To enable this kind of rule sharing, a rule model is proposed which allows rules to be shared without being restricted by the class hierarchy. This rule model is based on the ECA paradigm. The rationale of adopting such a paradigm is that there are applications which involve different kinds of, and hence different ways of handling, application rules. Some of these applications are better modelled by the Condition-Action (or "data-driven") paradigm while others by the ECA (or "event-driven") paradigm. Most active database systems adopt the ECA rule paradigm. While ECA rules provide explicit control of rule firing and a more general rule modelling ability, they have the drawback of being more difficult to program than condition-action rules because in the ECA rule paradigm, the burden of determining when to evaluate the rule lies with the users. It is, therefore, desirable to combine the usefulness of the two paradigms while minimizing their drawbacks. The rule model proposed here alleviates some of the need to determine the triggering events of a rule while allowing it to be shared like an ordinary condition-action rule.

3.3.1 The Rule Semantics

In the rule model proposed here, a rule is a Condition-Action pair, which is loosely associated with its triggering events and applicable entities; each of them can exist independently. A rule is activated only when the three components: its triggering events, the rule (condition-action pair) itself and its applicable entities, are associated or bound together explicitly. This is different from most of the ECA rule systems where an event and a condition-action pair activate an ECA rule, and objects are implicitly associated with events. Since a rule here is loosely associated with its triggering events, it is free from knowledge of them when it is shared among entities. In other words, its triggering events are omitted when a rule is shared.

This raises the question of whether generation of triggering events by syntactical analysis of a rule can be used to alleviate the burden of specifying a rule’s triggering events. For a few classes of rules, such as rules that enforce integrity constraints, the triggering events of the rules can be generated syntactically from their conditions,
thus enabling ECA rules to be used as condition-action rules. However, syntactical generation of triggering events is not a general solution to alleviate the burden of triggering event determination. There are many other rules for which syntactic generation of triggering events is not straightforward or even possible. Examples of such rules are rules triggered by abstract events; rules that do not have conditions (or their conditions are always true); and rules that invoke methods in their conditions in which some attributes or objects are referenced. For these kinds of rules, the triggering events usually must be determined manually. With the rule model proposed here, all these kinds of rules can be shared without knowledge of triggering events. The separation of the event type semantics from their applicable entities facilitates rule writing and sharing by allowing the rule end-users to concentrate solely on the semantics of rules. Consequently, the processes of rule writing and rule application are separated, which enables the burden of specifying events to be shifted to more expert rule writers.

3.3.2 Relationship among Events, Rules & Objects

The differences in inter-relationships of events, rules and objects of ADOME-II and other systems are intuitively illustrated in Figure 3-2. Note that although in ADOME-II (cf. Figure 3-2b) rules can be associated with either roles or class instances, associating rules with roles would benefit from the strength of roles. Specifically, by associating a set of rules with a role, objects from any class can choose to be subject to the set of rules by playing the role. Therefore, it is more preferable to associate rules with

![Diagram](image-url)
roles rather than objects. However, in order to compare our model with traditional ECA rule models, only the association of rules with objects is considered. Nevertheless, the discussion below is also applicable to the rule-role association.

It should be noted that many ECA rule systems only support a 1:1 relationship mapping between events and rules. Even when 1:M mappings are allowed between events and rules, the relationship cardinality between objects and rules is still 1:M at most. Our model is thus more flexible in this aspect. To see this, suppose a set of events \( \{e_1, e_2, \ldots, e_n\} \) is bound to a rule \( r \). This is functionally equivalent to the following statement:

\[
(e_1 \text{ or } e_2 \text{ or } \ldots \text{ or } e_n) \text{ triggers } r.
\]

In a 1:M event-rule mapping model (Figure 3-2a), the above relationship can be expressed by associating a composite event \( e = (e_1 \text{ or } e_2 \text{ or } \ldots \text{ or } e_n) \) with \( r \). However, this is undesirable as any new association/disassociation of events with the rule \( r \) requires the knowledge as well as modification of \( e \). Since rules can be shared by multiple database entities or users, it would be inefficient and infeasible for each sharing of the rule \( r \) to involve a checking and modification of \( e \). In our model, an event can be bound to \( r \) without the knowledge of other events, allowing more decentralized sharing of rules. Example 3-1 illustrates from the rule-sharing perspective the usefulness of such a M:N event-rule mapping model.

**Example 3-1** Consider the following rule which is typically found in an academic office environment. The rule performs some preprocessing on the form a junior clerk receives and dispatches it to relevant people. The state is an attribute of the role Clerk that denotes the state of a clerk, which can be 'ready', 'on-leave', 'having-lunch' etc. The rule can be triggered by abstract (or explicit) events like 'Reimbursement-Form-Arrived', 'Course-Registration-Form-Arrived' and so on. Senior clerks may request the junior clerk to perform preprocessing on various kinds of forms without knowing what other forms the junior clerk has to preprocess, since this is often irrelevant to the senior clerk who makes the request.

1: Create Rule PreProcess-Form
2: if (this.state = 'Ready' and this.year-of-experience < 1) then
3: begin
4: Form-PreProcess(event.form_sender, event.form_destination);
5: Dispatch-Form(event.form_sender, event.form_destination);
end;

-- Apply the rule to clerk Smith
Attach PreProcess-Form to ?c in Clerk where (?c@base.name = 'Smith');

-- Share the rule among multiple events.
-- By using On..Check statement, the events Reimbursement-Form-Arrived
-- and Course-Registration-Form-Arrived are bound to the rule
-- PreProcess-Form.
On Reimbursement-Form-Arrived Check PreProcess-Form;
On Course-Registration-Form-Arrived Check PreProcess-Form;
...

Elaboration of the above statements follows:

Line 1-6: Defines the intension of the rule.

Line 9: Defines the object (playing the role Clerk) applicable to the rule. In this example, we are assuming that there is only one clerk named 'Smith'. In general, if it is desirable to have arbitration among multiple junior clerks on receiving a form, it can be done by embedding an arbitration method in the rule condition and use the rule's event parameters as arbitration information. Such higher level semantics, however, are not the major concern here.

Line 15-16: Assigns more tasks (or forms) to be handled by the rule.

3.4 An Event Model

Since rules in ADOME-II can be considered a variant of traditional ECA rules, they are also triggered by events. Thus, we describe in this section the event model that enables rules to shared in a flexible manner.

3.4.1 Overview of Event Types

Events can be classified into three categories:

- *Events as results of database operations*. Due to the finite duration of database operations, each type in this category can be classified into *before* and *after* subtypes. An occurrence of a *before* event type is generated at the beginning of a database operation whereas that of an *after* event type is gen-
erated at the end of a database operation. Note that the *before* and *after* variants are meaningful to database operation events only, but not to the abstract events and composite events (since such semantics are unclear).

- **Abstract events.** These are user-defined events and raised by applications explicitly.

- **Composite events.** These are events that are composed by an extensible set of event operators (as described in chapter 4).

Database operations usually change database states, and the values affected by these operations are referred to as *transition values*, or sometimes *deltas*. Since they are often referred to in rules and are associated with database operations, it is natural to associate them with their resulting event occurrences, so that they can be subsequently accessed through querying the corresponding event occurrences. Each database operation results in a different transition value. The transition value for each database operation is summarized in Table 3-1. How these transition values are collected and stored are subjects of chapter 5, where architecture and implementation are discussed.

**Table 3-1 System Events and Transition Values**

<table>
<thead>
<tr>
<th>Database Operations</th>
<th>State of Operation</th>
<th>Transition Value Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Creation</td>
<td>Before</td>
<td>None.</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>OID of the new object.</td>
</tr>
<tr>
<td>Object Deletion</td>
<td>Before</td>
<td>OID of the object going to be deleted.</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>OID of the object just deleted.</td>
</tr>
<tr>
<td>Attribute(^a) Read</td>
<td>Before &amp; After</td>
<td>None.</td>
</tr>
<tr>
<td>Attribute Write</td>
<td>Before</td>
<td>The value to be assigned to the attribute.</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>The value of the attribute before the opera-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tion.</td>
</tr>
<tr>
<td>Set(^b) Insertion</td>
<td>Before &amp; After</td>
<td>The element (to be) inserted to the attribute.</td>
</tr>
</tbody>
</table>
| Set Deletion              | Before & After     | The element (to be) deleted from the attrib-
|                           |                    | ute.                                       |
| Role Instance Creation    | Before             | OID of player of the role instance.         |
|                           | After              | OID of the new role instance.               |

22
Table 3-1 System Events and Transition Values

<table>
<thead>
<tr>
<th>Database Operations</th>
<th>State of Operation</th>
<th>Transition Value Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role Instance Deletion</td>
<td>Before</td>
<td>OID of player of the role instance.</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>OID of the role instance just deleted.</td>
</tr>
<tr>
<td>Method Execution</td>
<td>Before &amp; After</td>
<td>The actual parameter of the method invocation.</td>
</tr>
</tbody>
</table>

a. For both atomic and set-valued attributes
b. For Set-valued attributes only

Orthogonal to the above three categories are event types that are object-/class-associated and/or object-/class-disassociated. For this we have the following three additional categories:

1. **Dynamically-Bound Events (DBE)**: These event types are object-independent: they do not pertain to any objects or classes/roles. However, event occurrences of any of them are always associated with objects, which may possibly belong to different roles/classes. Consequently, event occurrences of this category trigger associated rules defined on the triggering object of the occurrence.

2. **Instantiated Events (IE)**: To support inter-class rules, class-dependent event types are used. An instantiated event type is an event type on instances of a class, e.g. the event [update employee.salary] is an instantiated event type on class employee. An event occurrence of this category, unlike that of DBE events, triggers all of its associated rules on their associated objects.

3. **Untargeted Events (UE)**: These are events that do not originate from any database objects. The event occurrences of this category trigger rules in the same way as IE events.

### 3.4.2 Composite Events

A primitive event is the basis of reactive modeling, but it is the composite event that allows complex situations to be specified. Composite events are composed by event operators (described later in chapter 4). To capture knowledge-level semantics in general (other than just as integrity and consistency maintenance), it must be possible to
specify a rich set of composite events. Here, the set of composite events are not fixed; through an extensible approach, the set of composite event operators can be extended without affecting other database components and applications.

For the purpose of rule sharing, the categories of events have been introduced in the previous section. To determine the category of a composite event, the following three rules can be applied recursively to its constituent events:

1. If the composite event has a DBE constituent event, the composite event is also a DBE event. The triggering object of the composite event is that of the DBE constituent event.

2. If there is no DEB event and there is an instantiated constituent event, the composite event is an instantiated one. The triggering object of the composite event is that of the DBE constituent event.

3. If there is neither a DBE constituent event nor an instantiated constituent event, then the composite event is an untargeted event. There is no triggering object for the composite event.

Due to the nature of DBE composite events, they may generate excessive, semantically meaningful occurrences of these events if appropriate constraints are missing. As such, detection of DBE composite events must preserve, in addition to other correctness criteria, the following invariants:

- **Object-correspondence invariant:** This invariant specifies that the triggering objects of all DBE constituent events of a composite event must be identical. The constituent events are, therefore, related together by the same triggering object, and a composite event obeying this invariant is said to have a correspondence with a single object—the triggering object.

- **Object-isolation invariant:** The effects of an event occurrence on an object must be independent from other event occurrences on another object. For example, in identifying the most recent event occurrence of a DBE event type $E$ on $o2$, an occurrence $e$ of $E$ on object $o1$ should not affect the occurrence $e'$ on object $o2$ being identified as the most recent occurrence of $E$. 

24
• **Minimality invariant**: Composite events should be signaled only if they can trigger rules or potentially induce composite event occurrences. Redundancy of event occurrences may occur in a number of situations: when the triggering object of the occurrence is not attached to any rule, or when the event is disabled. Although violation of this invariant does not invalidate the correctness of event detection, it does impose unnecessary overhead on event detection and storage.

How these invariants are enforced is a subject of chapter 4, where the detection of composite events is discussed.

### 3.4.3 Modeling Events as Objects

The benefits of modeling events as objects have been well-discussed in [3]; by modeling event occurrences as objects, they can be queried and manipulated uniformly as ordinary objects; moreover, the classes of event occurrences serve as indices, which speed up processing of events. In this section, we propose an object-oriented event model that can accommodate all the features of events described in previous sections. The distinct characteristic of the model is the structuring of event types, such that a large amount of semantics is captured in the structure, reducing the amount of behavior specification required.

Event occurrences are modeled as objects and classified into event types, as illustrated in Figure 3-3. Each oval in the figure represents an event type on its own. Therefore, the before oval is an event type and so is the Insert oval. Both of them can have their own event occurrences. Parameters of events can be objects triggering the event occurrence, the actual parameters of methods, and/or transitional values (such as the deleted objects from some object deletion events). Parameters are stored as properties of event occurrences.
3.4.4 Layering of Event Types

To facilitate understanding of the usage of event types, event types are organized into four layers as depicted in Figure 3-3. Each layer of events has its own unique purpose:

1. **Internal layer**: This layer contains event types that are hidden and protected from users; they are for system implementation and internal monitoring only, not for general application.

2. **Operational layer**: In comparison, events in this layer capture the semantics of database operations.

3. **Application layer**: The events in this layer are the ones used primarily in applications. A large portion of events in this layer are those generated by database operations on object properties, such as updates of attributes or method executions. For instance, the oval with name Salary in Figure 3-3
represent the event that will be generated after the attribute Salary of some classes/roles is updated. Abstract and composite events compose the remaining portion of events in this layer.

4. **Instantiated layer.** All occurrences of an event type in this layer originate from a specific class. For instance, the event type Employee (cf. Figure 3-3) corresponds to the event of updating the attribute Salary of instances of class Employee. Since classes are subject to inclusion polymorphism, that is, each instance of a class is also an instance of its base class, an event from an class instance is also regarded as an event from its base class instance. Thus, to support such semantics on events, event types in this layer are organized in parallel with the hierarchy of their corresponding classes.

The stacking of event layers is in such a way that the more general events are placed above less general ones. There are certainly alternative stacking orders of event layers and organization of event types, depending on the interpretation of the meaning of "general". Here, the operational layer and application layer are placed above the instantiated layer for two reasons:

1. The former two layers of events carry definitions and semantics that are more relevant for reuse, and

2. by placing the instantiated layer at the bottom of the event type hierarchy, application event types are insulated from changes in the class hierarchy. That is, creation or deletion of classes do not affect the event types in upper layers.

### 3.4.5 Event Propagation

Since event types are modelled as classes and organized in a hierarchy, event types are also subject to semantics of class subtyping. Event subtyping (for primitive event types) has the same semantics as class inheritance: an instance of an event type \( e \) can be regarded as that of any supertype of \( e \). By organizing event types into a hierarchy, not only can the event definitions be re-used but also event propagation can be facili-
tated and enabled along the hierarchy. To support the semantics of event subtyping, an event occurrence propagates to all of its superevent types by default. That is, a single database operation may generate multiple event occurrences. This is convenient in cases where subset/superset event type relationships are desirable. As an example, it is sufficient for a statistical rule that gathers the statistics of every database update operations to be bound to the event type [update] as well as relevant objects in order to achieve the purpose. Once the binding is completed, every database update on a relevant object can then trigger the statistical rule. There is no need to bind the rule to events of all kinds of update operations, as this would be inefficient and infeasible. For user-defined primitive event types, the user can also choose to disable the propagation of event occurrences whenever such an action is undesirable.

The exceptions to event propagation are composite event types since they do not have explicit subtype relationships with each other. This is because they are always derived (directly or indirectly) from primitive event types. (Thus their triggering is also determined by their constituent event types.) In fact, specifying the super/subtype relationship explicitly between two composite event types can lead to some "side-effects", namely redundancy and inconsistency. For example, if we specify a composite event type \( E1 \) to be a supertype of another composite event type \( E2 \), then due to the subtyping relationship, an occurrence of \( E2 \) will trigger an occurrence of \( E1 \) (we say \( E2 \) triggers \( E1 \) for simplicity). Suppose, however, \( E2 \) can further be triggered by \( E3 \) (which may or may not be a subevent type of \( E2 \)), then we have two possibilities to consider: one is that \( E3 \) can also trigger \( E1 \), and the other is that \( E3 \) can not trigger \( E1 \). We regard either case as problematic. In the first case, in particular, \( E1 \) will be triggered twice: once by \( E3 \) and once by \( E2 \) which is itself also triggered by \( E3 \). This creates a redundant triggering of \( E1 \) and therefore incurs an overhead for the system to handle. In the second case (i.e., \( E3 \) can not trigger \( E1 \) according to \( E1 \)'s composition), inconsistency will result since \( E2 \) (triggered by \( E3 \)) will cause \( E1 \) to occur anyway due to the subtyping relationship. Thus the occurrence of \( E3 \) still causes \( E1 \) to occur, which is a contradiction. For these reasons, we do not support the super/subtype relationship between two composite event types in our model.

28
There will be no unwanted event occurrences produced as a result of event propagation, as an event occurrence is produced only if the occurrence is “useful” to a rule or to a composite event. (By “useful” we mean the event occurrence will actually trigger a rule or it can potentially induce a composite event). Our event generation algorithm specifies that an event occurrence of the DBE event type $E$ on object $o$ can only be generated if $E$ is in active state with respect to object $o$. We define informally the meaning of active state of an event type as follows:

Let $E$ be a DBE event type, $R$ be the set of rules associated with $E$ and $C$ be the set of composite events that each have $E$ as a constituent event. $E$ is in active state with respect to object $o$ if and only if the predicate $IN\_ACTIVE\_STATE(E, o)$ is true. $IN\_ACTIVE\_STATE(E, o)$ is true if and only if:

1. $E$ is not disabled explicitly, and

2. There exists a rule $r \in R$ such that $IN\_ACTIVE\_STATE(r, o)$ is true or there exists an event $E1 \in C$ such that $IN\_ACTIVE\_STATE(E1, o)$ is true.

Similarly, we define $IN\_ACTIVE\_STATE(r, o)$ to be true if and only if the rule $r$ is in active state with respect to object $o$. That is, $IN\_ACTIVE\_STATE(r, o)$ is true if and only if:

1. Rule $r$ is not disabled, and

2. There exists an event associated with $r$, and

3. Either object $o$ is associated with the rule $r$, or object $o$ is an instance of a class/role attached to the rule $r$.

### 3.5 Rule and Event Language

To allow applications to interface with the system, a language for definition/manipulation of database objects and performing other database operations is needed. In this section, we concentrate on the aspect of the language related to rules and events. Thus, we describe in the following subsections the syntax and semantics of rule and event specification and some auxiliary operations.
3.5.1 Rule Conditions & Actions

Rules in ADOME-II take the general form:

\[ \text{if } \langle \text{condition-spec} \rangle \text{ then } \langle \text{then-action-spec} \rangle \text{ else } \langle \text{else-action-spec} \rangle \]

The \( \langle \text{condition-spec} \rangle \) specifies the condition to be checked once the rule is triggered and before the action is executed. The \( \langle \text{then-action-spec} \rangle \) and \( \langle \text{else-action-spec} \rangle \) specify the actions to be executed when the condition is satisfied or not satisfied, respectively.

The condition of a rule can be expressed in several ways [44]: database predicates, restricted predicates, database queries, application procedures, or some combination of them. The choice depends on factors such as efficiency of evaluation of the condition, and expressiveness and declarativeness of the condition. Here the focus is on the expressiveness and declarativeness of the condition. Thus, the condition allows predicates as well as database queries in the form of SQL-like queries (extended with the object-oriented features). Predicates simplify specification of simple conditions while database queries allow the full functionality of the system to be accessed.

To elaborate, we describe in the following the components of the rule condition. There are four components that can be specified in a rule condition: queries, functions and set quantifiers (all, any), variables, and logical operators (and, or, not).

- **Queries.** These are SQL-like queries that allow objects to be queried and methods to be executed. Queries returning an empty set are considered as false, otherwise they are considered true.

- **Functions and set quantifiers.** Functions can perform computation and may return values: numbers, OID, boolean value or set. They can be any method (which can contain database queries) and may have side-effects. Further, relational functions, such as equality, “greater than” or “smaller than” etc., on objects of an ordinary class or a role class may be quantified by set quantifiers. Thus, the condition (Person.Age all < 18) evaluates to true if all persons are under 18. Such set quantified relational functions can be translated syntactically into equivalent database queries.
• **Variables.** Variables are used to bind event occurrence to the rule condition and to bind results of condition evaluations to the action. For example, a variable \( v \) can be bound to the result of a function \( F(x) \) by writing "\( v \text{ as } F(x) \)". Two pre-defined variables are available to use in a rule: **this** refers to the object that is matched by the rule, **event** refers to the event occurrence that triggers the rule.

Since parameters of event occurrences which trigger a rule are stored as properties of event occurrences, they can be accessed by referring to the corresponding attribute of event instance through composition links. Example 3-2 illustrates the use of variables to access the parameter of an event occurrence.

**Example 3-2** Consider the composite event Credits-Remain-Change as defined below, it can be applied to PhD, Master, etc. depending on the binding of associated rules and objects. The pre-defined variable **event** will be bound to either one of the constituent event occurrences and can be used in a rule body as following:

```
Create Event Credits-Remain-Change as
   ([Update Credits-Taken] or [Update Min-Credits-Required]);

-- Binding an event to a rule
On Credits-Remain-Change Check
   Monitor-Requirement =
      If (event.object.Credits-Taken < event.object.Min-Credits-Required) then
         this.Department.Alert(this);

-- Associating the rule to roles and players
   Attach Monitor-Requirement to PhD;
   Attach Monitor-Requirement to m in Master
      where mode-of-study = 'Full-Time';
```

References to transition values are via keywords **Old** and **New.** For example, **Old.Salary** denotes the previous value of attribute Salary after an attribute update operation. Transition value references via the keywords are translated into queries on event occurrences. Such queries are possible since transition values are stored as properties of event occurrences, which are themselves objects.
3.5.2 Event Specification

In our language, event types are specified according to the following BNF grammar:

\[
\text{<event-spec>} ::= [\text{before} \mid \text{after}] \text{<event-name>} [\text{<class-name>}]
\]

\[
[\text{<object-property-spec>}] \mid \text{only}\]
\]

Event modifier is specified by before and after and it is defaulted to after if not specified. <class-name> and <object-property-spec> are optional for some of the event types. Notice that the <event-name> should be the name of one of the event types in the internal layer or operational layer, or the name of an abstract or composite events. By default, all user-defined primitive events are subtypes of the system-defined event type Primitive. However, this can be overridden by using the SubType keyword. Thus, abstract events can define their own inheritance path. Definition of a composite event bears less resemblance to class definition because a composite event does not have attributes or methods of its own. Composite events are composed of constituent events (which can be primitive or composite) by logical connectives. The semantics of the event operators are given in chapter 4.

Event type can be disabled and later re-enabled by the following statement: Disable/Enable Event <event-spec>. A disabled event type retains its properties (attributes, composition in case of composite event and associated rules) but no event occurrence of it is generated until it is re-enabled.

Composite event types can be modified by using the Update statement, as exemplified below. Since event types are classes, they retain their identities after modifications.

update Event Credits-Remain-Change
set Composition = ((Update Credits-Taken] or [Update Min-Credits-Required] or
[Update Credits-Exempted]);

32
3.5.3 Event-Rule Association

In ADOME-II, events are bound to rules by the following two forms of constructs.

\[
\text{On} \ [<\text{event-spec}] \ [=] \ [<\text{composition-spec}>] \ \text{Check} \\
\quad \ <\text{rule-name}> \ | \ {<\text{rule-name}> = <\text{rule-spec}>}
\]
\[
\text{On} \ <\text{variable}> \ \text{Check} \ <\text{variable}> \ [\text{for} <\text{variable}> \ \text{in} <\text{set}> \ \{, <\text{variable}> \ \text{in} \ <\text{set}>\}+] \ \text{where} \ <\text{condition}>
\]

The above constructs support single and set operations, respectively; in the second form, a set of events can be bound to a set of rules based on the specified condition. Further, event types and rules can be created "on the fly". Creation of anonymous event types is also supported, which is useful when the event types are not intended to be shared. To further streamline the operation, if only <event-spec> is supplied and if the event type does not exist, a new abstract event type will be created. If <composition-spec> is supplied, a new composite event type will be created. In both cases, if event name is omitted, a system-generated name will be given to the newly created event type.

**Example 3-3** Suppose the library of a university wants to require all students leaving the university (due to graduation or withdrawal from study) to clear their outstanding fines and loans. This can be done by the following statement:

\[
\text{On} \ e \ \text{Check} \ r \ \text{for} \ e \ \text{in} \ \text{Class-Object(University).Student-Leave-Events,} \\
\quad r \ \text{in} \ \text{Class-Object(University).Library.Check-Outstanding-Rules};
\]

Disassociating an event type from a rule is done by the following:

\[
\text{Unbind} \ <\text{event-spec}> \ | \ <\text{variable}> \ \text{From} \ <\text{rule-name}> \ | \ <\text{variable}> \\
\quad \{\text{for} <\text{variable}> \ \text{in} \ <\text{set}> \ \{, <\text{variable}> \ \text{in} \ <\text{set}>\}\
\]

Thus, to make the library not to check for outstanding items when students take temporary leaves from their studies, we can use the following statement:

\[
\text{Unbind} \ \text{Temporary-Leave-Event} \ \text{From} \ r \\
\quad \text{for} \ r \ \text{in} \ \text{Class-Object(University).Library.Check-Outstanding-Rules};
\]
3.5.4 Rule-Object Association

Associating rules with entities explicitly is the only way to share rules in ADOME-II. Rules can be dynamically attached to/detached from objects, roles and classes in two modes:

1. **Operational mode.** In this mode, rules and entities are associated and disassociated manually. An entity that was attached to a rule in this mode will remain attached until the concerned entity is deleted or explicitly detached from the rule. The construct for associating rules and entities in the operational mode is as follows:

   Attach | Detach <rule-name> | <variable> to <variable> | <entity-set>
          | [for <variable> in <set> [(<variable> in <entity-set>)+] where <condition>] |

2. **Declarative mode.** Associating rules with a class/role can be done in this mode. A qualification invariant is required when performing association. The invariant restricts the applicability of the rules to a subset of the class/role. The subset is automatically maintained to contain only the objects satisfying the qualification invariant. Rule-object association in this mode can be done in the following constructs:

   Attach with invariant <rule-name> | <rule-var> to <class-role>
   [for <rule-var> in <set>] where <invariant>

   Detach with invariant <rule-name> | <rule-var> from <class-role>
   [for <rule-var> in <set>] where <invariant>

Here <class-role> is either a class name or role name; <invariant> is the invariant qualification on the <class-role>, which is a logical composition of predicates on attribute values. We restrict the qualification to predicates on attribute values so as to allow automatic generation of ECA rules for maintaining the association of rules with roles/classes according to the invariant qualification.

For clarity of intension and simplicity of implementation, we also restrict each role/class to have at most one invariant qualification with any rule.
The invariant qualification is enforced upon the execution of binding and it is removed when it is explicitly detached from `<class-role>`. An attempt to remove an invariant qualification without a corresponding execution of `attach with invariant` statement is invalid. On the other hand, an execution of `detach` on an object (of `<class-role>`) that was attached to a rule with an invariant qualification will be treated as an exception to `<class-role>`; and the invariant qualification will then not be applied to the object.

To cater for exceptions for role/class-level rules, the following constructs can be used to suspend (and later resume) a rule for individual objects/role instances:

```
Suspend | Resume <rule-name> | <variable> from <variable> in <set>
          [for <variable> in <set> [{<variable> in <set>}]| where <condition>]
```

As an example, enforcing academic rules on all students can be done by the following statement:

```
Attach ?r to Student for ?r in Class-Object(University).Academic-Rules;
```

On the other hand, exempting a student from some graduate requirements can be done by the following statement:

```
Suspend Graduate-Req-Rule from ?s for ?s in Student where ?s.name = 'Smith';
```

### 3.6 Execution Semantics

Rule execution and event processing semantics prescribe how the database behaves. Therefore, any active database or any system providing reactive capabilities must have a well-defined execution model. In the following the execution semantics of rule and composite event detection are described.

Rule and event processing and normal database operations are interleaved, as depicted below:

![Figure 3-4 Interleaving of Database Operations and Rule & Event Processing Cycle](image)

35
Initially, that is at the start of a transaction, no events have occurred and there are no rules pending to be executed. As database operations are executed, primitive (including abstract) events are raised, but they are not processed—rules are not triggered and composite events are not detected, until at some specific points—either a query statement is finished or at the request of the application, the rule and event processing cycle begins. The processing granularity is made configurable so that applications can tune the granularity to suit their requirements. For example, to obtain smaller response time of normal database operations, the processing granularity can be made coarser; to ignore some integrity constraints momentarily during some intermediate database operations, the processing granularity can be made larger than the block of the intermediate database operations. At the end of the rule and event processing cycle, all events and rules would be processed to completion. There would be no un-processed events or pending rules left, and normal database operations are resumed.

Similar to processing granularity, coupling modes of a rule can be used to control the time of execution of a rule. The Event-Condition (E-C) coupling mode controls the time of condition evaluation while the Condition-Action (C-A) coupling mode controls the time of action execution. Unlike processing granularity, coupling modes are on a per rule basis. Here, since rules are executed in the same transaction, only immediate and deferred coupling modes are supported. Deferred E-C coupling mode is useful for integrity maintenance rules where momentary violations of integrity within a transaction are tolerated, and such rules only need to be checked at the end of a transaction. In contrast, deferred C-A coupling mode is not currently supported partly because its utility is not as large as the E-C counterpart, and partly because of implementation concerns. However, it should be easily observed that deferred coupling mode can be simulated by two rules: one to just evaluate the condition and signal if the condition is satisfied and the other one (with deferred E-C coupling mode) to execute the action at the end of a transaction if a signal (embedded in the condition) is found.

During a rule's execution, events may be generated as a result of the evaluation of the rule's condition or actions but no other rules would be executed before comple-
tion of the executing rule (the detailed execution algorithm of rule and event processing can be found in chapter 4, section 4.2.1). This kind of execution is referred to as *iterative* or sometimes *non-interruptible*. The benefit of iterative rule execution is that the effects of database operations performed in a rule would not be affected or even un-done by rules resulting from the database operations. Consequently, rule behavior is more predictable.

At any point in a rule and event processing cycle, there may be more than one pending rule. This is because rules may be triggered simultaneously by the same event, rule execution is iterative, and the processing granularity of rules may be large enough so that multiple rules are delayed for processing. In such cases, it is desirable that rule execution is deterministic so that rule behavior can be more predictable. The policy here to enforce deterministic rule execution is a combination of absolute rule priority and time of rule activation. Activated rules with higher priorities are executed first. The execution order of rules with the same priority are resolved according to recency of their activations, with the earliest one being executed first. The choice of this policy is influenced by the native rule-based system, in which the policy is readily supported, and can be regarded as first attempt to rule scheduling.

As a concluding remark, the execution semantics are generally complicated in the context of active databases. To understand the behavior of rules and events, the execution semantics must be well-defined. These have been described in this section, and the reasons for choices also explained. Table 3-2 summarizes the execution semantics defined in this section.

**Table 3-2 Summary of execution semantics**

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing granularity</td>
<td>Configurable from statement-level to transaction-level.</td>
</tr>
<tr>
<td>Coupling Modes</td>
<td>E-C coupling modes: immediate, deferred.</td>
</tr>
<tr>
<td></td>
<td>C-A coupling mode: immediate only.</td>
</tr>
<tr>
<td>Cascading rule execution</td>
<td>Iterative rule execution.</td>
</tr>
<tr>
<td>Rule Scheduling</td>
<td>Absolute rule priority and activation recency.</td>
</tr>
</tbody>
</table>

37
Chapter 4

Extensible Approach To Reactive Processing

A number of active object-oriented database systems have been constructed in recent years. A few of them have very rich composite event specification languages including: Sentinel, ODE, and SAMOS. While their event expressions have many common (or equivalent) features with each other, their event detection mechanisms are totally different. In the context of ADOME-II, the need for rich composite event types is recognized for supporting traditional condition-action rules and for rules where explicit controls are desirable. As there is little consensus on a minimal and sufficient set of event operators and it is possible that new event operators may be needed when emerging requirements are identified, we therefore present in this chapter an extensible and uniform way of processing rules and events. We first describe the common framework for rule processing and event detection, followed by introducing a set of pre-defined event operators. Next we discuss in detail the approach to detecting composite events, and then show how some extensions to the pre-defined event operators can be easily accomplished. In the last section, the approach presented here is compared with other relevant approaches.

4.1 Reactive Functionalities

As can be seen from the current literature, the set of reactive functionalities varies greatly from one database system to another. To clearly understand the framework of reactive processing in ADOME-II, the set of essential reactive functionalities must first be defined. The system currently provides the following core reactive functionalities:
• **Event generation.** Most database operations generate events. Prior to event generation, the necessities of generation have to be checked. A situation where occurrences of an event type should not be generated is when the event type is disabled. Once necessities of generation are affirmed, these event occurrences have to be recorded and categorized into appropriate event types. Propagation of event occurrences may be required depending on the types of event occurrences and states of the event types. Moreover, transition values and other event parameters have to be collected.

• **Composite event detection.** Logical events, sequence events, temporal events and cumulative versions of these events have to be detected according to certain parameter contexts. The necessities of new composite event occurrence generation have to be checked and appropriate event parameters have to be collected. A vital requirement of composite event detection is efficiency; it is highly desirable that this can be done incrementally.

• **Rule Activation.** When an event occurs, its associated rules may be triggered. Because of the iterative nature of the rule processing cycle and the possibility of simultaneous triggering of rules, a rule agenda is maintained to store the rules pending execution. Every triggered rule is put into the agenda along with necessary information about the triggering of the rule and the rules in the agenda are executed sequentially with their order of execution determined by a scheduling and conflict resolution strategy. Every triggered rule must be activated before it can be scheduled and executed. The time of a rule’s activation depends on the state of a rule and the E-C (event-condition) coupling mode of the rule; e.g. a disabled rule or an enabled rule with deferred E-C coupling mode will not be activated. A rule with deferred E-C coupling mode will be activated before the transaction commits. Modifications to the state of a rule will immediately affect the entry of the rule in the agenda.
- **Evaluation of rule condition.** Rule execution begins only after the rule is activated. The first step of rule execution is condition evaluation which requires the ability to execute arbitrary database queries, and store the results of evaluation to be passed to actions of the rule. Since arbitrary database queries are allowed in the condition, events as results of such queries may be generated.

- **Execution of rule actions.** The actions of a rule are executed immediately after the condition is evaluated to be true. Corresponding actions are executed according to the evaluation result of the condition. The action part can execute arbitrary database queries, with the ability to refer to the result of condition evaluation as well as other information that can be referred to in the condition. It is important that operations carried out in the action part can also generate events so that cascading execution of rules is possible.

- **Rule scheduling and conflict resolution.** The presence of simultaneous activation of rules results in potential non-determinism in rule execution. Rule priority and conflict resolution strategy are two policies used together to enforce deterministic rule execution. Rule activations are scheduled according to their rule priorities, with the one of highest priority executed first. Activation of rules of the same priority are resolved by the first-in-first-out strategy. In this strategy, the one activated first will be executed earlier than all others with the same priority. An important property of rule scheduling is that any changes in rule priority will be reflected immediately in the rule execution schedule. Therefore, the ability to reorder the rule schedule is required.

### 4.2 Framework for Supporting Reactive Processing

#### 4.2.1 Production Rule for Supporting Reactive Processing

Production rules and ECA rules are similar; superficially, a production rule can be considered as an ECA rule with the event part being implicit. Whether an ECA rule can be translated into equivalent production rules is, however, not known. Interestingly, our approach demonstrates that, to a large extent, reactive capabilities typically
found in active databases can be achieved by production rules. Composite event
detection and rule processing are two major functions to be supported in active data-
bases. In [11], semantics of an event specification language Snoop are expressed for-
mally in a combination of logic and operations, indicating that it may be feasible to
express such semantics in production rules. The approach presented here enables not
only detection of composite events in production rules, but also ECA rule processing
in production rules.

An alternative to using production rules to detect composite events is using ECA
rules since they are more general than production rules. In this alternative, event
detection rules are ECA rules and composite event types are modelled as abstract
event types. Each event detection rule is triggered by the occurrence of the constituent
events of the composite event that the rule detects. Unlike their production rule coun-
terparts, where sentries for constituent events are embedded in the condition, the ECA
event detection rules are explicitly triggered by the constituent events specified as
event components of the rules. The advantage of this alternative is that a single rule
model (i.e. the ECA rule model) can be used. Whatever functionalities that can be
used in ECA rules (which are usually application-level functionalities) can be used in
event detection rules. The comprehensibility of event detection rules is therefore
much improved, leading to lower design complexity. The major disadvantage of using
ECA event detection rules is the extra level of overhead that results from event detec-
tion rule processing---detection of composite events are as slow as processing ECA
rules. Another reason for the unfavorable performance caused by the use of ECA
event detection rules is their lack of optimization of condition evaluation. Developing
an ECA rule system (i.e., active database) is already a demanding task. Therefore
most ECA rule systems do not support incremental condition evaluation. In contrast,
techniques for efficient evaluation of production rules exist and are widely used as
exemplified by the use of discriminant networks where conditions can be evaluated
incrementally and common expressions among conditions are evaluated only once for
all of them.

Based on the analysis above and the preference for composite event detection
efficiency over design complexity, production rules are therefore the candidate for
supporting composite event detection. The whole approach rests on the use of production rules as a unifying mechanism for supporting composite event detection, rule triggering, activation and execution. Composite event detection is done by a set of event detection rules. When an event occurs, its associated rules are triggered through the execution of its specialized internal rules, namely “rule-triggering” production rules. On the other hand, a set of “wrapper” production rules are used to monitor when to execute application rules. Figure 4-1 shows the steps involved in executing the three kinds of production rules. More details can be found in the following sections. Note that, event detection rules have higher priorities than wrapper rules. This ensures that composite events are always detected before subsequent execution of application rules can affect the detection.

TR: set of pending rule-triggering production rules;
WR: set of pending wrapper production rules, initially WR = Ø;
EDR: set of pending event detection rules;

repeat

while EDR ⊕ TR ≠ Ø do
    R ← First rule in EDR ⊕ TR;
    Execute R and remove R from EDR ⊕ TR;
end-while

if WR ≠ Ø then
    R ← First rule in the WR;
    if (R, EC-coupling-mode = IMMEDIATE) then
        Execute R and remove R from WR;
    else
        if (Event Pre-Commit occurred) then
            Execute R and remove R from WR;
        end-if
    end-if
    Re-order WR;
end-if

until EDR = Ø ⊕ TR = Ø ⊕ WR = Ø;

Figure 4-1 Event & Rule Processing Cycle

4.2.2 Supporting Rule Triggering, Activation & Execution

The stages of rule triggering for execution and the roles played by different rules at different stages are summarized as follows:
1. An application rule is triggered by an event occurrence of type E. The rule-triggering production rule of an event type E is responsible to trigger the application rule, and it has the form shown below:

   Rule TR:
   
   Condition: An event of type E occurred
   Action: Triggers rule(s) associated with E

   The rule TR above eventually gets executed after an event of type E occurs. As the result of the execution, an entry is created in the rule agenda for the associated application rule(s).

2. The wrapper production rule for each triggered rule in the agenda monitors the state and E-C coupling mode of the triggered rule. As soon as the condition of the wrapper rules is satisfied, the triggered rule is logically considered activated and scheduled for execution. The actual scheduled rule is, however, the wrapper rule of which an action is to execute the triggered rule. The forms of wrapper rules for different E-C coupling mode are shown below, with the first one being for immediate E-C coupling mode and the second one for deferred E-C coupling mode.

   Rule Wrapper-Immediate:
   
   Condition: rule R is triggered ^ R is enabled
   Action: if (R.Condition) then R.Then-Action else R.Else-Action, Remove agenda entry

   Rule Wrapper-Deferred:
   
   Condition: rule R is triggered ^ R is enabled ^ event Pre-Commit occurred
   Action: if (R.Condition) then R.Then-Action else R.Else-Action, Remove agenda entry

3. Wrapper rules are scheduled according to their priorities, each of which is determined by the priority of the application rule a wrapper rule wraps. Those with the same priority have their order of execution resolved by their time of activation, i.e. the relative time of condition satisfaction. Deterministic rule execution is thus guaranteed.

4. When a wrapper rule is executed, its actions are executed. The actions of a wrapper rule are to execute an application rule and remove the application rule's entry from the agenda. The application rule is represented in the procedural construct
if...then...else... of the production rule system. The reasons to wrap an application rule in a single procedural construct are: firstly, the application rule's condition will not be evaluated more than once which ensures that the time of condition evaluation is precise, and that side-effects of condition evaluation can be more predictable; secondly, variables that record the results of condition evaluation can be passed to actions of the same rule.

4.2.3 Supporting Composite Event Detection

Two fundamental steps are required to detect composite events: monitoring over the conditions of occurrence of composite events and signaling the occurrence when the conditions are met. Recall that an event occurrence is stored as an instance of an event class. Therefore, signaling an occurrence of a composite event is achieved by creating an instance of the appropriate event class. Monitoring the conditions of occurrence of composite events is done by event detection production rules, or simply event detection rules. For each composite event type, there is at least one such rule in which the condition of occurrence is embedded. A condition of an event detection rule is a set of first-order predicates and functions on event types, event instances and their properties. Actions are functions and method invocations. It is possible for several event detection rules to cooperate to detect occurrences of a single event type. The pattern of cooperation is totally under the control of event detection rules—usually in the form of state transitions of an event occurrence.

4.3 Pre-Defined Composite Event Expressions

The set of event expressions, which has been defined in ADOME-II, overlaps with Snoop [10] to a large extent. The event expressions in Snoop are incorporated here because of their richness and the observation that many event expressions in other event specification languages are expressible in Snoop. It is therefore a suitable candidate for us to demonstrate the expressive power of our approach. Each of the event expressions below has a Snoop counterpart, with some being slightly different from Snoop's. In particular, some of Snoop's event expressions allow temporal event
expressions to be used as arguments, but for simplicity we have constrained these to be events only:

- \textit{And}(E_1,\ldots,E_n) denotes the occurrence of a conjunction of \textit{E}_1 through \textit{E}_n. This composite event occurs when events \textit{E}_1 through \textit{E}_n are simultaneously present irrespective of their order of occurrence.

- \textit{Or}(E_1,\ldots,E_n) denotes the occurrence of a disjunction of \textit{E}_1 through \textit{E}_n (which are all distinct events). It occurs when any of the \textit{E}_i occur, where \textit{i} = 1,\ldots,n.

- \textit{Seq}(E_1,\ldots,E_n) denotes the occurrence of a sequence of \textit{E}_1 through \textit{E}_n in chronological order, with \textit{E}_1 being the first occurrence in the order.

- \textit{Not}(E_1, E_2, E_3) denotes the non-occurrence of \textit{E}_2 within a closed time interval, of which the starting and ending time are the occurrence time of \textit{E}_1 and \textit{E}_3, respectively.

- \textit{Peridoic}(E_1, F, E_3) denotes a periodic event which occurs at the frequency of \textit{F} within the closed time interval defined by the time of occurrence of \textit{E}_1 and \textit{E}_3. \textit{F} is a time period string of the form "s:m:h d" where \textit{s}, \textit{m}, \textit{h} and \textit{d} denotes the number of seconds, number of minutes, number of hours and number of days, respectively. Unlike in Snoop, the periodic event here does not collect parameters during each occurrence. (Therefore the cumulative version of periodic event is not targeted here.)

- \textit{Aperiodic}(E_1, E_2, E_3) denotes the aperiodic event expression which occurs each time \textit{E}_2 occurs within the closed time interval defined by \textit{E}_1 and \textit{E}_3 occurrences.

- \textit{Aperiodic*}(E_1, E_2, E_3) is the cumulative version of aperiodic event expression. It occurs at the occurrence of \textit{E}_3 and includes all occurrences of \textit{E}_2 within the closed time interval defined by the the time of occurrence of \textit{E}_1 and \textit{E}_3.

The following three new/extra event expressions augment those above with the ability to refer to time in event expressions:
• *Every*(EI,TP,E3) occurs repeatedly at times which match the time pattern TP within the closed time interval defined by EI and E3. TP is a string of the form "s:m:h day-of-week day-of-month/month/year" of which any field in the string can be a wild-card. Fields filled with wild-cards are ignored when matching the time pattern with an absolute time.

• *At*(TP) occurs at most once at time specified by TP. Here TP has the same form as given above except wild-cards are not allowed.

• *After*(EI, T) denotes a relative temporal event expression. It occurs at time T after the occurrence of EI.

Two more event expressions can also be easily accommodated:

• *Disjoint_seq*(EI,...,En) denotes a disjoint sequence of events EI through En. Two events are disjointed if their time intervals do not overlap. This event expression approximately corresponds to the expression relative(EI,...,En) in ODE [28].

• *Within*(EI, E2, T) occurs if the occurrences of EI and E2 are within the period of T regardless of the occurrence order.

These event expressions are pre-defined in ADOME-II in order to facilitate the formulation of various event expressions. In contrast with many other works on composite event support, where the focus was on the decalartiveness and expressiveness of the event specification language, the approach presented here is mainly targeted at extensibility, focusing on accommodating new event expressions rather than supporting a fixed set of event expressions.

### 4.4 Incremental Composite Event Detection with Production Rules

The requirement to enforce the invariants of detecting DBE composite events (as described in section 3.4.2 of chapter 3), in addition to handling parameter contexts correctly, adds to the complexity and overhead of composite event detection. We therefore propose here a modular, declarative and incremental approach to event
detection so as to alleviate such difficulties. In the following subsections, the methodology of detecting all pre-defined event expressions is described.

4.4.1 Event Detection Rules

Each composite event is detected by a set of event detection rules. Each event detection rule is locally visible to an event type and is composed of a condition part and action part. The condition is expressed in first-order and some second-order predicates (with the extension that functions and methods are allowed in logic formula and object properties can be referred to in the dot notation). Note that these second-order predicates can be transformed into first-order predicates because of the way they are defined. The presence of second-order predicates is only for brevity of formulation.

To aid understanding of event detection rules, predicates are classified into the following categories:

- **Event Type Qualifying Predicates**: these are used to place an event detection rule in the context of an event type and/or to detect the only occurrence of an event type.

- **Triggering Object Qualifying Predicates**: these predicates enforce restrictions on triggering objects of an event occurrence. Examples of such restrictions include the time of occurrence and also the correspondence among the triggering objects of constituent events.

- **Parameter Context Qualifying Predicates**: these predicates filter the event occurrences in a particular parameter context. Without specifying a parameter context for predicates in the event detection rules of an event type, occurrences of the event type are detected in an un-restricted context.

- **Auxiliary Predicates**: this category encompasses all other predicates that fall outside the above three categories. Examples of predicates of this category are: predicates for reasoning about object and event occurrence structures, temporal predicates, predicates on event occurrence states, etc.
All free variables in a condition are meant to be universally quantified in the range of the condition. The Closed World Assumption is used in evaluating negation predicates. Also, for the sake of brevity, conversion between different time representations is omitted.

A number of predicates, functions and methods are used in the formulation of several composite event detection rules. Therefore, we define them prior to formulating event detection rules. Note that for rules where arguments are constants, they can be computed at compile time.

**Definition 4-1** Instance($E, x$) = $E(x) \equiv x$ is an instance of class $E$. E.g. Employee($x$) denotes that $x$ is an instance of the class Employee.

**Definition 4-2** Dynamically-bound($x$) denotes that $x$ is a Dynamically-bound event type.

**Definition 4-3** The function Class($x$) returns the name of the class of $x$.

**Definition 4-4** The method Delete-Object() of an object deletes the object itself.

**Definition 4-5** The method Raise($e1, ..., en$) of composite event types raises an event with constituent events $e1, ..., en$ and returns the newly raised event occurrence.

**Definition 4-6** Enabled($E$) $\equiv$ $\neg$ E.is-disabled

**Definition 4-7** Matchable($x$) $\equiv$ $x$.state = MATCHABLE. This predicate means an event occurrence $x$ is ready to induce other composite event occurrences. (There are a number of situations where Matchable($x$) is false, e.g. one is when $x$ is subject to successive updates where intermediate states of $x$ are not to be exposed)

**Definition 4-8** Same-Object($x, y$) $\equiv$ $\neg$ Dynamically-bound(Class($x$)) $\lor$ Dynamically-Bound(Class($y$)) $\lor$ ($x$.obj = $y$.obj). The predicate means that the objects that generate events $x$ and $y$ are the same unless one of them is not of dynamically-bound event types. This predicate in effect enforces the object-correspondence invariant. Another form of the predicate, Same-Object($x1, ..., xn$), is equivalent to the conjunction of Same-Object($xi, xj$) where $(i, j) \in \{(a, b) : a = 1..n, b = 1..n and a \neq b\}$.

**Definition 4-9** Eligible($E, x$) $\equiv$ $E$.active-time $\leq$ $x$.te. This predicate asserts that the occurrence of $x$ is after the activation of event type $E$.

**Definition 4-10** Follows($x, y$) $\equiv$ $x$.te $\geq$ $y$.te. The value of the attribute $te$ is the ending time of an event’s time interval. The predicate asserts that the event $y$ must occur not later than the event $x$. The generalized form Follows($x1, ..., xn$) is equivalent to Follows($xn, xn-1$) $\wedge$...$\wedge$ Follows($x2, x1$).

48
Definition 4-11 Absolute-Follows(x, y) ≡ x.te > y.te. This predicate is similar to Follows(x, y) except x and y cannot overlap. The generalized form Absolute-Follows(x1,...,xn), is equivalent to Absolute-Follows(xn, xn-1) ^...^ Absolute-Follows(x2, x1).

Definition 4-12 Most-Recent(E, x) ≡ ∀y [(E(y) ^ Same-Object(x, y)) → Follows(x, y)]. This is one of the parameter context qualifying predicates. Altogether there are four such predicates (see section 4.5.1).

Definition 4-13 Occurred-Unrestricted-Context(E, E1, x) ≡ E1(x) ^ Matchable(x) ^ Eligible(E, x). This predicate detects events in unrestricted context.

Definition 4-14 Occurred(E, E1, x) ≡ Occurred-Unrestricted-Context(E, E1, x) ^ Most-Recent(E1, x). Similar to the one above, this predicate detects events in the most recent context.

4.4.2 Logical and Sequence Events Detection Rules

Logical events are events composed of logical event expressions. Conjunction event expressions and other logical event expressions can be transformed into corresponding event detection rules in a straightforward manner. For instance, the conjunction event expression below specifies that a conjunction event E will occur only if E is enabled, and both E1 and E2 occurred, and they both satisfy invariants on triggering objects and parameter context requirements.

\[ E = \text{And}(E1, E2) \]

Condition: \( Enabled(E) \land Occurred(E, E1, x1) \land Occurred(E, E2, x2) \land Same-Object(x1, x2) \)

Action : E.Raise(x1, x2)

The occurrence of event E is signaled by calling the method Raise() of event type E with constituent event occurrences. To detect the general case, that is, \( E = \text{And}(E1, ..., En) \), the event detection rule can be extended accordingly.

Disjunction event expressions can be handled similarly. Note, however, that simultaneous event occurrences of E1 and E2 will trigger two occurrences of E.

\[ E = \text{Or}(E1, E2) \]

Condition: \( Enabled(E) \land (Occured(E, E1, x) \lor Occurred(E, E2, x)) \)

Action : E.Raise(x)

The sequence event detection rule (as well as many others) is constructed from that of the conjunction event expression with additional auxiliary predicates. Note
that due to the similarity with the conjunction event detection rule, their common sub-expressions can be extracted and evaluated once if necessary.

\[ E = \text{Seq}(E_1, E_2) \]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E_1, x_1) \land \text{Occurred}(E, E_2, x_2) \land \text{Absolute-Follows}(x_1, x_2) \land \text{Same-Object}(x_1, x_2) \)

**Action:** \( E.\text{Raise}(x_1, x_2) \)

A small modification to a sequence event detection rule yields the one for a disjoint sequence event expression, as described below:

**Definition 4-15** \( \text{Disjoint-Absolute-Follows}(x, y) \equiv x.ts > y.te \). The value of the attribute \( ts \) is the starting time of an event’s time interval. The predicate asserts that the event \( x \) must occur later than the event \( y \) without overlapping.

The event detection rule for the disjoint sequence event is as follows:

\[ E = \text{Disjoint-Seg}(E_1, E_2) \]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E_1, x_1) \land \text{Occurred}(E, E_2, x_2) \land \text{Disjoint-Absolute-Follows}(x_1, x_2) \land \text{Same-Object}(x_1, x_2) \)

**Action:** \( E.\text{Raise}(x_1, x_2) \)

**Aperiodic, Within** and **Not** events can be detected with one rule for each event, as described below:

\[ E = \text{Aperiodic}(E_1, E_2, E_3) \]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E_1, x_1) \land \text{Occurred}(E, E_2, x_2) \land \text{Follows}(x_2, x_1) \land \text{Same-Object}(x_1, x_2) \land \sim (\text{Occurred}(E, E_3, x_3) \land \text{Absolute-Follows}(x_2, x_3) \land \text{Same-Object}(x_1, x_2, x_3)) \)

**Action:** \( E.\text{Raise}(x_1, x_2) \)

\[ E = \text{Within}(E_1, E_2, T = \text{"sec:min:hr d/m/y"}) \]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E_1, x_1) \land \text{Occurred}(E, E_2, x_2) \land |x_1.te - x_2.te| \leq T \)

**Action:** \( E.\text{Raise}(x_1, x_2) \)

\[ E = \text{Not}(E_1, E_2, E_3) \]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E_1, x_1) \land \text{Occurred}(E, E_3, x_3) \land \text{Follows}(x_3, x_1) \land \text{Same-Object}(x_1, x_3) \land \sim (\text{Occurred}(E, E_2, x_2) \land \text{Follows}(x_3, x_2, x_1) \land \text{Same-Object}(x_1, x_2, x_3)) \)

**Action:** \( E.\text{Raise}(x_1, x_3) \)

### 4.4.3 Temporal Event Detection Rules

Temporal events require references to time. Here the generation and detection of time changes are separated. An object is used to model the system clock, which is used for
all time references in event detection rules. This enables time to be referenced in the same model as that of the database.

**Definition 4-16** System-Clock(clock) asserts that clock is the system clock, which is modeled as an object.

The system clock object contains attributes that store break-down components of time like year, month, hour, minutes, etc. Detection of a time change is done by detecting the change in decreasing grain size of time components. For example, to be informed when the moment "12:00 1 Jan 1997" is reached, it is required to detect when the year reaches 1997, then to detect when January is reached and so on until the smallest required time granularity is reached (in this example, it is the minute).

A set of predicates and functions to reference time are defined as follows:

**Definition 4-17** Year(y) ≡ System-Clock(clock) ^ clock.year = y; Month(m) ≡ System-Clock(clock) ^ clock.month = m; Day-Of-Month(d) ≡ System-Clock(clock) ^ clock.day-of-month = d; Hour(h) ≡ System-Clock(clock) ^ clock.hour = h; Minute(m) ≡ System-Clock(clock) ^ clock.minute = m; Seconds(s) ≡ System-Clock(clock) ^ clock.second = s.

**Definition 4-18** Match-Time-Pattern(y, m, d, wd, hr, min, sec) ≡ Year(y) ^ Month(m) ^ Day-Of-Month(d) ^ Day-Of-Week(wd) ^ Hour(hr) ^ Minute(min) ^ System-Clock(clock) ^ clock.second ≥ sec.

If an argument to the predicate is a wild-card (denoted by a negative value), the component predicate that uses the argument is omitted. Thus, if the argument sec is a wild-card, the component predicate clock.second ≥ sec is omitted. Such omission reduces the need for re-evaluation of the whole predicate when the value of the omitted predicate changes.

**Definition 4-19** Match-Scalar-Time(T) ≡ Match-Time-Pattern(Extract-Year(T), Extract-Month(T), Extract-Day-Of-Month(T), Extract-Hour(T), Extract-Minute(T), Extract-Second(T))

**Definition 4-20** Extract-Year(t), Extract-Month(t), Extract-Day-Of-Month(t), Extract-Day-Of-Week(t), Extract-Hour(t), Extract-Minute(t) and Extract-Second(t). These functions accept a scalar time representation and return the extracted component in numbers.

The essential predicates for time references are Match-Time-Pattern and Match-Scalar-Time. They are satisfied when their given time specifications are matched. Note that, to be able to incrementally evaluate them, the “short-circuited evaluation strategy” is required and the formula is assumed to be evaluated from left to right.
This familiar strategy specifies that evaluation of a formula stops as soon as its truth value is known.

Detection of a periodic event \( Periodic(E1, F, E3) \) can be described as a combination of predicates and state transitions. An event occurrence is created and put into an intermediate state when \( E1 \) occurs. The intermediate occurrence also has an alarm attached, which is reset every \( F \) amount of time. The intermediate occurrence transits to the final state when \( E3 \) occurs. Note that, when a periodic event type is disabled, periodic monitoring continues but no occurrences would be generated. We now introduce some predicate, function and methods definitions before describing the actual detection rules.

**Definition 4-21** \( Occurred-Intermediate(E, x, state) \equiv E(x) \land x.state = state. \)

**Definition 4-22** \( Occurred-Any-Intermediate(E, x) \equiv E(x) \land \neg Matchable(x) \)

**Definition 4-23** The method \( Raise-Intermediate(x1, \ldots, xn) \) raises an intermediate event occurrence of the calling event class with constituent events \( x1 \) through \( xn \).

**Definition 4-24** The function \( Triggering-Object(x1, \ldots, xn) \) essentially computes the triggering object for a composite event. It returns the common triggering object of event occurrences \( x1 \) through \( xn \). Only dynamically-bound events are applicable to this function: if none of \( x1 \) through \( xn \) are dynamically-bound events, then \( null \) is returned.

**Definition 4-25** The expression \( x \leftarrow f \) binds the value \( f \) to variable \( x \).

\[
E = Periodic(E1, F = "sec:min:hr d/m/y", E3)
\]

There are five detection rules for a periodic event. Rules 1a and 1b are responsible for initialization when event \( E1 \) occurs. Rules 2a and 2b perform work related to periodic signaling of the event. Lastly, rule 3 terminates the event detection process when \( E3 \) occurs. Formulation of the rules for a periodic event is shown below:

Rule 1a:
**Condition:** \( Enabled(E) \land Occurred(E, E1, x1) \land \neg (Occurred-Any-Intermediate(E, x) \land Same-Object(x1, x)) \land \neg (Occurred(E, E3, x3) \land Absolute-Follows(x3, x1) \land Same-Object(x1, x3)) \)

**Action:** \( x \leftarrow E.Raise-Intermediate(x1), x.obj = Triggering-Object(x, x1) \)

Rule 1b:
Condition: \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x1, x) \land \sim (\text{Match-Scalar-Time}(x1.te + F) \land \sim (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x1) \land \text{Same-Object}(x, x3))) \)

Action: \( x.\text{next-alarm} = x1.te + F \)

Rule 2a:
Condition: \( \text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Match-Scalar-Time}(x.\text{next-alarm}) \land \sim (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3)) \)

Action: \( y \leftarrow E.\text{Raise-Intermediate}(), y.\text{Constituent-Events} = x.\text{Constituent-Events}, y.\text{obj} = x.\text{obj}, y.\text{next-alarm} = x.\text{te} + F, y.\text{ts} = x.\text{ts}, y.\text{te} = x.\text{te}, x.\text{te} = x.\text{next-alarm}, x.\text{state} = \text{MATCHABLE} \)

Rule 2b:
Condition: \( \sim \text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Match-Scalar-Time}(x.\text{next-alarm}) \land \sim (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3)) \)

Action: \( x.\text{next-alarm} = x.\text{te} + F \)

Rule 3:
Condition: \( \text{Occurred-Any-Intermediate}(E, x) \land \text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3) \)

Action: \( x.\text{Delete-Object}() \)

Periodic event detection rules have demonstrated essential techniques for detecting, which can be applied in formulating other temporal event detection rules as well. Therefore, the following formulations of three temporal event detection rules are shown only for completeness:

\[ E = \text{Every}(E1, TP = \text{"sec:min:hr wd d/m/y"}, E3) \]

Condition: \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Match-Time-Pattern}(\text{sec, min, hr, wd, d, m, y}) \land \sim (\text{Occurred}(E, E3, x3) \land \text{Follows}(x3, x1) \land \text{Same-Object}(x1, x3)) \)

Action: \( x \leftarrow E.\text{Raise}(x1), x.\text{te} = TP \)

\[ E = \text{At}(TP = \text{"sec:min:hr * d/m/y"}) \]

Condition: \( \text{Enabled}(E) \land \text{Match-Time-Pattern}(\text{sec, min, hr, -1, d, m, y}) \)

Action: \( x.ts = TP, x.te = TP, x.state = \text{MATCHABLE}, E.isdisabled = true \)

\[ E = \text{After}(E1, T = \text{"sec:min:hr d/m/y"}) \]

Condition: \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Match-Scalar-Time}(x1.te + T) \)

Action: \( x \leftarrow E.\text{Raise}(x1), x.te = x1.te + T \)

### 4.4.4 Cumulative Event Detection Rules

Like periodic events, the \textit{aperiodic} event shown below employs intermediate event occurrences. Here intermediate event occurrences are used to accumulate event occurrences. There are four detection rules used for detecting the event. Rules 1a and
1b perform initialization when the event \( E1 \) occurs. Rule 2 accumulates occurrences of \( E2 \) and rule 3 terminates the detection process and signals the occurrence of the event.

\[ E = \text{Aperiodic}^*(E1, E2, E3) \]

**Rule 1a:**

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \neg (\text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x1, x)) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x1) \land \text{Same-Object}(x1, x3)) \)

**Action:** \( x \leftarrow E.\text{Raise-Intermediate}(x1), x.obj = \text{Triggering-Object}(x, x1) \)

**Rule 1b:**

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x1, x) \land \neg (\text{Occurred}(E, E2, x2) \land \text{Follows}(x2, x1) \land \text{Same-Object}(x2, x1) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x1) \land \text{Same-Object}(x3, x1))) \)

**Action:** \( x.ts = x1.te, x.te = x1.te \)

**Rule 2:**

**Condition:** \( \text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Occurred}(E, E2, x2) \land \text{Follows}(x2, x) \land \text{Same-Object}(x2, x) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3)) \)

**Action:** \( x.\text{Add-Constituent-Event}(x2), x.obj = \text{Triggering-Object}(x, x2) \)

**Rule 3:**

**Condition:** \( \text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3) \)

**Action:** \( x.\text{Add-Constituent-Event}(x3), x.te = x3.te, x.obj = \text{Triggering-Object}(x, x2), x.state = \text{MATCHABLE} \)

### 4.4.5 Feasibility of Evaluating Event Detection Rules

All event detection rules described previously can be evaluated by a production rule system conforming to the following requirements, in addition to standard ones:

1. Arguments to predicates in the condition part of a rule can be objects or object properties, and object operations are allowed in the action part.

2. Functions and methods can be used in conjunction with predicates in the condition part, for example, the formula \( x < (y + z.value()) \), should be allowed.

3. Implication predicates of the form: \( \forall x \ [P(x) \rightarrow Q(x)] \), in the condition part of a rule are supported.

4. Bound variables in the condition part of a rule can be passed to its actions.
The second-order predicates such as "Occurred(E, El, x)" defined in previous sections are convertible to first-order equivalences since the predicate arguments to these second-order predicates are constants, which are known at compile time.

Another concern on the feasibility of evaluating the event detection rules is performance. The actual performance depends on the implementation of the native production rule system, though in principle it is claimed that rule evaluation can be done efficiently. This is because the number of event detection rules of an event type is constant, and the number of condition elements in an event detection rule is also proportional to the number of its arguments. Hence, there is no exponential explosion of rules or condition elements. Also, the event detection rules can be evaluated incrementally. To see this, it is sufficient to recognize that predicates on event occurrences are evaluated only when a new occurrence is created (or deleted, if the predicates contain a negation on the event occurrence), or when attributes of event occurrences are changed. Functions and methods are re-evaluated only when their arguments are changed. Therefore, by observing the event detection rules above, it is clear that event detection rules can be evaluated incrementally.

4.5 Extending Reactive Functionality

One of the strengths of the approach proposed in this chapter is extensibility. However, "extensibility" is a qualitative term which lacks objective measurements. Therefore, we illustrate the extensibility of the approach by example. In particular, methods for handling other parameter contexts and introducing some extra event operators using the approach are outlined.

4.5.1 Handling Other Parameter Contexts

The notion of parameter contexts introduced in Snoop restricts the combination of event occurrences of a composite event in meaningful ways according to the semantics of several classes of applications. Four parameter contexts were introduced in Snoop: recent, chronicle, continuous, and cumulative. The semantics of composite event detection with respect to these contexts are illustrated by the following example.
Example 4-1  Let $E = \text{Seq}(E_1, E_2, E_3)$ and suppose the set of events $\{e_1^i, e_2^j, e_3^j, e_4^j, e_5^j\}$ have occurred where $e_j^i$ denotes the $i$th occurrence of event type $E_j$. The set of event occurrences of $E$ in different parameter contexts are shown in the table below.

<table>
<thead>
<tr>
<th>Parameter Contexts</th>
<th>Set of Event Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recent</td>
<td>$(e_1^1, e_2^1, e_3^1)$</td>
</tr>
<tr>
<td>Chronicle</td>
<td>$(e_1^1, e_2^1, e_3^1)$</td>
</tr>
<tr>
<td>Continuous</td>
<td>$(e_1^1, e_2^1, e_3^1), (e_1^2, e_2^3, e_3^1), (e_1^1, e_2^3, e_3^1)$</td>
</tr>
<tr>
<td>Cumulative</td>
<td>$({e_1^1, e_3^1}, {e_1^2, e_3^1}, {e_1^1})$</td>
</tr>
</tbody>
</table>

Having briefly explained the notion of parameter contexts, it is now ready to illustrate how they can be handled by the approach proposed in this chapter. The most recent context\(^1\) has been incorporated in the formulations presented earlier in the chapter. (cf. Definition (4-12)-(4-14).) The methods for handling the remaining three are outlined below.

**Chronicle Context:**

In the chronicle context each constituent event of the composite event is the oldest un-consumed one among occurrences of its type. To detect events in this context, it is necessary to consume and know whether an event is consumed, and to identify the oldest event among occurrences of the same type. The following two methods can achieve the first two requirements.

**Definition 4-26** The method $\text{Consume}(E, \text{obj})$ of an event type consumes the calling event occurrence. The consumer of the occurrence is an occurrence event type $E$ with the triggering object $\text{obj}$. Subsequent invocations of the method with the arguments on the same event occurrence will be ignored. Event consumption essentially prevents an consumed event from participating in future event compositions.

**Definition 4-27** The method $\text{isConsumed}(E, \text{obj})$ of an event type determines whether the calling event occurrence is consumed by an occurrence of the event type $E$ with triggering object $\text{obj}$. An important characteristic of this method is that once an event occurrence is consumed, it remains consumed permanently. The value returned by the method depends only on its arguments, which can be regarded as

---

1. The name "most-recent context" is synonymous to the name "recent context" in Snoop. We prefer the first one because it conveys more precisely the semantics of composite event detection in this context.
object-identities. Therefore, the method can be evaluated once for each unique pair of arguments, and it can be evaluated incrementally.

We now specify two predicates meaningful for the chronicle context:

**Definition 4-28** *Occurred(E, E1, x1, E2, x2, ..., En, xn) ≡ Occurred-Unrestricted-Context(E, E1, x) ∧ Same-Object(x1, ..., xn) ∧ x1.changed ∧ ~ x1.isConsumed(E, Triggering-Object(x1, ..., xn)).* This predicate redefines the one in definition 4-14, and is to be used by the next predicate.

**Definition 4-29** *Chronicle-Context(E, E1, x1, E2, x2, ..., En, xn) ≡ ∀y [Occurred(E, E1, y, E2, x2, ..., En, xn) → Follows(y, x1)].* This predicate detects events in the chronicle context; it asserts that *x1* is an un-consumed event occurrence of type *E* in the unrestricted context and *x1* is the oldest one among all un-consumed event occurrences of type *E*.

To illustrate how events can be detected in the chronicle context, we formulate the detection rules for an event type in the following example.

**Example 4-2** The expression *E = And(E1, E2)* can be detected in the chronicle context with the following event detection rule:

**Condition:** *Enabled(E) ∧ Occurred-Unrestricted-Context(E, E1, x1) ∧ Occurred-Unrestricted-Context(E, E2, x2) ∧ ~ x1.isConsumed(E, Triggering-Object(x1, x2)) ∧ ~ x2.isConsumed(E, Triggering-Object(x1, x2)) ∧ Chronicle-Context(E, E1, x1, E2, x2) ∧ Chronicle-Context(E, E2, x2, E1, x1) ∧ Same-Object(x1, x2)*

**Action:** *x ← E.Raise(x1, x2), x1.Consume(E, x.obj), x2.Consume(E, x.obj)*

The main differences between this rule with the one for the most recent context are their parameter context qualifying predicates; also the above rule performs event consumption whereas the one for the most recent context does not.

**Continuous Context:**

This context is very similar to the unrestricted context in that event occurrences of the same type can induce multiple occurrences of a composite event. But unlike the unrestricted context, only un-consumed events can induce other event occurrences and events are consumed after they induce other event occurrences.

**Example 4-3** The expression *E = And(E1, E2)* can be detected in continuous context with the following event detection rules:

**Definition 4-30** The method *Consume-Constituent-Events()* of composite event types consumes all the constituent events of the calling event occurrence.
Rule E-Production:
Condition: $\text{Enabled}(E) \land \text{Occurred-Unrestricted-Context}(E, E1, x1) \land \text{Occurred-Unrestricted-Context}(E, E2, x2) \land x1.\text{changed} \land \neg x1.\text{isConsumed}(E, \text{Triggering-Object}(x1, x2)) \land x2.\text{changed} \land \neg x2.\text{isConsumed}(E, \text{Triggering-Object}(x1, x2)) \land \text{Same-Object}(x1, x2)$
Action: $x \leftarrow \text{E.Raise-Intermediate}(x1, x2)$

Rule A1 with priority = priority(E-Production) - 1:
Condition: $\text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x)$
Action: $x.\text{Consume-Constituent-Events}(), x.\text{state} = \text{MATCHABLE}$

**Cumulative Context:**

In this context, the parameters of all occurrences of each constituent event type are accumulated until the composite event is detected. All the occurrences of its constituent event types can no longer induce further occurrences of the composite event. The approach here to handle the cumulative context is to first detect the composite event in the unrestricted context and then accumulate occurrences of the composite event type into a single occurrence.

**Example 4-4** To illustrate detection in the cumulative context, we use the event expression $E = \text{Seq}(E1, E2)$ as an example. The necessary changes in the event detection rules of $E$ are discussed below:

Firstly, $E$ has to be detected in the unrestricted context to collect intermediate occurrences of $E$ by the use of the following event detection rule:

Rule E-Production:
Condition: $\text{Enabled}(E) \land \text{Occurred-Unrestricted-Context}(E, E1, x1) \land x1.\text{changed} \land \neg x1.\text{isConsumed}(E, \text{Triggering-Object}(x1, x2)) \land \text{Occurred-Unrestricted-Context}(E, E2, x2) \land x2.\text{changed} \land \neg x2.\text{isConsumed}(E, \text{Triggering-Object}(x1, x2)) \land \text{Follows}(x1, x2) \land \text{Same-Object}(x1, x2)$
Action: $x \leftarrow \text{E.Raise-Intermediate}(x1, x2), x.\text{state} = \text{INTERMEDIATE-OCCURRED}$

Secondly, additional event detection rules are included to accumulate parameters of intermediate occurrences and perform garbage-collection as depicted in the finite automaton of Figure 4-2. State transitions of each state are performed by an event detection rule: in state $S_0$, transitions are performed by rule A1; in state $S_1$ they are performed by rule A2 and A3; in state $S_2$ they are performed by rule A4 and A5.

Rule A1 with priority = priority(E-Production) - 1:
Condition: $\text{Enabled}(E) \land \text{Occurred-Intermediate}(E, x, \text{INTERMEDIATE-OCCURRED}) \land \neg \text{Occurred-Intermediate}(E, y, \text{INTERMEDIATE-ACCUMULATE})$
Action: $y \leftarrow \text{E.Raise-Intermediate}(), y.\text{state} = \text{INTERMEDIATE-ACCUMULATE}$
Figure 4-2 A finite automaton for detecting events in the cumulative context

Rule A2:
Condition: \( Enabled(E) \land Occurred-Intermediate(E, y, INTERMEDIATE-ACCUMULATE) \land Occurred-Intermediate(E, x, INTERMEDIATE-occurred) \land Same-Object(x, y) \)
Action: \( y.Add-Constituent-Event(x.Constituent-Events), y.ts = Min(y.te, x.te), y.te = Max(y.te, x.te), x.Consume-Constituent-Events(E, y.obj), x.state = INTERMEDIATE-CONSUMED \)

Rule A3:
Condition: \( Enabled(E) \land Occurred-Intermediate(E, y, INTERMEDIATE-ACCUMULATE) \land \sim (Occurred-Intermediate(E, x, INTERMEDIATE-occurred) \land Same-Object(x, y)) \)
Action: \( y.state = INTERMEDIATE-CLEANUP \)

Rule A4:
Condition: \( Enabled(E) \land Occurred-Intermediate(E, y, INTERMEDIATE-CLEANUP) \land Occurred-Intermediate(E, x, INTERMEDIATE-CONSUMED) \land Same-Object(x, y) \)
Action: \( x.Delete-Object() \)

Rule A5:
Condition: \( Enabled(E) \land Occurred-Intermediate(E, y, INTERMEDIATE-CLEANUP) \land \sim (Occurred-Intermediate(E, x, INTERMEDIATE-CONSUMED) \land Same-Object(x, y)) \)
Action: \( y.state = MATCHABLE \)

Theses rules are generic to every event type; they do not include event specific predicates. Therefore, it is concluded that every event type can be detected in the cumulative context with the modifications as outlined in the above examples.
4.5.2 Some Extra Event Operators

In this section, we demonstrate how new event operators can be constructed in the same framework modularly and effectively. Constructing a new event operator requires providing a set of event detection rules for the operator, as well as modifying the event specification parser. Our focus here is on the former step. Hence, only event detection rules for two new event operators are shown below. Their event detection rules are constructed with similar predicates as the existing event detection rules, yet their functionalities cannot be expressed with the existing event operators. The first new event expression \( \text{Count}(E1, n) \) specifies its occurrence at the \( n \)th occurrence of \( E1 \).

\[
E = \text{Count}(E1, n)
\]

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \neg (\text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x, x1)) \)

**Action:** \( y \leftarrow E.\text{Raise-Intermediate}(x1), y.\text{count} = 1 \)

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x, x1) \land x.\text{count} < (n - 1) \)

**Action:** \( x.\text{count} = x.\text{count} + 1 \)

**Condition:** \( \text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x, x1) \land x.\text{count} = (n - 1) \)

**Action:** \( x.\text{Add-Constituent-Event}(x1), x.\text{te} = x1.\text{te}, x.\text{state} = \text{MATCHABLE} \)

The second event operator \( \text{Periodic}^* \) corresponds to the cumulative version of the periodic event operator. Here, however, an additional parameter: \(<\text{action}>\) is required. Valid actions are any function/method that can be executed in the action part of a detection rule (i.e. production rule). An example of \(<\text{action}>\) is computing an aggregate value of one of the triggering object's attribute and storing it in the event occurrence, e.g. \( x.\text{max-price} = \text{Max}(x.\text{obj-price}, x.\text{max-price}) \). The detection rules for the following event expression resembles that of \( \text{Periodic} \) and \( \text{Aperiodic}^* \) events: rules 1a and 1b perform initialization, rules 2a and 2b perform periodic accumulation of parameters and rule 3 terminates the detection process and signals the occurrence of the \( \text{Periodic}^* \) event.

\[
E = \text{Periodic}^*(E1, F:<\text{action}>, E3)
\]
Rule 1a:
Condition: $\text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \neg (\text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x1, x)) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x1) \land \text{Same-Object}(x1, x3))$
Action: $x \leftarrow E.\text{Raise-Intermediate}(x1), x.\text{obj} = \text{Triggering-Object}(x, x1)$

Rule 1b:
Condition: $\text{Enabled}(E) \land \text{Occurred}(E, E1, x1) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Same-Object}(x1, x) \land \neg (\text{Match-Scalar-Time}(x1.\text{te} + F) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x1) \land \text{Same-Object}(x3, x1)))$
Action: $x.\text{ts} = x1.\text{te}, x.\text{te} = x1.\text{te}$

Rule 2a:
Condition: $\text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Match-Scalar-Time}(x.\text{next-alarm}) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3))$
Action: $<\text{action}>$

Rule 2b:
Condition: $\neg \text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Match-Scalar-Time}(x.\text{next-alarm}) \land \neg (\text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3))$
Action: $x.\text{next-alarm} = x.\text{te} + F$

Rule 3:
Condition: $\text{Enabled}(E) \land \text{Occurred-Any-Intermediate}(E, x) \land \text{Occurred}(E, E3, x3) \land \text{Absolute-Follows}(x3, x) \land \text{Same-Object}(x, x3)$
Action: $x.\text{Add-Constituent-Event}(x3), x.\text{te} = x3.\text{te}, x.\text{obj} = \text{Triggering-Object}(x, x3), x.\text{state} = \text{MATCHABLE}$

4.6 Comparison with Other Approaches

There are several approaches reported for processing rules and detecting a rich set of composite events. It is interesting to compare their differences and similarities with our approach. In this section, ODE [26, 29], SAMOS [22, 23], Sentinel [11], [43], Ariel [31], A-RDL [41], and Chimera [8] are compared with our production rule approach presented in this chapter.

Detecting composite events in ODE is done by finite state machines (FSMs). The attractive characteristics of FSMs are their low storage requirement and straightforward implementation. The production rule approach, on the other hand, requires much more storage for maintaining the discriminant network of the rule-based system. This requirement can be partially alleviated by flushing events in past transactions. Also, the actual impact of the high storage requirement on a database system is highly dependent on the architecture and implementation of the database system. On
the other hand, the advantage of the production rule approach over FSMs' is the intuitiveness, compactness and flexibility of production rules.

In SAMOS, petri-nets [33] are employed to detect composite events. Like FSMs, petri-nets require less storage than production rules. Petri-nets are more powerful and usually more compact than FSMs. But they are also less declarative and flexible than production rules.

In Sentinel, a data structure, namely an event graph, is used to detect composite events. The approach is purely procedural, and extending composite event detection functionalities would involve modifying the database system's internal modules. In contrast, extending reactive functionalities in the production rule approach can be done by adding event detection rules and/or overriding event class methods. Both can be done dynamically and without affecting other system components.

[43] presented an approach to introducing active behaviour into a relational DBMS extended with abstract data types. There, a rule is implemented by an activity which, like a task in Ada, is a program unit that runs in parallel with other units where the synchronization between them is done using the principle of rendezvous. Conditions and actions of a rule can be expressed as a combination of Ada's procedural constructs and SQL, and composite events are detected with rules. So a composite event is seen as a rule that waits for a rendezvous with its constituent events. The approach is similar to the production rule approach in two aspects. Firstly, the unit (which is an activity in [43] and a set of event detection rules in our approach) responsible for detecting a composite event is defined without affecting other units. Secondly, composite event detection is incremental. However, it is not addressed in [43] whether an activity can be dynamically defined. The ability to construct new event types without affecting running database applications is beneficial. In contrast, in the production rule approach, the event classes and their event detection rules can be defined and modified dynamically. The approach in [43] apparently detects composite events in the chronicle context only. Handling other contexts, such as the most recent context, may not be straightforward. This is because constituent events of a composite event in [43] are consumed in a First-Come-First-Serve manner, which is
also the rendezvous requests queueing policy. This policy is inherent to the programming language and thus it cannot be circumvented easily. On the other hand, production rules do not have this limitation and parameter contexts can be handled with relatively few extensions as illustrated in the previous section. Finally, the Ada approach tends to be procedural and imperative whereas the production rule approach is more declarative.

Ariel [31], A-RDL [41], and Chimera [8] are production rule-based systems that incorporate events into rule conditions. Their focus is on production rules rather than events. In particular, Ariel only allows primitive events in the rule condition part while A-RDL and Chimera only allow conjunction and disjunction of primitive events. To support more complex composite events and other advanced features of active databases such as temporal events and parameter contexts, significant extensions to them are needed.

All the approaches except that of [43] concern only composite event detection. Our production rule approach and that of [43], on the other hand, detect composite events and process rules within a uniform paradigm: in our approach, only production rules are utilized whereas in [43], only activities are employed. In view of the proliferation of active database systems employing different knowledge models and complex execution semantics, the ability to extend the existing functionalities (both event detection and rule processing) so as to accommodate new requirements is highly desirable. The production rule approach presented in this chapter can thus be regarded as an attempt to achieve such an ambitious goal.
Chapter 5

Architecture & Implementation

The implementation of ADOME-II is based on a loosely-coupled approach: a passive OODBMS is coupled with a production rule system without modifying either system. A software bridging-layer (written in C++) integrates these two independent systems and delivers a tightly-coupled interface. The facilities of underlying systems are heavily reused. In particular, the production rule system is used for both rule and event processing. This chapter describes various modules of the software bridging-layer and how the underlying components’ functionality is utilized. In the last section, a method which allows quick implementation of a query processor, which is currently lacking, is proposed.

5.1 Kernel Architecture

The implementation of the system is guided by two objectives: loosely-coupled architecture and extensibility. Thus, a production rule system (CLIPS [45]) is loosely coupled with a passive OODBMS (ITASCA [46]) to provide required functionalities. The advantages of such an architecture are:

1. Development effort is substantially reduced because the functionality of both systems can be re-used avoiding duplication.

2. Both systems can be independently changed; thus portability is enhanced. This characteristic is useful for heterogeneous DBMSs in which they can potentially be converted to provide a uniform interface for the purpose of interoperability.

The functional modules of the system kernel that realize the loosely coupled architecture are shown in Figure 5-1 (where the newly introduced and developed
modules are enclosed in the dotted polygon). The functionalities of each module are

![Image of a diagram showing the functional modules of the System Kernel]

**Figure 5-1** Functional Modules of the System Kernel

summarized below:

- **Object Access Interfaces.** This module consists of two interfaces: one for the production rule system and one for C++. The one for the production rule system effectively presents a uniform interface, as its native objects, for access to persistent objects. Thus, persistent objects not only can be accessed by production rules but also can be manipulated by the procedural language of the production rule system. On the other hand, the C++ interface allows dynamically created database classes/objects to be accessible in the statically typed language, enabling operations on database objects to be efficiently implemented. The last, but definitely not the least, function of this module is to generate events on object operations.

- **Cache Manager.** This module performs the usual functions related to storage of objects in the main memory, such as cache look-up, *pinning/un-pinning* objects, and object replacement. Due to the lack of control over physical DBMS storage operations, database data are cached on a per object basis.

- **Event Manager.** This module comprises the generation of abstract events, detection of composite events, rule triggering and definition and manipulation of events. Because of the approach to reactive processing introduced in chapter 4, this module relies on production rules to perform various functions.
• **Rule Manager.** This module provides various operations on rules such as definition, activation/deactivation, and association with events and objects. It also supports specific queries on rules, which can be issued by events or other objects. Operations on the rule agenda are also provided by the rule manager. Note that rule execution (cf. chapter 4 and section 3.7 of chapter 3), however, is not under the control of the rule manager. The execution is controlled by the rule engine of the CLIPS production rule system, and to a certain extent, through its procedural language.

### 5.2 Implementation of Role

The distinct feature of the role model is that it can be readily implemented in an object-oriented data model. Role types are modeled as classes and a role instance is created for each player of the role type. Dependencies among roles of the same player and associations between a player and its roles are modeled by composition links. The implementation of roles can be best illustrated in Figure 5-2 where the roles currently played by a person are shown. By modeling role types and role instances as first-class objects, not only can new operations be defined on them but also rules as well. Since object properties and relationships are visible, unanticipated usage of them can be easily accommodated, making the system extensible. An example of such unanticipated usage is illustrated below:

**Example 5-1** Suppose the role instance System-Administrator of a player A is to be delegated to a successor player B. This can be done by modifying appropriate object attribute values with normal database operations. The identity of this particular System-Administrator is thus retained and previous references to this administrator kept valid. Consequently, the notion of such a long-lived, player-independent role can be supported readily with only a few extensions to the system.

### 5.3 Rule & Event Mechanism

There are a number of issues to be addressed in implementing reactive capabilities, some of which are discussed below:
Figure 5-2 An Object Composition Diagram Illustrating the Implementation of Role

- **Rule, event and object associations.** Since rules and events are implemented as first-class objects, association of rules with events and objects, and visibility of rules in the context of roles can all be realized by different composition links as shown in Figure 5-3. The benefits of such an approach are similar to those for roles discussed in the preceding section.

- **Event generation and propagation.** System events are generated by using the wrapper approach. Each database operation that would generate an event is wrapped by a wrapper provided by the Object Access Interfaces; the wrapper collects transition values, if any, and invokes the method raise of the corresponding event class. The method raise creates an event instance if required.
and propagates the event instance to the immediate base event class by calling the base event class's method `raise`. Propagation ends when the event root class is reached.

- **Transition value representation.** Since there is a need to store transition values with arbitrary domain in an event instance, every transition value is stored as an ordered list of string with the first element being the domain string of the value and the remaining elements being the string representation of the transition value. Special methods are provided for format conversions.

- **Composite event detection and rule triggering and execution.** These functions are implemented by production rules, which are discussed in chapter 4.

- **Rule agenda.** The rule agenda is a repository of rules pending execution, which serves both internal and external purposes. Internally, it is an auxiliary data structure for rule execution; externally, it allows the state of rule execution to be examined. The rule agenda here is slightly different from that of A.I. production rule systems. Whereas in A.I. production rule systems an entry in the rule agenda may be removed as soon as its rule condition
becomes \textit{false}, here, entries stay in rule agenda until they are executed. Such behavior is realized by re-using the rule agenda facility of the native production rule system. Specifically, a class is introduced to store the ECA rules triggered and wrapper rules are used to initiate execution of application rules as described in section 4.2.1 of chapter 4. A snapshot of the rule agenda of the system is depicted in Figure 5-4.

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
Rule-Name & Event & Object & Trigger Time \\
\hline
"Rule A" & eid1 & oid1 & 1212432334 \\
"Rule A" & eid2 & oid1 & 1212432335 \\
"Rule B" & eid3 & oid2 & 1212435935 \\
\hline
\end{tabular}
\end{center}

Figure 5-4 Rule Agenda for ECA Rules

\section{5.4 Object Management}

The \textit{Object Access Interfaces} and \textit{Cache Manager} together serve as a bridge providing the mapping between objects in the cache and in the database. The \textit{Cache Manager} controls the storage of objects in main memory whereas the \textit{Object Access Interfaces} exports all object manipulation functions for internal modules and for applications. This section describes the implementation of both modules. For the \textit{Object Access Interfaces}, the focus is on the representation and mapping of objects.

\subsection{5.4.1 Object Representation}

When an object is cached, the cache image of the database object C (cf. Figure 5-5) is represented by two objects: object A, which is a CLIPS object, and object B, which is an instance of a C++ class in which most of the cache management logic is embedded. Every property of object C is available to applications via the message interfaces of object A. When object A receives a message, the message is channeled to object B where events are generated accordingly and updates, if any, are propagated to the database object.
The class hierarchy for representing database objects in C++ can be found in Figure 5-6 while that for representing objects in the production rule system is shown in Figure 5-7. Both representations allow database classes to be dynamically created.

Figure 5-5 Mappings of Object in Cache and Database

Figure 5-6 Representing Database Objects in C++

Figure 5-7 Representing Database Objects in the Production Rule System
deleted. The difference between the representation of an object in C++ and that in the production rule system is that the latter stores attribute values of the object while the one in C++ does not.

5.4.2 Cache Manager

As mentioned before, this module performs the usual functions related to storage of objects in main memory, such as cache look-up, pinning/un-pinning objects, and object replacement. When the cache is full and a new object is to be brought in, an object in the cache is chosen randomly and removed from the cache. This simple cache replacement policy has the benefits that no information about object accesses is required, and selection can be made quickly. Performance, which is the main purpose of using a cache in many systems, is not the primary objective here. Rather, the cache is used here to enable access to database objects uniformly as “native objects” from within the production rule system. The replacement policy of the cache is not expected to be a significant factor of the systems’s performance since the native DBMS already provides the primary caching of objects.

5.5 Implementation of the Query Language

Currently, users interact with the system through an interactive, procedural language interface, which is built-in to the production rule system. Although most of the kernel functionalities can be accessed by using the native procedural language, having a declarative query language is the ultimate goal. In this section, a method of implementing the query language is proposed. Specifically, queries are directly translated into procedures that perform equivalent functions.

5.5.1 Strength of the Native Procedural Language

Since the native procedural language is the building block of the query language, some of its strengths and their effects on the query language are highlighted below:
• *Computationally Complete*. The procedural language can not only manipulate objects individually, but it also allow various controls to be specified such as sequence, conditional branch and iteration. Various arithmetic and relational expressions can be specified. Thus, it facilitates implementation of associative queries.

• *Interpretive*. This enables queries implemented by the procedural language to be dynamically submitted and executed.

• *Extensible*. New native functions written in the native language can be defined and external functions (written in C) can be registered to the production rule system and invoked as other native functions.

• *Object-Oriented*. The similarity in the data model of the procedural language and that of ADOME-II facilitates the translation of the query language into the procedural language.

### 5.5.2 Translation of Query Language into Procedures

To implement the query language while at the same time maximizing reuse of components and reducing duplication of functionalities, a direct translation approach is proposed. The main benefit of such a translation approach is that it reduces the time of implementation of the query language. To give readers a flavor of the translation, the one for a sample query is shown in Table 5-1.

### 5.5.3 Query Processor

The query processor is the module that compiles and executes queries. A proposed architecture of the query language is depicted in Figure 5-8 and its components are briefly described below.

• *Interactive Query Language Interface*. This is the most fundamental and primitive interactive query language interface to the system for submitting ad-hoc queries. It is actually an application indistinguishable from other applications.
Table 5-1 Translation of the Query Language into the Native Procedural Language

<table>
<thead>
<tr>
<th>Native Procedural Language</th>
<th>ADOME-II Query Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(bind ?level1-col1 (create$))</td>
<td>select ?p from Person ?p</td>
</tr>
<tr>
<td>(progn$ (?p (Instances Person))</td>
<td>where ?p.age &lt; 18;</td>
</tr>
<tr>
<td>(if (&lt; (send (Obj ?p) get-age) 18)</td>
<td></td>
</tr>
<tr>
<td>then (insert$ ?level1-col1</td>
<td></td>
</tr>
<tr>
<td>(+ (length$ ?level1-col1) 1) ?p)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>(bind ?level1-col1 ?level1-col1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(progn$</td>
<td>foreach ?p in</td>
</tr>
<tr>
<td>(?p (progn (bind ?level1-col1 (create$))</td>
<td>(select ?p from Person ?p</td>
</tr>
<tr>
<td>(progn$ (?p (progn (Instances Person))</td>
<td>where ?p.age &lt; 18)</td>
</tr>
<tr>
<td>(if (&lt; (send (Obj ?p) get-age) 18)</td>
<td>do</td>
</tr>
<tr>
<td>then (insert$ ?level1-col1</td>
<td>Pretty-Print(?p);</td>
</tr>
<tr>
<td>(+ (length$ ?level1-col1) 1) ?p)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>(bind ?level1-col1 ?level1-col1)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>(Pretty-Print ?p)</td>
</tr>
</tbody>
</table>

a. The built-in functions of the procedural language are: create$: creates an ordered set; bind: assigns a value to a variable; progn$: performs actions on each element of a set; insert$: inserts an element to a set at a given position; length$: returns the size of a set; send: sends a message to an object;

NPL: Native Procedural Language

Figure 5-8 Query Language Modules

- **Query API.** This module initiates query compilation, execution, and returns query results to the application.

73
• **Query Compiler.** Instead of generating a query plan, the compiler directly generates the procedures for query execution in the CLIPS procedural language. During the course of compilation, the compiler may require some classes and objects for error checking.

5.6 **Current Status**

Currently, the *Object Access Interfaces* and the *Rule Manager* are completed. The *Cache Manager* can cache objects but flushing of objects in the cache is under application control. This is to maximize system performance during development. The only uncompleted component of the *Event Manager* is the translation of some of the event detection rules into CLIPS production rules. As a result, the functional event operators are *And, Or, Seq, Disjoint_Seq, Not, Aperiodic, After,* and *Within.* By using the C++ programming language interface, composite events based on these event operators can be defined/deleted. Also, the query language is not implemented. The only interactive interface to the system is the procedural language interface provided by the production rule system, through which, objects and roles can be manipulated. Moreover, operations such as role playing/unplaying, rule-event association and rule-object association can also be performed. All these operations generate events which trigger rules associated with them. The triggered rules can also be executed according to the E-C coupling modes (immediate and deferred) and the rule scheduling policy (rule priority and time of activation). The condition and action part of a rule can be any valid function of the procedural language interface, which includes the functions exported by the ADOME-II system.
Chapter 6

Conclusions & Future Work

6.1 Conclusions

To extend the scope of ADOME so as to better support active aspects of NGISs and their applications, reactive capabilities have been introduced into it, resulting in the successor ADOME-II. In this thesis, the data/knowledge model of ADOME is extended to incorporate reactive capabilities without sacrificing the flexibility of rule sharing in ADOME. Further, the thesis also shows how the reactive capabilities can be integrated into the architecture of ADOME in an extensible manner, which is crucial for the system to be able to evolve in accommodating dynamic/unanticipated NGIS applications’ requirements.

Thus, the main contributions of this thesis have been on two related aspects: introducing reactive capabilities into the data/knowledge model of ADOME and an extensible approach to reactive processing. On the first aspect, a rule model is proposed that allows rules to be flexibly and dynamically shared by arbitrary entities, transcending the restriction imposed by the class hierarchy on rule inheritance. The need for this kind of flexible rule sharing over rule inheritance is discussed in section 3.2 of chapter 3. Another characteristic of the rule model is that the applicability of a rule (on an entity) is independent of the triggering event types of the rule. For example, the rule “Student-Performance-Review-Rule” can be applicable to undergraduates, postgraduates, local or foreign students, which is independent of the event types that trigger the evaluation of the rule. Such independence enables rules to be manipulated without knowledge of their triggering event types. Apart from the issues related to the use of rules, a rich set of composite events is supported to enhance the ability of the system in modeling complex event-driven activities, so as to better support CSCW applications.
On the second aspect, it is shown how a rich set of composite event expressions can be detected and how rules can be processed uniformly within a single paradigm—production rules. Detection of composite events is performed by a set of event detection rules, and sets of event detection rules for all event expressions are described. The use of event detection rules allows new reactive capabilities such as new composite event expressions to be introduced into the system in a declarative and compact manner, thus improving the extensibility of the system.

Finally, the system has a loosely-coupled architecture which allows the development to be done cost-effectively by reusing individual components. From a software engineering perspective, the ability to rapidly prototype the system, and the extensibility of the system are crucial for such experimental prototypes as ADOME and ADOME-II.

6.2 Future Work

6.2.1 Enhancement of the Prototype

An immediate extension to the system is an implementation of the query language. A method of implementing the query language is outlined in section 5.5 of chapter 5. Optimization of queries is not an immediate issue although it is worthwhile to investigate when the system is to be used on large databases.

The complexity of designing/maintaining ECA rules has been recognized. Because of the complex execution semantics of rule execution, it is difficult to foresee the interaction of rules even for a moderate number of rules. The problem is even more serious in ADOME-II where an object can play multiple roles, each of which may have a set of rules attached. As a first step, a graphical user interface (GUI) for visualization of rules, events and roles, and their relationships would significantly alleviate the problem.

Another enhancement to the system is the concurrency control. Currently, only sequential transactions are supported. Cache management for sequential transactions is much simpler than that of concurrent transactions since in concurrent transactions,
the objects in the cache may belong to multiple transactions. Although concurrent access to objects (including rules and events) in the database is controlled by the DBMS, that of objects in the cache is not. One possible architecture supporting concurrent transactions is to have one system process for each transaction. Thus the cache in a system process is only affected by one transaction at a time. However, the caches of different system processes would not be synchronized. Thus, the issue of cache coherency has to be solved first before such an architecture is realized. Other architecture alternatives may also be feasible and further investigation is needed.

6.2.2 Computer Supported Cooperative Work

In [38], the requirements for CSCW were classified into various perspectives and the correspondence between ADOME’s facilities and these perspectives was established. The role facilities of ADOME were shown to be versatile in accommodating various modelling perspective of CSCW such as organization modelling perspective, resource modelling perspective and data/knowledge modelling perspectives. The deficiency of ADOME is in the activity modelling perspective. Regarding this aspect, ADOME-II has the reactive capabilities required to support activity execution by active rules. Further extensions to ADOME-II are however required to support activity modelling. In particular, a knowledge model for activities is required which describes the structure of activities and the relationships/dependencies among them and with other components. An execution model for the activities is also required, which concerns issues like concurrency control of activity execution and recovery (backward and/or forward) of activities. To interface the resultant system with applications/users, a higher-level specification/manipulation language than the rule language is needed.
Appendix A

Translation of Composite Event Detection Rules into CLIPS Production Rules

Chapter 4 introduces the framework and derivation of composite event detection rules. This appendix lists the translations of event expressions to CLIPS production rules. The syntax of CLIPS production rules are described in [45].

The set of event detection rules for an event expression is presented in the following structure:

```<Event-Expression>
<Macro-Definitions>
<=>
<Rules>
```

The `<Event-Expression>` denotes the name and parameters of an event expression. The `<Macro-Definitions>` contains a set of local macros that are used in the production rules. Note that these macros are not part of the CLIPS language and they are used only to facilitate the reading of the material. The `<rules>` contains the event detection rules in CLIPS syntax. Note that some variables, which contain indices, are not in CLIPS syntax, e.g., the variable "?obj(i)", which contains an index "i", is not in CLIPS syntax. The parentheses are only for enhancing the readability of the variable. During translation, the parentheses will be removed and the index "i" will be replaced by an actual value.

We firstly list the macros that are common to all event detection rules:

```
MATCH_EVENT_TYPE(E):
?E<-{(object (is-a meta-E) (name [class-object-E]) (ACTIVE-TIME ?Eat) (isDisabled 0))
MATCH(Ei):
  // If Ei is dynamically bound, then ?obj(i) will be replaced by ?obj
  ?ei<-(object (is-a Ei) (STATE MATCHABLE) (OBJ ?obj(i)) (TS $?tsi)
  (TE $?te(i)&:(<= (Time $?te(i)) ?Eat)))
MATCH_MOST_RECENT(Ei):
  // If Ei is dynamically bound, then ?obj(i) will be replaced by ?obj
MATCH(Ei) ^
  (forall (object (is-a Ei) (STATE MATCHABLE) (OBJ ?obj(i))) (TS $?tsi)
  (TE $?te(i)&:(<= (Time $?t) ?Eat)))
```

78
(or (test (isEarlier $?t $?te(i))) (test (eq $?te(i) $?t))))

MIN_START_TIMESTAMP(e1,e2,...,en):
\(\min\ (\text{Time } {?ts}(1))\ (\text{Time } {?ts}(2))\ ...\ (\text{Time } {?ts}(n))\)

MAX_END_TIMESTAMP(e1,e2,...,en):
\(\max\ (\text{Time } {?te}(1))\ (\text{Time } {?te}(2))\ ...\ (\text{Time } {?te}(3))\)

MAKE_NEW_OCCURRENCE(E, e1,e2,...,en):
// RaiseComplex will check if e1,...,en are triggered by the same object.
// Object-independent events are excluded from the check.
// Also note that time-triggered events are considered triggered by an
// object with NULL OID.
(bind ?occ
  (send ?E RaiseComplex ?obj MIN_START_TIMESTAMP(e1,...,en)
    MAX_END_TIMESTAMP(e1,...,en) e1 e2 ... en )
  (send ?occ put-STATE MATCHABLE))

From here onwards, the event detection rules of individual event expression are
listed:

\[ E = \text{AND}(E_1,E_2,...,E_n) \]

\[ \text{AND} \]

Rule 1:
Condition: \(\text{MATCH}_{\text{EVENT \_ TYPE}}(E) \land \text{MATCH}_{\text{MOST \_ RECENT}}(E_1) \land \ldots \land \text{MATCH}_{\text{MOST \_ RECENT}}(E_n)\)
Action : MAKE_NEW_OCCURRENCE\(E,e_1,...,e_n\)

\[ E = \text{OR}(E_1,E_2,...,E_n) \]

MATCH\(E_i)\):
// If \(E_i\) is dynamically bound, then \(?obj(i)\) will be replaced by \(?obj\)
\(e_i\)\-(object \(\text{is-a} E_i\) \(\text{STATE MATCHABLE}\) \(\text{OBJ} \?obj(i)\) \(\text{TS} {?ts}(i)\)
  (TE $?te(i)@$:<(\text{Time } {?te(i)} \?E@) (TS $?ts(i)@))

\[ \text{OR} \]

Rule 1:
Condition: \(\text{MATCH}_{\text{EVENT \_ TYPE}}(E) \lor \ldots \lor \text{MATCH}_{\text{MOST \_ RECENT}}(E_i) \lor \text{MATCH}_{\text{MOST \_ RECENT}}(E_n)\)
Action : MAKE_NEW_OCCURRENCE\(E,e_1\)

\[ E = \text{SEQ}(E_1,E_2,...,E_n) \]

FOLLOWED_BY\(E_i,E_j)\):
(test (isEarlier $?te(i) $?te(j)))

\[ \text{SEQ} \]

Rule 1:
Condition: \(\text{MATCH}_{\text{EVENT \_ TYPE}}(E) \land \text{MATCH}_{\text{MOST \_ RECENT}}(E_1) \land \text{MATCH}_{\text{MOST \_ RECENT}}(E_2) \land \ldots \land \text{MATCH}_{\text{MOST \_ RECENT}}(E_n)\)
Action : MAKE_NEW_OCCURRENCE\(E,e_1\)

\[ E = \text{DISJOINT} \_ \text{SEQ}(E_1,E_2,...,E_n) \]
DISJOINT_FOLLOWED_BY(Ei, Ej):
(test (isEarlier $?te(i) $ts(j))

<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_EVENT_TYPE(E1)
  ^ MATCH_EVENT_TYPE(E2) ^ DISJOINT_FOLLOWED_BY(E1, E2)
    ^ MATCH_EVENT_TYPE(En) ^ DISJOINT_FOLLOWED_BY(En-1, En)
Action : MAKE_NEW_OCCURENCE(E, en)

E = NOT(E1, E2, E3)
// Non occurrence of E2 within the closed interval [E1,E3]. Signaled at E3
MATCH_BETWEEN(Ei, Ej, Ek):
?ej<-(object (is-a Ej) (STATE MATCHABLE) (OBJ ?obj(j)) (TS $?tsj)
  (TE $?te(j)&:(>= (Time $?te(j)) ?Eat)
   &:(isEarlier $?te(i) $?te(j))&:(isEarlier $?te(k) $?te(j))))

<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_EVENT_TYPE(E1)
  ^ MATCH_EVENT_TYPE(E3) ^ FOLLOWED_BY(E1, E3)
  ^ (not MATCH_BETWEEN(E1, E2, E3))
Action : MAKE_NEW_OCCURENCE(E, ei, e3)

E = PERIODIC(E1, F, E3)
// Occurs with frequency F within [E1, E3]. F is a string of "s:m:h d"
// State of E: INTERMEDIATE-SEC, INTERMEDIATE-MIN, INTERMEDIATE-HOUR,
// INTERVAL-DAY, MATCHABLE.
CURRENT_TIME:
  (object (is-a TimeVector) (name [SystemClock]) (time $now))
CURRENT_YEAR:
  (object (is-a TimeVector) (name [SystemClock]) (year $year))
CURRENT_MONTH:
  (object (is-a TimeVector) (name [SystemClock]) (month $month))
CURRENT_DAY:
  (object (is-a TimeVector) (name [SystemClock]) (day $day))
CURRENT_HOUR:
  (object (is-a TimeVector) (name [SystemClock]) (hour $hour))
CURRENT_MIN:
  (object (is-a TimeVector) (name [SystemClock]) (min $min))
CURRENT_SEC:
  (object (is-a TimeVector) (name [SystemClock]) (sec $sec))
MATCH_ANY_INTERMEDIATE_EVENT(Ei):
?ei<-(object (is-a Ei) (STATE INTERMEDIATE-SEC|INTERMEDIATE-MIN|INTERMEDIATE-HOUR|INTERMEDIATE-DAY)
  (OBJ ?obj(i)) (NEXT-ALARM ?a(i))
  (TE $?tei&:(>= (Time $?tei) ?Eat)))
MATCH_DAY(Ei):
  CURRENT_DAY ^
?ei<-(object (is-a Ei) (STATE INTERMEDIATE-DAY)
  (NEXT-ALARM ?a(i))&:(Same Day (Now) ?a(i))
  (TE $?tei&:(>= (Time $?tei) ?Eat)))
MATCH_HOUR(Ei):
  CURRENT_HOUR ^
?ei<-(object (is-a Ei) (STATE INTERMEDIATE-HOUR)
  (NEXT-ALARM ?a(i))&:(Same Hour (Now) ?a(i))
  (TE $?tei&:(>= (Time $?tei) ?Eat)))
MATCH_MIN(Ei):
CURRENT_MIN ^
?ei<=(object (is-a Ei) (STATE INTERMEDIATE-MIN)
  (NEXT-ALARM ?a(i)&:(Same Min (Now) ?a(i))
  (TE $??tei&:(>= (Time $??te(i)) ?Eat)))
MATCH_SEC(Ei):
CURRENT_SEC ^
?ei<=(object (is-a Ei) (STATE INTERMEDIATE-SEC) (OBJ ?obj(i))
  (NEXT-ALARM ?a(i)&:(<= ?a(i) (Now))
  (TE $??tei&:(>= (Time $??te(i)) ?Eat)))
MATCHSAME_MOST_RECENT_EVENT(Ei)
?ei<=(object (is-a Ei) (STATE MATCHABLE) (OBJ ?obj(i))
  (TE $??tei&:(>= (Time $??tei) ?Eat)))
(forall (object (is-a Ei) (STATE MATCHABLE) (OBJ ?obj(i))
  (TE $??t&:(>= (Time $??t) ?Eat)) (TS $??ts(i))
  (test (or (isEarlier $?t $?te(i)) (eq $??te(i) $?t)))
  MAKE_INTERMEDIATE_OCCURRENCE(E, e1,e2,...,en)
  (bind ?occ (send ?E RaiseComplpex ?obj MIN_START_TIMESTAMP(e1,...,en)
  MAX_END_TIMESTAMP(e1,...,en) e1 e2 ... en) )

<=>

Rule 1
Condition: MATCH_EVENT_TYPE(E)^ MATCH_MOST_RECENT(E1)
Actions : MAKE_NEW_OCCURRENCE(E, e1)
          MAKE_INTERMEDIATE_OCCURRENCE(E, e1)
          (send ?occ put-NEXT-ALARM (+ (Time $??te(1)) (Scalar F)))
          (send ?occ put-STATE_INTERMEDIATE-DAY)

Rule 2a:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_DAY(E)
Action : (send ?eo put-STATE_INTERMEDIATE-HOUR))

Rule 2b:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_HOUR(E)
Action : (send ?eo put-STATE_INTERMEDIATE-MIN))

Rule 2c:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_MIN(E)
Action : (send ?eo put-STATE_INTERMEDIATE-SEC))

Rule 2d:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_SEC(E)
Action : (send ?eo put-TE (MakeTimeVector (Now)))
          (send ?eo put-STATE MATCHABLE)
          MAKE_INTERMEDIATE_OCCURRENCE(E, e1)
          (send ?occ put-NEXT-ALARM (+ (Time $??te(1)) (Scalar F)))
          (send ?occ put-STATE_INTERMEDIATE-YEAR)

Rule 3:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_ANY_INTERMEDIATE_EVENT(E)
  ^ MATCH_MOST_RECENT_EVENT(E3) ^ CURRENT_TIME ^ (> ?now ?ao)
Action : (send ?eo DeleteObject)

E = APERIODIC(E1, E2, E3)
/* Occurs when E2 occured within [E1, E3] */

<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E)^ MATCH_MOST_RECENT(E1)
MATCH_MOST_RECENT(E1, E2, E3)

Action: MAKE_NEW_OCCURRENCE(E1, E2, E3)

E = APERIODIC*(E1, E2, E3)

// Occurs when E1 occurred followed by E3, all intermediate E2s are
// accumulated.
MATCH_WITHIN(E1, E2, E3):
// If E1 is dynamically bound, then ?obj will be replaced by ?obj
// E2 <-> object (is-a E1) (STATE MATCHABLE) (OBJ ?obj)
// (TS $ts)
// (TE $?te) & (isEarlier $?ts ?te)
// (isEarlier $?te ?ts)
MATCH_INTERMEDIATE_EVENT(E1):
// If E1 is dynamically bound, then ?obj will be replaced by ?obj
// E2 <-> object (is-a E1) (STATE MATCHABLE) (OBJ ?obj)
// (TE $?te) & (isEarlier $?ts ?te)

<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_MOST_RECENT(E1)
Action: MAKE_INTERMEDIATE_OCCURRENCE(E1, E2)

Rule 2:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_INTERMEDIATE_EVENT(E)
^ MATCH_MOST_RECENT(E2, E3) ^ FOLLOWED_BY(E, E2)
Action: (send ?eo add-CONSTITUENT-EVENTS ?e2)
(snd ?eo put-TE $?te)

Rule 3:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_INTERMEDIATE_EVENT(E)
^ MATCH_MOST_RECENT(E3) ^ FOLLOWED_BY(E, E3)
Action: (send ?eo add-CONSTITUENT-EVENTS ?e3)
(snd ?eo put-TE $?te)
(snd ?eo put-STATE MATCHABLE)

E = EVERY(E1, s:m:h w/d/m/y*, E3)

MATCH_TIME:
// The patterns in [ ] are optional. They are omitted when the corresponding
// components in the time pattern are wild-cards. This information can be
// determined at compile time. Also note that omitting a pattern reduces
// the frequency of matching of a pattern, e.g., with a time pattern
// having only the hour as non-wild-card, the following pattern will be
// matched once every hour.
// (object (is-a TimeVector) (name [SystemClock])
//   [sec s] [min m] [hour h] [month m] [year y]
//   [day-of-week wd] [day-of-month d]
}

<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_MOST_RECENT(E1)
Action: MAKE_INTERMEDIATE_OCCURRENCE(E1, E2)

Rule 2:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_INTERMEDIATE_EVENT(E)
^ MATCH_TIME
Action: (send ?eo put-TE (MakeTimeVector (Now))))

82
Rule 3:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_INTERMEDIATE_EVENT(E)
           ^ MATCH_MOST_RECENT(E3) ^ FOLLOWED_BY(E,E3) ^ (not MATCH_TIME)
Action   : (send ?eo DeleteObject)

E = AT("s:m:h wd d/m/y")
// Occurs at the given absolute time pattern. Once it is triggered,
// it will never be triggered again.
T: Scalar("s:m:h wd d/m/y")
MATCH_ABSOLUTE_YEAR(E,t):
  CURRENT_YEAR ^ (<= t (Now))
MATCH_ABSOLUTE_MONTH(E,t):
  (object (is-a E) (STATE INTERMEDIATE-MONTH)) ^ CURRENT_MONTH ^ (<= t (Now))
MATCH_ABSOLUTE_DAY(E,t):
  (object (is-a E) (STATE INTERMEDIATE-DAY)) ^ CURRENT_DAY ^ (<= t (Now))
MATCH_ABSOLUTE_HOUR(E,t):
  (object (is-a E) (STATE INTERMEDIATE-HOUR)) ^ CURRENT_HOUR ^ (<= t (Now))
MATCH_ABSOLUTE_MIN(E,t):
  (object (is-a E) (STATE INTERMEDIATE-MIN)) ^ CURRENT_MIN ^ (<= t (Now))
MATCH_ABSOLUTE_TIME(E,t):
  (object (is-a E) (STATE INTERMEDIATE-SEC)) ^ CURRENT_TIME ^ (<= t (Now))
<=>

Rule 1:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_YEAR(E,T)
Action   : MAKE_INTERMEDIATE_OCCURENCE(E)
           (send ?occ put-STATE INTERMEDIATE-MONTH)

Rule 2a:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_MONTH(E,T)
Action   : (send ?occ put-STATE INTERMEDIATE-DAY)

Rule 2b:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_DAY(E,T)
Action   : (send ?occ put-STATE INTERMEDIATE-HOUR)

Rule 2c:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_HOUR(E,T)
Action   : (send ?occ put-STATE INTERMEDIATE-MIN)

Rule 2d:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_MIN(E,T)
Action   : (send ?occ put-STATE INTERMEDIATE-SEC)

Rule 2e:
Condition: MATCH_EVENT_TYPE(E) ^ MATCH_ABSOLUTE_TIME(E,T)
Action   : (bind ?x (MakeTimeVector ?now))
           (send ?occ put-TS ?x)
           (send ?occ put-TE ?x)
           (send ?occ put-STATE MATCHABLE)
           (send ?E put-ISDISABLED TRUE)
REFERENCES


ment of Computer Science, The Hong Kong University of Science and Technology, 1995.


