

# User-Oriented Rule Management for Complex Event Processing Applications

DISSERTATION

zur Erlangung des akademischen Grades

**Doktor der technischen Wissenschaften**

eingereicht von

**Hannes Obweger**

Matrikelnummer 0425962

an der  
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Christian Huemer

Diese Dissertation haben begutachtet:

---

(Ao.Univ.Prof. Mag. Dr.  
Christian Huemer)

---

(Ao.Univ.Prof. Dipl.-Ing. Mag.  
Dr. Stefan Biffl)

Wien, 15.12.2011

---

(Hannes Obweger)

# User-Oriented Rule Management for Complex Event Processing Applications

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktor der technischen Wissenschaften**

by

**Hannes Obweger**

Registration Number 0425962

to the Faculty of Informatics  
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Christian Huemer

The dissertation has been reviewed by:

---

(Ao.Univ.Prof. Mag. Dr.  
Christian Huemer)

---

(Ao.Univ.Prof. Dipl.-Ing. Mag.  
Dr. Stefan Biffl)

Wien, 15.12.2011

---

(Hannes Obweger)



---

## Erklärung zur Verfassung der Arbeit

Hannes Obwegger  
Staudingergasse 13/24, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

---

(Ort, Datum)

---

(Unterschrift Verfasser)



---

## Abstract

In recent years, Complex Event Processing (CEP) [71] emerged as a new paradigm for monitoring business environments and automated, event-driven decision making. Application thereof are manifold, ranging from logistics to fraud detection and automated trading. The business model behind CEP is Sense and Respond (S&R) as proposed by Stephan Haeckel [49]. It is rooted in the idea that purposeful adaptive system design is more effective to deal with discontinuities and fast-moving industry environments as compared to traditional plan-and-execute strategies.

One of the key elements of almost any CEP application are *event-pattern rules*, which have been called “the foundation for successful applications of CEP” [71]. Describing event-processing logic of the form

**if** an event pattern  $p$  is detected, **then** execute action(s)  $A$

these rules showed to be useful across different conceptual layers of a system: On the level of low-level (pre-)processing and integration logic, event-pattern rules can be applied to continuously filter, transform, and aggregate events as emerging from underlying source systems. On the level of high-level business logic, event-pattern rules allow detecting noteworthy business situations and acting on their occurrence in near real time.

With the ongoing move of CEP towards the mainstream of enterprise computing, more and more persons are involved in the design, creation, and maintenance of event-driven applications. This raises concerns about the usability and manageability of CEP within the organizational framework conditions of an enterprise: Given larger and more heterogeneous user groups, it becomes increasingly important that the different event-pattern rules of an application can be created, deployed, and administrated by responsible and qualified personnel. Business logic, on the one hand, requires deep knowledge of the business environment, and will typically be in the responsibility of domain experts. Processing logic, on the other hand, requires a detailed understand-

ing of the given CEP framework and its integration with underlying source systems, and will typically be managed by IT experts. As a consequence, a comprehensive approach to *rule management* for CEP applications must provide workflows and tools that are tailored to the particular skills, competences, and responsibilities of these user groups.

This thesis contributes to the field of Complex Event Processing a novel approach to user-oriented rule management. Our approach caters to the needs of IT experts as well as technically inexperienced business users, for which complementary, yet clearly decoupled workflows are presented. These workflows are rooted in a conceptual differentiation of event-pattern rules by their general function within an event-based application: *Infrastructural rules*, on the one hand, prepare data for other parts of the application, but do not by themselves respond to the underlying business environment. *Sense-and-respond rules*, on the other hand, set up on the resulting, readily-preprocessed event-based image of underlying source systems and act on noteworthy business situations by directly or indirectly triggering actions in the business environment.

In the proposed framework, IT experts define infrastructural rules in a single, comprehensive model, in parallel and fully integrated with the other elements of an application's event-processing infrastructure. Sense-and-respond rule management, by contrast, builds upon a sophisticated system of configurable building blocks of pattern-detection and reaction logic. Prepared by technically versed domain experts according to the general requirements of an application scenario, these building blocks can be assembled to concrete event-pattern rules by technically inexperienced business users in a way that entirely abstracts from underlying complexity. The proposed framework has been successfully implemented as part of the general-purpose event-processing framework Sense-and-Respond Infrastructure (SARI) [114]. We present the key architectural elements of this implementation and elaborate on the front-end tools of our system.

Further contributions of this thesis are novel approaches to *entity-based state management* and *hierarchical pattern modeling*. These approaches were designed to naturally complement the proposed approach to user-oriented rule management and have been successfully implemented as part of SARI.

Entity-based state management addresses the problem of monitoring a complex, durable entity – e.g., a counter, a customer, or a queue – which state is accessible only in the form of continuous, low-level update events. We extend SARI by the concept of *business entity providers*, which encapsulate arbitrary state-calculation logic and manage state in the form of typed, application-wide, identifiable data structures. These so-called *business entities* can then be updated and monitored for exceptional states using event-pattern rules. Using entity-based state management, the proposed rule-management framework becomes applicable also in entity-centric environments, which are difficult, if not impossible, to approach with purely event-based strategies.

Hierarchical pattern modeling facilitates reuse of pattern-detection logic on the level of individual event patterns, which otherwise have to be modeled in separate, potentially redundant decision graphs. Reusability is achieved through special language elements – so-called sub-pattern components – which serve as references to sub-level pattern-detection logic in the super-level event patterns in which they are used. These elements can be integrated with the other elements of SARI’s graphical event-pattern language in an accustomed workflow and configured to the given application context through input parameters. Tailored evaluation strategies enable high-performance event processing as well as arbitrary nestings of pattern-detection logic.

Our research is framed by a novel, model-driven view on SARI. Splitting the overall complexity of a SARI application into smaller, easier-to-understand sub-models, this view aims to serve as a basis for future research and facilitate communication, interchange, and cooperations within the event-processing community.

The presented contributions are thoroughly evaluated for technical feasibility, applicability, and utility in real-world business scenarios. Technical feasibility of our concepts is proofed by their successful implementation within SARI. Applicability is demonstrated using an exemplary SARI application for event-based service assurance, where SARI is applied as an extension to a widespread workload automation and job scheduling platform. Utility in practical business environments has been investigated in a case study, conducted at a leading manufacturer of agricultural machinery.





---

## Kurzfassung der Dissertation

Complex Event Processing (CEP) [71] ermöglicht die kontinuierliche, ereignisbasierte Überwachung von komplexen Systemen. Relevante Ereignisse in der Geschäftsumgebung – etwa der Eingang einer Bestellung, das Fehlschlagen eines Transportprozesses oder das Versenden einer E-Mail – werden dabei in naher Echtzeit verarbeitet und anhand kausaler, temporaler oder räumlicher Beziehungen zueinander in Kontext gesetzt. Auf Basis dieser Relationen können schließlich geschäftsrelevante Informationen hergeleitet werden, welche im Falle einer getrennten Verarbeitung der einzelnen Ereignisse nicht ersichtlich wären. Mögliche Anwendungsgebiete für CEP finden sich u.A. in den Bereichen Logistik und Betrugserkennung, sowie im automatisierten Aktienhandel.

Ein zentrales Element vieler CEP-Applikationen sind sogenannte Ereignisregeln. Diese Regeln beschreiben Ereignisverarbeitungslogik der Form

**Wenn** ein Beziehungsmuster (event pattern)  $p$  erkannt wird,  
**dann** führe Aktion(en)  $A$  aus

und können auf unterschiedlichen konzeptuellen Ebenen einer Applikation eingesetzt werden. Im Bereich der Vorverarbeitungs- und Integrationslogik werden Ereignisregeln für das Filtern, Transformieren, und Aggregieren von Ereignisdaten eingesetzt. Im Bereich der Geschäftslogik ermöglichen Ereignisregeln die automatisierte Erkennung von relevanten Geschäftssituationen. Als Reaktion auf solche Situationen können dann entsprechende (Gegen-)Maßnahmen angestoßen werden.

Mit der fortschreitenden Verbreitung von CEP wird es aus Unternehmenssicht zunehmend bedeutsamer, dass die verschiedenen Ereignisregeln einer CEP-Anwendung durch entsprechend qualifiziertes Personal erzeugt, angewandt und verwaltet werden können. Vorverarbeitungs- und Integrationslogik

---

<sup>0</sup> Die in dieser Kurzfassung gewählten Fachbegriffe entsprechen der Terminologie Bruns und Dunkels [15].

benötigt detailliertes Wissen über die Gesamtarchitektur einer Applikation, sowie die Einbindung dieser Applikation mit darunterliegenden Quellsystemen. Solche Kenntnisse bestehen üblicherweise in der IT-Abteilung eines Unternehmens, sodass die Erstellung und Verwaltung entsprechender Ereignisregeln von technisch-versierten Mitarbeitern betrieben werden sollte. Im Gegensatz dazu benötigt Geschäftslogik das Wissen von Domänenexperten: Welche Situationen stellen eine Gefahr oder Chance dar, und welche erfordern keine weitere Beachtung? Welche Aktionen sollen im Falle einer bestimmten Geschäftssituation getätigt werden? Ein gesamtheitliches, benutzergruppenübergreifendes Regelverwaltungssystem für CEP-Anwendung muss daher für IT-Experten ebenso wie für technisch unerfahrene Domänenexperten effizient benutzbar sein.

Die vorliegende Dissertation beschreibt ein neuartiges Regelverwaltungssystem für CEP-Anwendungen. Dieses System ermöglicht es IT-Experten ebenso wie Domänenexperten, Ereignisregeln entsprechend ihrer spezifischen Fähigkeiten und Verantwortungen zu erstellen, auszuführen und zu administrieren. Das vorgestellte Framework basiert dabei auf einer konzeptuellen Unterscheidung von Ereignisregeln nach ihrer grundsätzlichen Funktion innerhalb einer CEP-Anwendung: Die Gruppe der sogenannten *Infrastrukturregeln* beinhaltet all jene Regeln, welche Daten für andere Teile einer CEP Anwendung vorbereiten, jedoch nicht selbstständig in die Geschäftsumgebung zurückwirken. Im Gegensatz dazu beinhaltet die Gruppe der sogenannten *Sense-and-Respond-Regeln* all jene Regeln, die direkt oder indirekt in das Geschäftsumfeld zurückwirken, dabei aber keine Daten für andere Teile der Applikation bereitstellen. Für die beschriebenen Gruppen werden sich gegenseitig ergänzende, jedoch sauber getrennte Regelverwaltungsstrategien vorgestellt.

Infrastrukturregeln werden im vorliegenden Framework als vollständige, unmittelbar ausführbare Artefakte modelliert. Technisch-versierte Benutzer erzeugen und verwalten diese Artefakte gemeinsam mit den anderen Elementen der sogenannten Ereignisverarbeitungsinfrastruktur einer CEP-Anwendung, also all jenen Elementen, die eine vollständige, ereignisbasierte Abbildung der darunterliegenden Geschäftsumgebung erzeugen ohne selbst in diese zurückzuwirken. Im Gegensatz dazu basiert die Erzeugung und Verwaltung von Sense-and-Respond-Regeln auf einem Bausteinsystem konfigurierbarer Mustererkennungs- und Reaktionslogik. Diese Bausteine werden von technisch-versierten Benutzern vorbereitet und können von Domänenexperten ohne technisches Hintergrundwissen zu konkreter Geschäftslogik zusammengesetzt werden. Das vorgestellte Framework wurde als Erweiterung zum generischen Ereignisverarbeitungsframework Sense-and-Respond Infrastructure (SARI) [114] erfolgreich implementiert. Implementierungsdetails werden im Zuge dieser Dissertation ausführlich beschrieben.

Die vorliegende Dissertation beschreibt weiters neuartige Ansätze zu *Entitäten-basierter Zustandsverwaltung* und *hierarchischer Muster-Modellierung*.

Diese Konzepte ergänzen das vorgestellte Regelverwaltungssystem und wurden als Teil von SARI implementiert.

Entitäten-basierte Zustandsverwaltung ermöglicht die Überwachung von dauerhaften, komplexen Entitäten wie etwa Zählern, Benutzerkonten, oder Warteschlangen, selbst wenn der gegenwärtige Zustand dieser Entitäten ausschließlich als eine Folge inkrementeller Änderungsereignisse verfügbar ist. Der vorgestellte Ansatz erweitert SARI um sogenannte *Business-Entity-Provider*; diese – als weitestgehend frei implementierbare Plugins konzipierten – Komponenten kapseln beliebige Zustandsberechnungen und verwalten Zustände als applikationsweite, eindeutig identifizierbare Datenstrukturen. Die verwalteten Datenstrukturen – in weiterer Folge als *Geschäftsobjekte* bezeichnet – können dann über Ereignisregeln ausgelesen und aktualisiert werden. Mittels des vorgestellten Ansatzes wird das beschriebene Regelverwaltungssystem auch in Entitäten-lastigen Geschäftsumgebungen anwendbar. Solche Szenarien sind mit ausschließlich ereignisbasierten Strategien kaum handhabbar.

Hierarchische Muster-Modellierung ermöglicht die Wiederverwendung von Mustererkennungslogik auf der Ebene von individuellen Beziehungsmustern. Eine solche Wiederverwendbarkeit ist in SARI, ebenso wie in zahlreichen anderen CEP-Frameworks, nicht oder nur in eingeschränktem Maße gegeben, so dass Ereignismuster stets von Grund auf und in potentiell redundanter Form erstellt werden müssen. Der vorgestellte Ansatz basiert auf neuartigen Sprachelementen, welche als Referenzen auf andere, in weiterer Folge logisch untergeordnete Beziehungsmuster interpretiert werden. Diese Sprachelemente können in gewohnten Abläufen mit bestehenden Sprachelementen integriert und hinsichtlich des gegebenen Anwendungsfalles konfiguriert werden. Spezialisierte Auswertungsstrategien ermöglichen performante Ereignisverarbeitung sowie beliebige Verschachtelung von Mustererkennungslogik.

Die vorgestellten Konzepte sind in eine umfangreiche, modellorientierte Beschreibung von SARI eingebunden. Diese Beschreibung unterteilt SARI-Anwendungen in kleinere, einfacher zu verstehende Sub-Modelle und soll eine Basis für zukünftige Erweiterungen darstellen. Weiters soll die Beschreibung Austausch und Zusammenarbeit innerhalb der Fachwelt erleichtern.

Die vorgestellten Konzepte werden abschließend hinsichtlich ihrer technischen Machbarkeit, Anwendbarkeit und ihrer Nützlichkeit in praktischen Anwendungsfällen evaluiert. Technische Machbarkeit wird anhand der vollständigen Implementierung der Konzepte dargelegt. Anwendbarkeit wird auf Basis einer Beispielsanwendung für ereignisbasierte *Service Assurance* untersucht; hierbei wird SARI als Erweiterung zu einer weitverbreiteten Job-Scheduling- und Workload-Automation-Plattform eingesetzt. Die Nützlichkeit des vorgestellten Regelverwaltungssystems in praktischen Anwendungsfällen wird in einer Fallstudie, durchgeführt bei einem weltweit führenden Produzenten landwirtschaftlicher Gerätschaft, untersucht.



---

## Contents

|          |  |    |
|----------|--|----|
| <b>1</b> | <b>Introduction</b> .....                              | 21 |
| 1.1      | Motivation .....                                       | 21 |
| 1.2      | Event-Pattern Rules: Applications and Challenges ..... | 24 |
| 1.3      | Scope of this Thesis .....                             | 26 |
| 1.4      | Research Method .....                                  | 29 |
| 1.5      | Contributions .....                                    | 29 |
| 1.6      | Evaluation .....                                       | 33 |
| 1.7      | Structure of this Thesis .....                         | 34 |
| <br>     |  |    |
| <b>2</b> | <b>Principles of Complex Event Processing</b> .....    | 37 |
| 2.1      | Events .....   | 37 |
| 2.2      | Event Types .....                                      | 38 |
| 2.3      | Event Patterns .....                                   | 39 |
| 2.4      | Event-Pattern Rules .....                              | 39 |
| 2.5      | Complex Events .....                                   | 40 |
| 2.6      | Preliminary Example .....                              | 40 |
| <br>     |  |    |
| <b>3</b> | <b>Sense-and-Respond Infrastructure</b> .....          | 43 |
| 3.1      | Introduction .....                                     | 43 |
| 3.1.1    | Key Characteristics .....                              | 44 |
| 3.1.2    | Sense-and-Respond Loop .....                           | 44 |
| 3.1.3    | SARI Applications in a Nutshell .....                  | 45 |
| 3.2      | A Model-Based View on SARI Applications .....          | 48 |
| 3.3      | Event Model .....                                      | 51 |
| 3.3.1    | Meta Model .....                                       | 51 |
| 3.3.2    | Example .....  | 52 |
| 3.4      | Correlation Model .....                                | 53 |
| 3.4.1    | Meta Model .....                                       | 53 |
| 3.4.2    | Correlation Band Implementations .....                 | 54 |
| 3.4.3    | Example .....  | 55 |
| 3.4.4    | Correlating Events at Run Time .....                   | 56 |

|          |  |           |
|----------|--|-----------|
| 3.5      | Event Processing Model .....                               | 57        |
| 3.5.1    | Meta Model .....   | 57        |
| 3.5.2    | Event Adapters .....                                       | 59        |
| 3.5.3    | Event Services .....                                       | 59        |
| 3.5.4    | Intermediate Sockets .....                                 | 60        |
| 3.5.5    | Example .....  | 60        |
| 3.6      | System Architecture .....                                  | 61        |
| 3.6.1    | Data Layer .....   | 61        |
| 3.6.2    | Back-End Layer .....                                       | 63        |
| 3.6.3    | Front-End Layer .....                                      | 64        |
| 3.7      | Event Access Expressions .....                             | 64        |
| 3.7.1    | Basic Operators .....                                      | 65        |
| 3.7.2    | Handling Collection Types .....                            | 65        |
| 3.7.3    | Accessing Single Events .....                              | 65        |
| 3.7.4    | Accessing Sequences of Events .....                        | 66        |
| 3.7.5    | Functions .....  | 67        |
| <b>4</b> | <b>Decision Graphs .....</b>                               | <b>69</b> |
| 4.1      | Introduction .....   | 69        |
| 4.1.1    | Key Characteristics .....                                  | 71        |
| 4.1.2    | Decision Graphs and Rules .....                            | 72        |
| 4.1.3    | Outlook .....  | 72        |
| 4.2      | Meta Model .....   | 72        |
| 4.3      | Rule Components .....                                      | 74        |
| 4.3.1    | Common Characteristics .....                               | 74        |
| 4.3.2    | Condition Components .....                                 | 75        |
| 4.3.3    | Time-Based Components .....                                | 77        |
| 4.3.4    | Action Components .....                                    | 79        |
| 4.4      | Example .....  | 80        |
| 4.5      | Managing Decision Graph State .....                        | 81        |
| 4.5.1    | Merging Decision-Graph States .....                        | 82        |
| 4.5.2    | Stateless Decision Graphs .....                            | 82        |
| <b>5</b> | <b>A Framework for User-Oriented Rule Management .....</b> | <b>85</b> |
| 5.1      | Introduction .....   | 85        |
| 5.2      | Conceptual Foundations .....                               | 88        |
| 5.3      | Related Work .....   | 91        |
| 5.3.1    | Rule Management for Event-Based Systems .....              | 91        |
| 5.3.2    | Complex Event Processing for Business Users .....          | 92        |
| 5.3.3    | Layered Event-Processing Models .....                      | 94        |
| 5.3.4    | Differentiating Rules in Event Processing .....            | 95        |
| 5.3.5    | Business Rule Management Systems .....                     | 96        |
| 5.4      | Infrastructural Rule Management .....                      | 98        |
| 5.4.1    | Requirements .....   | 98        |
| 5.4.2    | Model Overview .....                                       | 99        |

|          |  |            |
|----------|--|------------|
| 5.4.3    | Rule Definitions . . . . .                                     | 100        |
| 5.5      | Sense-and-Respond Rule Management . . . . .                    | 102        |
| 5.5.1    | Requirements . . . . .   | 103        |
| 5.5.2    | Model Overview . . . . .                                       | 104        |
| 5.5.3    | Pattern Definitions . . . . .                                  | 110        |
| 5.5.4    | Action Definitions . . . . .                                   | 113        |
| 5.5.5    | Business Patterns . . . . .                                    | 115        |
| 5.5.6    | Business Actions . . . . .                                     | 118        |
| 5.5.7    | Sense-and-Respond Rules . . . . .                              | 119        |
| 5.5.8    | Rule Spaces . . . . .  | 122        |
| 5.6      | User Rights Management . . . . .                               | 125        |
| 5.7      | Implementation Architecture . . . . .                          | 127        |
| 5.7.1    | Data Layer . . . . .   | 128        |
| 5.7.2    | Back-End Layer . . . . .                                       | 129        |
| 5.7.3    | Front-End Layer . . . . .                                      | 131        |
| 5.8      | Modeling Studio . . . . .                                      | 131        |
| 5.8.1    | Pattern and Rule Definition Editor . . . . .                   | 132        |
| 5.8.2    | Action Definition Editor . . . . .                             | 132        |
| 5.8.3    | Rule Space Editor . . . . .                                    | 132        |
| 5.9      | Web Client . . . . .   | 137        |
| 5.9.1    | Integration . . . . .  | 137        |
| 5.9.2    | Interface Overview . . . . .                                   | 138        |
| 5.9.3    | Rule Monitoring . . . . .                                      | 139        |
| 5.9.4    | Rule Creation from Scratch . . . . .                           | 139        |
| 5.9.5    | Rule Creation from Template . . . . .                          | 145        |
| 5.9.6    | Handling Inconsistent Rules . . . . .                          | 146        |
| <b>6</b> | <b>Entity-Based State Management . . . . .</b>                 | <b>149</b> |
| 6.1      | Introduction . . . . .   | 149        |
| 6.1.1    | State Management in Complex Event Processing . . . . .         | 150        |
| 6.1.2    | Business Entity Providers: An Architectural Overview . . . . . | 151        |
| 6.1.3    | Business Entities in SARI Rule Management . . . . .            | 152        |
| 6.1.4    | SARI Application Model – Revisited . . . . .                   | 155        |
| 6.1.5    | Outlook . . . . .  | 155        |
| 6.2      | Related Work . . . . .   | 156        |
| 6.3      | Meta Model . . . . .   | 158        |
| 6.4      | Exemplary Business Entity Providers . . . . .                  | 160        |
| 6.4.1    | Scores . . . . .   | 161        |
| 6.4.2    | Base Entities . . . . .  | 162        |
| 6.4.3    | Sets . . . . .   | 165        |
| 6.5      | Correlation Model Extensions . . . . .                         | 167        |
| 6.5.1    | Meta Model . . . . .   | 167        |
| 6.5.2    | Example . . . . .  | 168        |
| 6.6      | Decision Graph Model Extensions . . . . .                      | 169        |
| 6.6.1    | Business Entity Actions . . . . .                              | 169        |



|          |   |            |
|----------|---|------------|
| 6.6.2    | Business Entity Conditions                                    | 171        |
| 6.6.3    | Handling Internal Business Entities                           | 174        |
| 6.7      | Implementation  | 175        |
| 6.7.1    | Data Layer  | 176        |
| 6.7.2    | Back-End Layer  | 177        |
| 6.7.3    | Front-End Layer   | 178        |
| 6.7.4    | Business-Entity Management at Run Time                        | 178        |
| 6.7.5    | Reference Business-Entity Providers                           | 179        |
| 6.8      | Example   | 181        |
| 6.9      | Summary   | 183        |
| <b>7</b> | <b>Hierarchical Pattern Modeling</b>                          | <b>185</b> |
| 7.1      | Introduction  | 185        |
| 7.2      | Related Work  | 190        |
| 7.3      | Decision Graph Model Extensions                               | 192        |
| 7.3.1    | Pattern Definitions   | 192        |
| 7.3.2    | Sub-Pattern Component   | 194        |
| 7.4      | Evaluation  | 197        |
| 7.4.1    | Evaluation by Expansion                                       | 198        |
| 7.4.2    | Hierarchical Evaluation                                       | 201        |
| 7.4.3    | Discussion and Comparison                                     | 202        |
| 7.5      | Example   | 204        |
| <b>8</b> | <b>Example</b>  | <b>209</b> |
| 8.1      | Introduction  | 209        |
| 8.2      | System Overview   | 210        |
| 8.3      | Event Model   | 211        |
| 8.4      | Business Entity Model   | 212        |
| 8.5      | Correlation Model   | 213        |
| 8.6      | Event Processing Model  | 213        |
| 8.7      | Infrastructural Rules   | 215        |
| 8.8      | Sense-and-Respond Rules                                       | 217        |
| 8.8.1    | Rule Spaces Overview  | 217        |
| 8.8.2    | Runtime Monitoring: Pattern Definitions, Business<br>Patterns | 218        |
| 8.8.3    | Runtime Monitoring: Action Definitions, Business<br>Actions   | 221        |
| 8.9      | Discussion  | 221        |
| <b>9</b> | <b>Case Study</b>   | <b>223</b> |
| 9.1      | Introduction  | 223        |
| 9.2      | Project Environment   | 224        |
| 9.3      | Problem   | 225        |
| 9.4      | Project Structure   | 225        |
| 9.5      | Application Overview  | 226        |

|           |   |            |
|-----------|---|------------|
|           | Contents                                | 19         |
| 9.6       | Discussion .....                        | 228        |
| <b>10</b> | <b>Conclusion</b> .....                 | <b>233</b> |
| 10.1      | Summary .....                           | 233        |
| 10.2      | Evaluation Against Design Science ..... | 237        |
| 10.3      | Open Issues and Future Work .....       | 241        |
|           | <b>References</b> .....                 | <b>253</b> |



## Introduction

### 1.1 Motivation

*There are events which are so great that if a writer has participated in them his obligation is to write truly rather than assume the presumption of altering them with invention.*

When Hemingway [52] wrote this, he was talking about the Spanish civil war, and he could have said similar things about a graceful *corrida de torros*, a devastating knockout, a long but fruitless safari, or a hazy sunrise on the Cuban sea. Albeit honorable and truthful, thoughts like this may not always be necessary when it comes to the more “everyday” events a person or company is faced with day by day: The ringing of an alarm clock, the late arrival of a subway train, the receipt of a customer request, or the cancellation of a cargo flight. Still, the appropriate and timely reception of such occurrences and their possible incorporation into future actions and decisions is crucial for the immediate and long-term success of an individual or corporation. “Event processing” in this very common sense is fundamental to every intelligent being, and as soon as individual events need to be combined based on complex interrelationships, it requires skills that go far beyond primitive stimulus-response patterns.

In computer science, event-based computing is nothing new, but “has been going on for more than fifty years” [73]. Over decades, event-based techniques were developed independently “in different areas to address similar challenges introduced by scale, system evolution, and real-time requirements” [83]. In discrete event simulation, for example, the behavior of a complex system such as a hardware design, a factory production line, or a natural phenomenon like weather is modeled by generating events that mimic the interactions between

---

<sup>0</sup> This thesis is formatted based on a L<sup>A</sup>T<sub>E</sub>X template provided by Gockel [45].

the components of the system.<sup>1</sup> Active databases enable application developers to automatically invoke operations in response to database update events such as inserts. In software engineering, event-based interactions can be found in software patterns such as the observer pattern [55] and are prominently employed in graphical user interface systems [83].

With the invention of message-oriented middlewares, the rise of the Internet, and the rapid uptake of e-commerce, event-based communication had eventually become the basis for “running applications everywhere – in business, government, and in the military” [74]:

“Any large enterprise had linked its applications across the networks from office to office, sometimes around the globe. It now operated on top of what was referred to as ‘the IT layer’. Business and management level events – say trading orders, or planning schedules or just plain email – were entering its IT layer from all corners of the globe, from external sources as well as from its own internal offices. [...] Enterprises were essentially operating in a veritable cloud of application level events.” [74]

In this new and challenging context, considerable efforts were made to establish the technical infrastructures and operational workflows that enable enterprises to (i) transfer events in a fast and reliable manner, and (ii) react timely, and often in a fully automated manner, to the occurrence of individual events such as the receipt of an order. Although counterexamples are available until today, these problems could be solved well by many companies. But what about the high-level trends, processes, and activities that are *not* signified by separate data items? That can only be recognized as a combination of dozens, if not hundreds or thousands, of structurally and temporally disconnected events? That, in some cases, are intended to remain undiscovered, e.g., when fraud is committed? What about “connecting the dots” [27]? High-level information that one might want to extract from a company’s event cloud may include:

- Are all customers served according to agreed service levels? Does a particular customer qualify for promotions or premium conditions, e.g., due to loyal buying behavior?
- Is the system at risk? Are our services abused for criminal purposes, e.g., money laundry?
- Are there new market opportunities, e.g., due to price adjustments of multiple competitors?

For a long time, companies relied on technologies such as data warehousing, data mining, or visual analysis to answer questions like these. All these ap-

---

<sup>1</sup> Discrete event simulation, network development, active databases, and middleware solutions have been identified as the “primeval soup” of Complex Event Processing by Luckham [73]. Other uses of events are discussed in the literature (e.g., [83, 37]).

proaches can provide great insights into the state of a company. However, they depend on discrete snapshots of a system (e.g., created periodically using an Extraction/Transformation/Load process) and are typically not optimized for the characteristics of event data.

Complex Event Processing (CEP) as originally proposed by David Luckham [71] emerged as a discipline around the year 2000 to fill this gap in enterprise computing. CEP is no longer limited to the transport of events, or the handling of individual ones, although these steps occur in almost any CEP application. Its main purpose is to put events *in context*, and to deduce knowledge that is otherwise inaccessible when viewing events one by one. The key to such knowledge are *complex events*, i.e., events that signify occurrences at a higher level of abstraction and are derived from sets of lower-level events through specialized algorithms. In the online-gambling domain, for example, a particular sequence of pay-in, bet placement, and cash-out actions could be combined into a higher-level “money laundry” event based on known fraud patterns. Still, detecting and creating such events is not possible without complementary processing steps such as the enrichment, filtering, or transformation of (simple as well as complex) events. In contrast to data warehousing, data mining, or visual analysis, CEP processes events “on the fly” and in a fully automated manner. It therewith facilitates what Stephan Haeckel [49] called a *sense-and-respond organization*: A company that is “aggressively open”, actively probes for environmental and internal signals, identifies business challenges as they happen, and responds to them with minimal latency.

Today, about ten years after its introduction, Complex Event Processing has made its way into the mainstream of enterprise computing. Numerous projects have been initiated, in the industry (cf. [48, 135]) as well as in academia (e.g., [1, 2, 4, 31, 117, 140, 141]) and the Open Source community (e.g., [36, 93]). The CEP market, long dominated by startups and university spin-offs, is estimated at US\$ 580 million by 2013 [19] and fiercely contested by some of the industry’s most prominent players. At the time of writing, this includes Microsoft [78], Oracle [94], IBM [56], and SAP [84]. According to studies, “event processing is the fastest growing segment of enterprise middleware software” [37].

As a result of these developments, more and more people are involved in the design, creation, and maintenance of event-driven applications. At the same time, vendors of CEP technology have not been able to provide technologies, tools, and workflows that are oriented towards larger, more heterogeneous user groups. Still, notable skills and high-end development environments are required also for tasks that should be in the responsibility of domain experts and business users. This is especially fatal when it comes to the creation, deployment, and administration of *event-pattern rules*, which find use in almost every event-processing framework and have been called “the foundation for successful applications of CEP” [71].

In this thesis, we present a novel approach to rule management for Complex Event Processing applications. It caters to the needs of IT experts as well as business users through tailored workflows, artifacts, and tools. Our approach has been fully implemented as part of the general-purpose event-processing framework Sense-and-Respond Infrastructure (SARI) [114] and successfully applied in use cases from different business domains.

## Outlook

The remainder of this chapter is structured as follows: In Section 1.2, we discuss the extraordinary role of event-pattern rules in CEP and the challenges that arise from their management in practical business environments. In Section 1.3, we define the scope of this thesis. The applied research method is discussed in Section 1.4. In Section 1.5 and Section 1.6, contributions and evaluations of our research are summarized. The structure of this thesis is outlined in Section 1.7.

## 1.2 Event-Pattern Rules: Applications and Challenges

The ultimate aim of CEP is to detect noteworthy business situations in large sets of events and react to their occurrences in near real time. With the evolution of event processing from hardcoded conditions deep within purpose-specific applications towards stand-alone, general-purpose event-processing frameworks, event-pattern rules of the form

**if** an event pattern  $p$  is detected, **then** execute action(s)  $A$

have proved particularly suitable for this purpose. Within an event-processing application, event-pattern rules may be evaluated on streams of input events and produce new events in response to the presence or absence of a specified class of event situations. In further consequence, these response events may cause respective actions in the business environment.

Practical CEP is, however, much more than “just” high-level sense-and-respond business logic. CEP instead reveals much of its power when it comes to the integration of multiple and highly heterogeneous event producers and event consumers, the preprocessing of raw, per se disconnected event data at a low level of abstraction, and the removal of erroneous, duplicate, or irrelevant input. Sense-and-respond business logic may thus be considered the top of a pyramid, resting upon considerable amounts of what we call “processing logic” to be applied prior to and after the high-level decision making. Processing tasks typically performed within an event-based application include:

- **Aggregations** describe the generation of more abstract, higher-level complex events from collections of lower-level occurrences, making the signified activities better understandable to humans and easier to handle in downstream event-processing logic. A typical event aggregation may, for instance, aggregate related pairs of “Account Opened” and “Account Closed” events that occur within a specified time frame to “Short-Term Account Creation” events.
- **Filtering** allows reducing the overall set of events to those that are relevant for the given processing task.
- **Transformations** take a single event as input and produce a single event as output, whereas the output event is derived from the input event.

Just like high-level situation detection, all these processing tasks (i) depend on the detection of a specific event pattern in order to be executed – single events in case of transformations and simple filters, event situations in case of complex filters and aggregations – and (ii) generate output events in response to a matched pattern. In practical CEP applications, event-pattern rules are therefore used for both: high-level, sense-and-respond business logic on the one hand, low-level event aggregations, filtering, and transformations on the other. In the majority of frameworks, one and the same rule evaluation technique, one and the same rule description language, and one and the same rule management approach are used for these purposes.<sup>2</sup>

Event-pattern rules still differ, however, regarding their particular function in an application, show different characteristics, and will typically be associated with different user groups within an enterprise.

Business logic, on the one hand, is likely to change over time and to evolve along with the objectives of a business, e.g., when new risks and opportunities are identified, or new markets and customer groups emerge. In the online-gambling domain, for example, “the ever-evolving cat-and-mouse game between players and merchants” [29] forces betting providers to continuously widen their fraud-detection strategies and tweak existing patterns of fraudulent user behavior. Business logic will typically be defined by domain users with a deep knowledge of the business environment and a clear understanding of the specific actions needed to keep business going day to day. However, these users will typically lack the technical skills that are necessary to define complex event-processing logic in a way that can be interpreted by the CEP engine at hand.

---

<sup>2</sup> Using one and the same technique for expressing business logic and processing logic becomes especially obvious when comparing situation detection and aggregation. In the words of Luckham [71], “constraints [rules that are concerned with situation detection, H.O.] can be regarded a special kind of map [rules that are concerned with aggregation, H.O.]. However, the purpose of a constraint is not filtering or aggregation, but detection. This is a rather subtle distinction, but it is an important one.”



Processing logic, on the other hand, typically remains relatively stable over time and is seldom affected by changes in the high-level decision making; for example, when monitoring a stream of betting events for a newly-discovered fraud technique there is usually no need to change the way in which erroneous events are rectified or filtered. Processing logic requires, however, a deep knowledge about the various event producers, event consumers and their interfaces, the interaction patterns employed to sense and respond events, and the ways in which event data shall be processed and routed through the application. As a consequence, processing logic will typically be administrated by IT experts with a deep understanding of the employed CEP framework. In practice, such users will often lack the skills and competences for high-level, operational decision making.

To fully unleash the power of CEP across all parts of an organization, an approach to rule management for Complex Event Processing systems – i.e., the overall set of tools and workflows provided for the creation, application, and administration of event-pattern rules – must therefore cater to the needs of IT experts as well as business users. Business users must be provided facilities to define high-level decision-making logic in a non-technical, business-oriented manner. IT experts, by contrast, must be provided facilities to define processing logic such as event aggregations, filters, and transformations in an effective manner with little administrative overhead. Following from these highly divergent and partly opposing requirements, approaches that focus on only one group of users will inevitably fail to support the work of the other. This, in turn, leads to increased costs and reduced agility, an error-prone IT infrastructure, and discontent among employees.

### 1.3 Scope of this Thesis

For many years, discussions on CEP have primarily focused on purely technical qualities such as expressiveness and performance (cf. [54]). These issues are doubtlessly essential for CEP to qualify for large-scale environments, and also serve as an argument “pro CEP” compared with well-established approaches such as Business Rule Management (BRM) and Business Process Management (BPM) [14]. We claim, however, that usability and maintenance are equally important for the success of CEP in industrial settings where large, heterogeneous user groups are involved in the setup and maintenance of event-driven applications.

This thesis addresses the problem of designing, implementing, and evaluating a rule management system for Complex Event Processing applications that caters to the needs of IT experts as well as business users. Rule management, which we understand as

the overall set of tools and workflows provided for the creation, deployment, and administration of event-pattern rules,

has been identified as a key issue for the future development of CEP both in the academic community [27, 72] and in the industry [33].

In the course of our research, the following concepts have been elaborated:

- **Rule management framework.** The centerpiece of our work is an innovative framework for user-oriented rule management. Starting from a conceptual differentiation of event-pattern rules into *infrastructural rules* (processing logic) and *sense-and-respond rules* (business logic), we present a suite of user roles, workflows, tools, and artifacts that is tailored to the particular skills, responsibilities, and requirements of IT experts as well as business users.
- **Entity-based state management.** As a first extension to this framework, we present a novel approach to state management for CEP applications. It is based on the idea of durable, application-wide data structures – so-called *business entities* – which can be updated and monitored for exceptional situations through event-pattern rules. Using entity-based state management, the presented rule-management framework becomes applicable also in data-centric business environments that are difficult, if not impossible, to approach with purely event-based strategies.
- **Hierarchical pattern modeling.** As a second extension to the proposed framework, we investigate a novel approach to hierarchical pattern modeling. It enables reuse of event patterns within other, higher-level event patterns and aims to reduce the overall degree of complexity and redundancy of an application’s event pattern library. Within the proposed rule-management framework, hierarchical pattern modeling aids the work of IT experts and technically versed domain experts.

The presented concepts are designed and implemented on the basis of Sense-and-Respond Infrastructure (SARI) as originally proposed by Schiefer and Seufert [114]. SARI is a general-purpose CEP framework that has proved successful in a wide range of business domains, ranging from fraud detection and prevention [109, 125] to logistics [112], medical care [133], and automated product recommendation [67]. Commercial distributions of SARI are available under the name UC4 Decision [132] (formerly: Senactive InTime) and have been awarded “innovative, impactful, and intriguing” in Gartner’s Cool Vendor Report 2008 [43]. The conceptual foundations of SARI, its architecture and key concepts are discussed in great detail in Chapter 3 of this thesis.

Figure 1.1 sketches the scope of the proposed concepts within SARI.

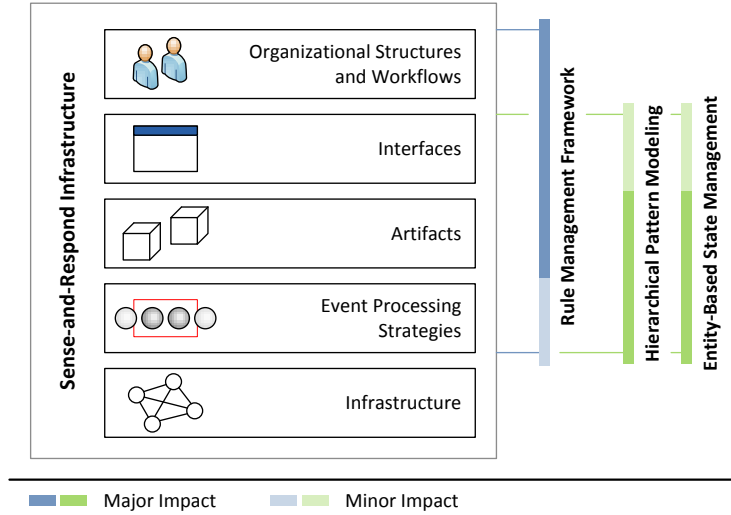


Figure 1.1. Scope of this Thesis

On the left-hand side, SARI is divided into five layers ranging from the organizational settings in which it is used to the low-level infrastructure of the framework. The *organizational structures and workflows* layer refers to the distribution of roles and responsibilities within a company and the strategies applied in dealing with the system. The *interfaces* layer refers to the suite of front-end tools provided for setting up and maintaining a SARI application. The *artifacts* layer refers to the logical elements of a SARI application to be created and administrated through these interfaces. This may, for example, include event types, event patterns, or event-pattern rules. The *event processing strategies* layer refers to the methods and techniques employed for the execution of user-defined event processing logic. Most notably, this includes the evaluation of event-pattern rules on incoming event streams. The *infrastructure* layer refers to low-level activities such as the balancing of event processing workload among distributed machines, synchronization between these machines, event data persistence, or fail-over mechanisms.

On the right-hand side, the proposed concepts are shown together with their impact on existing SARI functionality and usage patterns. The core rule-management framework imposes a set of user roles and workflows, for which tailored interfaces, rule-management artifacts, and event-processing strategies are provided. The proposed approaches to entity-based state management and hierarchical pattern modeling are generally independent from the organizational settings of a company. However, they come with new artifacts and require significant changes in SARI's event processing algorithms. Minor changes are also needed in SARI's front-end tools. Our research does not touch on the infrastructural layer of SARI.

## 1.4 Research Method

Research on Complex Event Processing is often conducted at the intersection of computer science and the *Information Systems* (IS) discipline (e.g., [66]), who’s goal is to develop “knowledge concerning both the management of information technology and the use of information technology for managerial and organizational purposes” [143]. This is especially true if the research focus is on usability and manageability within the organizational framework conditions of an enterprise, as it is the case in the present thesis. Within the information systems discipline, two complementary paradigms exist [13]: The *behavioral-science paradigm*, on the one hand, seeks to empirically investigate the use and impact of existing information systems on individuals, groups, or organizations, and to develop theories that explain or predict human behavior. The *design-science paradigm*, on the other hand, is rooted in engineering [121] and seeks to create and evaluate innovative IT artifacts that solve organizational problems. Such IT artifacts can be separated into *constructs* (i.e., vocabulary and symbols), *models* (i.e., abstractions and representations), *methods* (i.e., algorithms and practices), and *instantiations* (i.e., implemented and prototype systems) [53].

In their widely cited work on “Design Science in Information Systems Research”, Hevner et al. [53] present a framework of guidelines and criteria for the successful conduction of design-science research (Table 1.1). The work presented in this thesis is oriented towards Hevner’s framework and focuses on the design and implementation of an innovative rule management framework for Complex Event Processing applications; while technical feasibility, applicability, and utility of the proposed concepts are thoroughly evaluated, behavioral user studies are outside the scope of this thesis. In the following sections, the core guidelines of Hevner’s framework – namely, (i) providing valuable contributions in the area of the developed design artifacts, and (ii) the rigorous evaluation of these artifacts – are discussed in the context of our work.

An evaluation against the complete set of criteria is given in Chapter 10, “Conclusion”.

## 1.5 Contributions

Following Hevner et al. [53], design-science research “must provide clear contributions in the areas of the design artifact, design construction knowledge [...] and/or design evaluation knowledge [...]” This thesis contributes to the field of Complex Event Processing, for which a novel approach to user-oriented rule management is presented. Further contributions are a model-driven reference description of SARI, a novel approach to entity-based state management, and a novel approach to hierarchical pattern modeling.

|   | Guideline                         | Description  |
|---|-----------------------------------|--|
| 1 | <b>Design as an Artifact</b>      | Design science requires the creation of a purposeful IT artifact.  |
| 2 | <b>Problem Relevance</b>          | Design science must be relevant with respect to an important business problem.   |
| 3 | <b>Design Evaluation</b>          | Design science must evaluate the utility, quality, and efficacy of a design artifact using well-executed evaluation methods.                         |
| 4 | <b>Research Contributions</b>     | Design science must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. |
| 5 | <b>Research Rigor</b>             | Design science relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.                       |
| 6 | <b>Design as a Search Process</b> | Design science must apply a search process to reach desired ends while satisfying laws in the problem environment.                                   |
| 7 | <b>Communication of Research</b>  | Design science must be presented effectively both to technology-oriented as well as management-oriented audiences.                                   |

**Table 1.1.** Guidelines for Design-Science Research [53]

## A Model-Driven View on Sense-and-Respond Infrastructure

**Problem.** Sense-and-Respond Infrastructure (SARI) [114] is both the result and the foundation of considerable research efforts in the fields Complex Event Processing, Information Visualization, Data Mining, and Data Management (e.g, [92, 104, 105, 107, 108, 109, 111, 112, 113, 114, 123, 125, 133]). Commercial distributions of SARI have been successfully applied in industrial settings ranging from finance to manufacturing and e-commerce, and are well-recognized in leading analyst reports [43, 48, 135]. Still, SARI lacks a complete and technically oriented reference description that may serve as a basis for future research and facilitate communication, interchange, and cooperations within the event processing community.

**Contribution.** This thesis introduces a model-driven view on SARI. In this view, the overall complexity of a SARI application is decomposed into a collection of smaller, easier-to-understand sub-models, each of which is in defined relationships to other sub-models. Therewith, this thesis aims to contribute to ongoing efforts towards clarifying terminologies and classifying event-processing strategies among the members of the event processing community, especially driven by the Event Processing Technical Society (EPTS).<sup>3</sup>

<sup>3</sup> <http://www.ep-ts.com>

## A Framework for User-Oriented Rule Management

**Problem.** Event-pattern rules of the form “if an event pattern  $p$  is detected, then execute action(s)  $A$ ” can be used on different conceptual layers of an event-based application, in low-level pre-processing and integration steps (such as aggregation, filtering, and transformation) as well as in high-level, *sense-and-respond*-based decision making. With the growing role of CEP in enterprise computing, it hence becomes increasingly important for companies that event-pattern rules are manageable to personnel that are best qualified to operate on these layers, using workflows and tools that are geared towards the particular skills, competences, and responsibilities of these user groups. To our best knowledge, existing CEP frameworks have not been able to satisfy this requirement.

**Contribution.** This thesis presents a novel approach to rule management for Complex Event Processing applications. It caters to the needs of IT experts as well as business users, for which complementary, yet clearly decoupled workflows are presented. These workflows are rooted in a conceptual differentiation of event-pattern rules by their general function within an event-based application: *Infrastructural rules*, on the one hand, prepare data for other parts of the application, but do not by themselves respond to the underlying business environment. *Sense-and-respond rules*, on the other hand, set up on the resulting, readily-preprocessed event-based image of the underlying source systems and act on noteworthy business situations by directly or indirectly triggering actions in the business environment.

In the proposed framework, IT experts define infrastructural rules in a single, comprehensive model, in parallel and fully integrated with the other elements of an application’s event-processing infrastructure. Sense-and-respond rule management, by contrast, builds upon a sophisticated system of configurable building blocks of pattern-detection and reaction logic. Prepared by technically versed domain experts according to the general requirements of an application scenario, these building blocks can be assembled to concrete event-pattern rules by technically inexperienced business users in a way that entirely abstracts from underlying complexity. Similar, the deployment of so-created sense-and-respond rules is performed in a fully automated and transparent manner based on predefined associations between building blocks and rule-execution agents. Despite significant differences in the creation and deployment of event-pattern rules, both infrastructural and sense-and-respond rule management set up on SARI’s unified rule-evaluation model, which facilitates a lean, easy-to-maintain back-end architecture.

The proposed framework has been successfully implemented within SARI. Applicability and utility in real-world use cases is demonstrated using an exemplary application scenario and a case study.

## Entity-Based State Management

**Problem.** Complex Event Processing using event-pattern rules has proved suitable for detecting noteworthy business situations of a defined length and structure, where the focus is on relationships between the involved events. By contrast, challenges arise when the overall state of a complex, durable business entity – e.g., a counter, a server, or a task queue – shall be derived from incremental, low-level updates of that state, and each update is represented by a (possibly complex) event. Depending on the given event-processing architecture, these challenges may include (i) the demand for a separate, non-volatile data management layer, (ii) the potential complexity of calculating overall entity state from low-level updates, (iii) the need for actively reacting on entity-level state changes, (iv) the integration of entity data with existing, otherwise event-specific concepts, and (v) general ease of use for end users.

**Contribution.** In this thesis, we present a novel approach to state management for Complex Event Processing applications. We propose *business entity providers*, which encapsulate arbitrary state-calculation logic and manage state in the form of typed, application-wide data structures. Using a plug-in-based component model, business entity providers can be integrated into a SARI application based on the specific requirements of a business scenario. We present extensions to SARI’s original event-processing capabilities that enable accessing business entities well-integrated with event-pattern detection, and demonstrate our approach in a real-world scenario from the workload automation domain. While business entity providers may be used with any application architecture that is suitable for the particular problem at hand, the proposed concept is designed to naturally complement the novel approach to rule management and make it applicable also in entity-centric business environments. Technical feasibility is demonstrated in an experimental extension of SARI and reference implementations for commonly needed business entity providers.

## Hierarchical Pattern Management

**Problem.** The novel rule-management framework of Sense-and-Respond Infrastructure inherently builds upon the reuse of event patterns on the level of business-user-defined *sense-and-respond rules*, which are assembled from building blocks of pattern-detection and reaction logic. By contrast, SARI’s original event-processing facilities do not support the reuse of pattern-detection logic on the level of event patterns: Each event pattern must instead be defined individually and from scratch, even though considerable commonalities may exist between event patterns. As a result thereof, event patterns may reach a high level of complexity, show redundancies, and become limited in expressiveness whenever configurations shall apply to conceptually separated, self-contained parts of an event pattern. In general, reusability on the level of event patterns

is not well supported in state-of-the-art CEP frameworks, especially when the focus is on a natural and consistent workflow for end users of a system.

**Contribution.** This thesis presents a novel approach to hierarchical pattern modeling for SARI. Our approach allows incorporating sub-level pattern-detection logic into super-level event patterns through novel event-pattern language elements – so-called *sub-pattern components* – which serve as a reference to the incorporated event pattern and can be integrated with the other elements of SARI’s event-pattern language. Through the use of typed input and output parameters, sub-level event patterns can be configured based on the specific context in which they are used, and underlying complexity is hidden from the user. We present tailored evaluation strategies that enable high-performance event processing as well as arbitrary nestings of pattern-detection logic and demonstrate the feasibility of our approach through a full-featured implementation.

## 1.6 Evaluation

In the course of our research, the proposed concepts have been evaluated for technical feasibility, applicability, and utility in industrial settings using the following methods:

### Implementation

Technical feasibility of the presented concepts is demonstrated through their successful implementation within SARI. Full implementations are available for the proposed approaches to *user-oriented rule management* and *hierarchical pattern modeling*, which could also be commercialized as part of UC4 Decision from version 9.00 onwards. The presented approach to entity-based state management was implemented as an experimental extension to SARI along with reference implementations for commonly required business entity providers. Commercial distributions of SARI currently ship with a simplified version of the proposed approach that is restricted to the use of *scores* and persistent data management (see Chapter 6 for further details). Implementation details are presented in the respective chapters.

### Exemplary Application Scenario

Applicability of the proposed concepts is evaluated using an exemplary SARI application for *event-based service assurance*, where SARI is applied as an



extension to the UC4 Automation Engine [130].<sup>4</sup> The presented application implements the proposed differentiation of event-pattern rules into *infrastructural rules* and *sense-and-respond rules* and makes extensive use of entity-based state management; hierarchical pattern modeling is demonstrated in a separate example to be found in the respective chapter. The application has been developed by members of the UC4 Senactive development team and has been successfully set up at customers from different business domains.

## Case Study

Utility in practical business environments is evaluated using a case study, where the setup and operation of the proposed concepts is analyzed in the context of a real-use situation. Through observation and qualitative interviews, it especially aims to investigate how the proposed distribution of user roles is put into practice and how presented rule-management concepts are adopted in the given application scenario. Our case study spans a period of seven days and is conducted at a leading manufacturer of agricultural machinery.

## 1.7 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 discusses the fundamental concepts of Complex Event Processing, namely events, event types, event patterns, event-pattern rules, and complex events. The presented concepts are demonstrated in a preliminary example from the logistics domain.

Chapter 3 provides a detailed introduction to Sense-and-Respond Infrastructure. It particularly focuses on SARI's application model, for which a novel, model-driven view is elaborated. Each sub-model of this view is discussed in terms of one or more abstract meta models and exemplary realizations of these models. Chapter 3 furthermore presents the implementation architecture of SARI and discusses the most relevant features of SARI's tailored language for accessing event streams, Event Access Expressions.

Chapter 4 completes the model-driven view on SARI applications by discussing the central *decision graph model*, which forms the basis for any rule-based event processing in SARI. We discuss the basic meta model for decision graphs, their evaluation during run time and give a detailed introduction to SARI's core *rule component library*. Rule components encapsulate easy-to-understand pieces of

---

<sup>4</sup> Service assurance is understood as the proactive monitoring of business environments with the goal of detecting fault patterns and ensuring reliability and performance in a system landscape.

event-processing logic (such as the occurrence of an event of type  $T$ , with certain event attribute values) and, together with precondition relationships between them, form the base elements of any decision graph.

Chapter 5 presents the proposed framework for user-oriented rule management. Following an introductory discussion on the conceptual foundations of our work – i.e., a differentiation of rules into *infrastructural rules* and *sense-and-respond rules*, a collection of user roles and appropriate workflows – we present in detail our approaches to infrastructural rule management and sense-and-respond rule management. We discuss user rights management for these approaches and illustrate the implementation of our framework as an extension to the base architecture of SARI. A particular focus is placed on extensions of the front-end layer of SARI, namely, an extended IDE for power users and a simplified web interface for business users.

Chapter 6 presents our approach to entity-based state management. We introduce the concept of *business entity providers*, which encapsulate arbitrary state-calculation logic and manage state in the form of typed, application-wide data structures. Based on an abstract meta model, reference implementations for three commonly required kinds of business entities – namely, *scores*, *base entities*, and *sets* – are discussed. We elaborate extensions to SARI’s original correlation and decision-graph model that enable accessing business entities well-integrated with event-pattern detection and discuss the implementation of our framework within SARI. Our approach is demonstrated using a real-world scenario from the workload automation domain.

Chapter 7 presents our approach to hierarchical pattern modeling. We extend SARI’s original decision graph model by the novel *sub-pattern component* and discuss tailored evaluation strategies that enable high-performance event processing as well as arbitrary nestings of pattern-detection logic. Applicability of our approach is demonstrated using an example from the fraud-detection domain.

Chapter 8 demonstrates the proposed framework using a real-world SARI application for *event-based service assurance*, which we understand as the proactive monitoring of business environment with the goal of detecting fault patterns and ensuring reliability and performance in a system landscape. The application is discussed following the model-driven view on SARI applications with a particular focus on the application’s infrastructural rules and sense-and-respond rule-management artifacts.

Chapter 9 presents the results of a case study conducted at a leading manufacturer of agricultural machinery.

Chapter 10 summarizes and concludes this thesis and gives an outlook to future research issues.



## Principles of Complex Event Processing

**Abstract** Complex Event Processing (CEP) provides methods and techniques to extract high-level information from large sets of events, identify noteworthy business situations, and take subsequent action in near real time. In this chapter, the fundamental principles of CEP are summarized: *Events* refer to “things that happen” as well as to the programming entities that represent such occurrences. *Event types* describe the structure of a class of events. *Event patterns* are abstract definitions of classes of (complex) event situations such as a delayed shipment process or an attempt to commit fraud. *Event-pattern rules* associate such patterns with one or more actions to be executed whenever a combination of events matches the pattern. We conclude this section by a preliminary example from the logistics domain.

### 2.1 Events

The concept of events is pertinent to almost every scientific discipline. In computer science, events are used in different areas such as active databases, distributed computing, and software engineering, and several definitions have been developed (e.g., [39, 40, 61, 75, 142]).

In the context of (complex) event processing, Etzion and Niblett [37] define events as follows:

“An *event* is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word *event* is also used to mean a programming entity that represents such an occurrence in a computing system.”

The above definition gives two meanings of the term *event*. The first meaning matches the very common understanding of events and is closely related to the *Oxford English Dictionary*’s definition of event as “something that happens or is thought of as happening” (cf. [71]). The second meaning refers to the data

item that *represents* such a occurrence within an event processing application. To avoid ambiguity, the term *event object* is sometimes used to refer to events as programming entities; in the remainder of this thesis, it is used only if the meaning could not otherwise be told from the context.

For events in the sense of programming entities, Luckham [71] identified the following *aspects*:

- **Form.** Events have a particular *form*, i.e., are instances of a certain kind of data structure. In most of today's CEP framework, events are represented as tuples of named data components, so-called *event attributes*. Such a tuple of event attributes may, for example, be implemented as an XML document or an object in an object-oriented programming language.
- **Significance.** An event always *signifies* an occurrence, i.e., an actual action or state change in the monitored source system. Typically, this so-called *significance* is further characterized via the event's event attributes; for example, an event indicating the start of a transport process could include the transport's start time (i.e., the event's time of occurrence), the carrier, the truck's ID, and the quality and quantity of transported goods.
- **Relativity.** An event is always related to other events, by *time* (i.e., two or more events occur within a specified time frame), *causality* (i.e., one event *causes* another event), or *membership* (i.e., two or more events belong to one and the same higher level event, e.g., to the same order process). Relationships are typically encoded in the event's event attributes, so that relationships can be reconstructed on structurally decoupled events; for example, two events of a particular transport process would typically include the same transport ID in their event attributes.

In the following, we refer as *raw events* to events that are retrieved directly from the source system and as *virtual events* to events that are generated within the event-processing application. Moreover, we refer as *output events* to events that are intended to directly or indirectly impact on the source system, e.g., that cause an email to be sent.

## 2.2 Event Types

Modern CEP frameworks are typically *typed*, i.e., the form of a class of events is defined through an *event type* to which all members of this class must conform. A detailed discussion of event type models in CEP is given by Rozsnyai et al. [108].

## 2.3 Event Patterns

According to Eckert and Bry [34], CEP use cases can be distinguished into (i) the detection of predefined combinations of events – so-called *event patterns* – within a company’s event cloud, and (ii) the identification of previously unknown patterns through machine learning or data mining algorithms. At the time of writing, research primarily focuses on the detection of predefined event patterns, and so does this thesis.

Following Etzion and Niblett [37],

“an *event pattern* is a template specifying one or more combinations of events. Given any collection of events, you may be able to find one or more subsets of those events that match a particular pattern. We say that such a subset *satisfies* the pattern.”

In general-purpose CEP frameworks, event patterns must be defined by the adopters of a framework according to the particular high-level trends, processes, and activities that they are interested in. Most CEP engines provide a tailored language for describing such a pattern; this language is typically referred to as Event Pattern Language (EPL). The efficient, near real-time evaluation of so-defined event patterns also for large amounts of events is a key requirement to any CEP framework.

## 2.4 Event-Pattern Rules

Only in the rarest of cases, event-pattern detection is performed as an end in itself – if a relevant pattern is detected, it will instead be demanded to automatically trigger an action (e.g., block a user that shows fraudulent behavior), notify a responsible user (e.g., send an email to an administrator), or consider this fact in further, higher-level pattern detection steps. In CEP, so-called *event-pattern rules* associate an event pattern – the so-called *trigger* – with a collection of actions to be taken whenever the event pattern is matched. Event-pattern rules are “the foundation for successful applications of CEP” [71], and their end-user-oriented creation, deployment, and administration within practical business environments is the main research objective of this thesis.<sup>1</sup>

---

<sup>1</sup> At the time of writing, alternative approaches to event processing are, in fact, still in their infancy (although for an integration of CEP with neural networks refer to Widder et al. [138]).

## 2.5 Complex Events

Eponymous to CEP, the term *complex event* generally refers to events that are aggregated from sets of lower-level events, their so-called members. These members may comprise multiple event types, occur at different points in time, and emerge from different sources. Complex events are typically generated using above-described *event-pattern rules*, then sometimes referred to as *aggregation rules*.

## 2.6 Preliminary Example

In the following, the above-presented core concepts of CEP shall be demonstrated using a simplified example from the logistics domain.<sup>2</sup> In the given scenario, all drivers of a logistics company are equipped with handheld devices on which both the sender and the receiver of a freight confirm the successful handover of the transported goods. Moreover, all trucks are equipped with thermometers that measure the inside temperature at regular time intervals. All these data are continuously transferred to the carrier’s head office, where they are transformed into event objects of respective event types – “Transport Started”, “Transport Ended”, and “Temperature Update” – and fed into a CEP application. Table 2.1 lists the event types of the presented scenario along with their visual representation used in Figure 2.1 below; all events carry the transport’s unique transport ID as an event attribute.

In the CEP application, an event pattern “Failed Transport” is evaluated on the incoming event stream to detect all transports where

*more than 5% of all measurements are above a threshold of +5 °F.*

In event-pattern rules, this pattern can be used to trigger counteractions in the underlying business environment, e.g., by sending an email to responsible personnel or automatically invoking an operation in the carrier’s IT system. Alternatively, an event-pattern rule could be applied to create a complex “Transport Failed” event; this event could then contribute to higher level event patterns, e.g., to detect recurring failures of a particular truck.

---

<sup>2</sup> The presented example is loosely based on an example originally given by Rozsnyai [105].




| Event Type  | Description   |
|---|---|
|  | <b>Transport Start</b> Signifies the start of a transport process.  |
|  | <b>Transport End</b> Signifies the end of a transport process.  |
|  | <b>Temperature Update</b> Signifies a measurement made by a truck's temperature sensor. Temperature updates are generated at regular time intervals and carry the current temperature as an event attribute. In Figure 2.1, temperature updates are depicted in green if the temperature is below +5 °F and in red otherwise. |

Table 2.1. Event Types

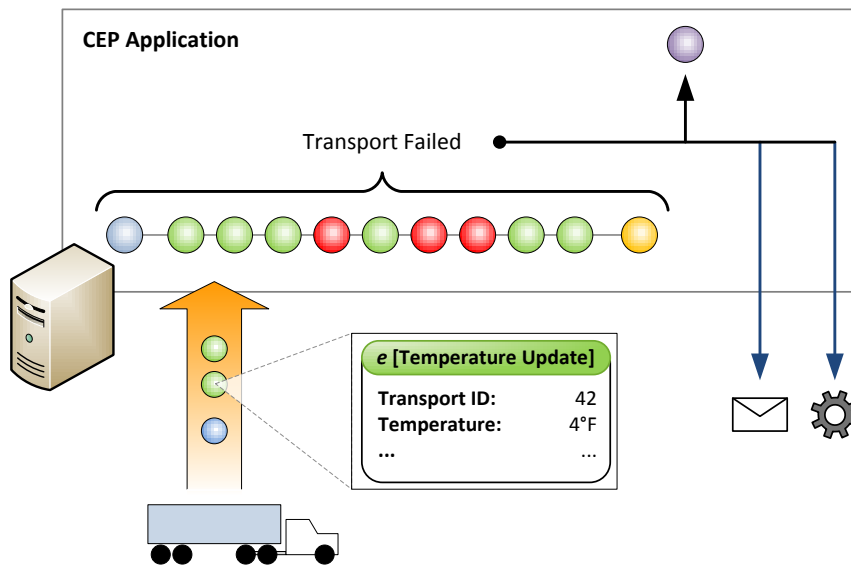


Figure 2.1. Preliminary Example (cf. [105])





## Sense-and-Respond Infrastructure

**Abstract** Since introduced to a wider community by David Luckham, Complex Event Processing has inspired projects of commercial as well as academic nature. This chapter presents Sense-and-Respond Infrastructure (SARI), a general-purpose event-processing framework which forms the basis for our research on user-oriented rule management. We particularly focus on SARI's application model, for which a novel, model-driven view is elaborated. In this view, the overall complexity of a SARI application is decomposed into a collection of smaller, easier-to-understand sub-models, each of which is in defined relationships to other sub-models. We present the implementation architecture of SARI and discuss the most relevant features of SARI's tailored language for accessing event streams, Event Access Expressions.<sup>1</sup>

### 3.1 Introduction

In recent years, Complex Event Processing (CEP) has emerged as a new paradigm for the real-time monitoring of business environments and automated, event-driven decision making. Since introduced to a wider community by David Luckham and his seminal work on “The Power of Events” [71], CEP has inspired numerous projects of academic as well as commercial nature, and also in the Open Source community.

In this chapter, we present Sense-and-Respond Infrastructure (SARI) as originally proposed by Schiefer and Seufert [114]. SARI is a generic CEP framework that has proved successful in a wide range of business domains ranging from fraud detection and prevention [109, 125] to logistics [112], medical care [133], and automated product recommendation [67]. SARI, along with its specific characteristics, features, and restrictions, serves as a basis for our research on user-oriented rule management as presented in Chapter 5, Chapter 6, and Chapter 7 of this thesis.

---

<sup>1</sup> This chapter is based on the work of Obweger et al. [89].

### 3.1.1 Key Characteristics

In the course of our research, we identified the following key characteristics of SARI in relation to other CEP frameworks:

**Applicability across business domains.** In contrast to solutions that are tailored to the requirements of a specific field such as finance, SARI was designed as a general-purpose event-processing framework to be used in arbitrary business environments. Use-case-specific logic is managed in so-called SARI applications, which can be defined by the end user or provided as part of a ready-to-use, customer-specific distribution. The different logical parts of a SARI applications are discussed in Section 3.3 to Section 3.5 of this chapter.

**Rule-based computing using Event/Condition/Action rules.** Rule-based event processing is a fundamental principle of CEP. In contrast to query-based (e.g., [36, 78, 127]) and inference-based (e.g., [59, 128]) strategies that have also been discussed in the literature, SARI employs an Event/Condition/Action (ECA) based approach to rule-based event processing. In an ECA rule, an event pattern – e.g., a particular sequence of cash-in, bet placement, and cash-out actions taken by a user of an online-gambling platform – is associated with one or more actions to be executed whenever the pattern is fulfilled. Decision graphs, which form the basis for rule-based event processing in SARI, are discussed in great detail in Chapter 4.

**Usability and manageability.** SARI was designed with a particular focus on usability and manageability from an end-user perspective. This reflects in a clear, well-structured application model and sophisticated front-end tools. Our research on user-oriented rule management further reinforces this emphasis by providing rule-management facilities tailored to the skills, responsibilities, and requirements of technically versed users as well as business users.

**Scalability and reliability.** Several event processing solutions have been designed to provide efficient centralized situation-detection mechanisms (e.g., [4, 25, 44] – cf. [115]). SARI allows distributing the event-processing workload of an application over a network of self-contained event-processing nodes, with failover strategies in case of crashes. The execution of SARI applications in a distributed environment is discussed in greater detail in Section 3.6.

### 3.1.2 Sense-and-Respond Loop

The main concept behind SARI's approach to event processing is the so-called Sense-and-Respond Loop [114] as shown in Figure 3.1. Especially emphasizing the idea of a continuous, self-adjusting integration between CEP and underlying business environments, Sense-and-Respond Loops divide event processing into a sequence of five stages, namely (i) Sense, (ii) Interpret, (iii) Analyze, (iv) Decide, and (v) Respond. Table 3.1 discusses these stages in greater detail.

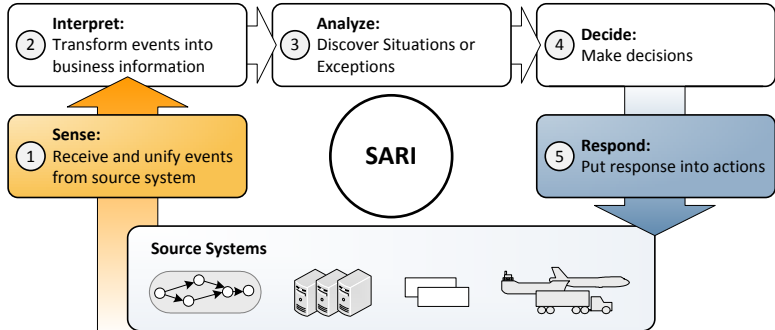


Figure 3.1. Sense-and-Respond Loop

3.1.3 SARI Applications in a Nutshell

SARI was designed as a generic event-processing framework that may set up on arbitrary source systems. Solutions to concrete business problems – for example, for detecting fraudulent user behavior in an online-gambling platform, or monitoring and controlling the transports of a logistics company – are managed as so-called SARI applications, which encapsulate the environment-specific integration approach and the problem-specific event-processing logic for one particular use case. As an introduction to the more detailed, primarily model-driven view on SARI applications as is presented throughout the following sections, Figure 3.2 provides a first overview of the most relevant parts of a SARI application.

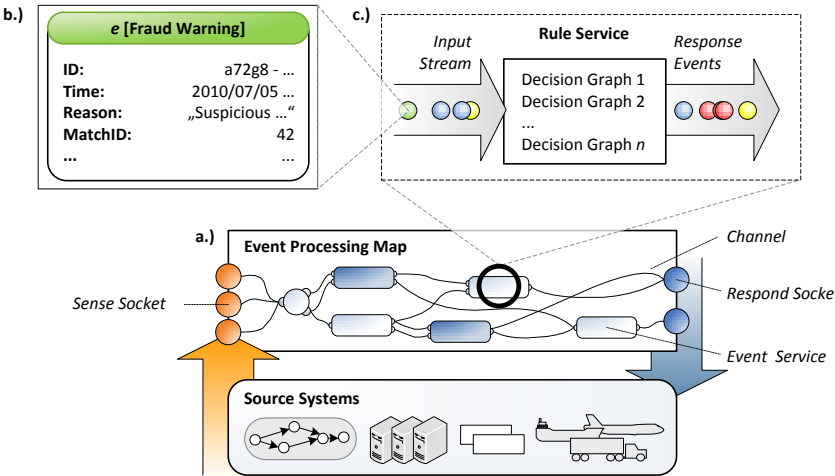


Figure 3.2. SARI Application Overview

**Event processing maps.** The central concept of a SARI application is the event processing map (Figure 3.2a.), a user-defined orchestration of *sense sockets*, *respond sockets*, and *event services*. Following the idea of a service-oriented approach to event processing, elements of an event processing map operate as independent, self-contained processing units that may receive input events from other elements of a processing map, and/or publish output events. Output events produced by a map element are routed to other map elements according to a set of *channels* between these elements. As shown for an exemplary “Fraud Warning” event in Figure 3.2b., events in SARI may further characterize an action or state change through a collection of *event attributes* as defined in their *event type*.

**Sockets.** Sense sockets and respond sockets implement the “entry points” (sense) and “exit points” (respond) for event data within an event processing map. So-called *event adapters* are the most common kind of sockets and serve as the actual interface between a SARI application and underlying source systems: Depending on their implementation, event adapters translate real-world actions and state changes (such as a user *actually* placing a bet in an online-gambling platform) into event representations of a certain event type (sense), and vice versa (respond). Typical event-adapter implementations exchange data via message-queue systems, read them from log files, or invoke methods on an API.

**Event services.** Event services receive events from sockets or other event services, process them based on implementation-specific logic and, where necessary, feed back to the event processing map through output events. Event services cover the actual event-processing logic of an SARI application; typical implementations could, for instance, filter duplicate events, publish events to a data repository for later analysis, or serve as a hub.

**Rule services and decision graphs.** Rule services (3.2c.) are special event-service implementations that allow evaluating a collection of so-called *decision graphs* on the incoming event stream. Decision graphs describe in an integrated model (i) a noteworthy event situation – a so-called *event pattern* – to be fulfilled by one or more events of an event stream, and (ii) one or more actions to be executed when such a pattern is detected. In its basic form, the action part of a decision graph always describes the generation of one or more *response events*; an exemplary decision graph could, for instance, detect a certain fraud technique and create an event of type “Fraud Alarm”, with event attributes for the involved user account, etc. Depending on the function of a decision graph within SARI, a response event may be subject to further event-processing steps or trigger a real-world action in the underlying source system.

| <b>Sense</b>       |   |
|--------------------|---|
| <b>Description</b> | What actions and state-changes occur in the business environment?   |
| <b>Function</b>    | Continuous capturing of raw data, indicating real-world events, from underlying source systems. Unification of these data based on their basic semantics and delegation to SARI's event-processing facilities.  |
| <b>Example</b>     | A user logged in as "JQ Public" places a bet on the website of an online-gambling provider. SARI retrieves chunks of data from the provider's web servers via a messaging system. Depending on their semantics, incoming data are transformed to events of a respective event type, e.g., into events of type "Bet Placed", "Bet Canceled", etc.  |
| <b>Interpret</b>   |   |
| <b>Description</b> | What do the captured data indicate? What do the data mean for the current situation of the organization?  |
| <b>Function</b>    | Transformation of the captured events into higher-level business information such as business situations and key performance indicators. (Key performance indicators are not discussed as part of SARI's basic event-processing capabilities as presented in this chapter, but are introduced along with their integration to SARI's approach to rule-based event-processing in Chapter 6.) |
| <b>Example</b>     | Incoming events relating to a concrete betting process – e.g., indicating the placement, possible adaptations, and the final result of bet #1742 – are identified as related, in sum representing a coherent real-world situation.  |
| <b>Analyze</b>     |   |
| <b>Description</b> | Which business opportunities and risks arise in the business environment? What are the possibilities to improve the current situation of the organization?  |
| <b>Function</b>    | Analysis of incoming business information for exceptional situations in the business environment and time-critical business opportunities. Prediction of the performance and assessment of the risks for changing the business environment.   |
| <b>Example</b>     | The behavior of user account "JQ Public" is continuously monitored for fraudulent behavior. Since bet #1742 is one of several bets from this user concerning a sports event that was identified as potentially fixed by an international early-warning system, SARI detects a potential fraud for "JQ Public" and bet #1742.  |
| <b>Decide</b>      |   |
| <b>Description</b> | Which strategy is the best to improve the current situation of the organization? What are the actions required to successfully put a decision into action?  |
| <b>Function</b>    | Finding the best option for improving the current business situations. Determine the most appropriate action for a response to the business environment.  |
| <b>Example</b>     | Having detected fraudulent behavior of user account "JQ Public", SARI decides to immediately block the concerned account from all online activities, including the possibility to cash-out money.   |
| <b>Respond</b>     |   |
| <b>Description</b> | Who has to implement the decision? How can the decision be put into action?   |
| <b>Function</b>    | Respond to business environment by communicating the decision to the business environment as a command or suggestion (e.g. by e-mail), or by directly adapting and reconfiguring business processes in a source system.   |
| <b>Example</b>     | The decision to block "JQ Public" from all online activities is forwarded to the provider's web server via a messaging system. The web server implements the decision and marks "JQ Public" as blocked in the user-management system.   |

**Table 3.1.** Stages of the Sense-and-Respond Loop [114, 131]

## Outlook

The remainder of this chapter is structured as follows: Section 3.2 presents a model-driven view on SARI, splitting the overall complexity of a SARI application into a collection of smaller, easier-to-understand sub-models. The different sub-models of this view, namely the event model, the correlation model, and the event processing model, are discussed in great detail in Section 3.3 to Section 3.5; the central decision-graph model, serving as the immediate foundation for our research contributions, receives special attention in Chapter 4. Section 3.6 provides an overview of SARI's basic implementation architecture. We conclude this chapter with a brief introduction to Event Access Expressions, a tailored language for accessing streams of events, in Section 3.7.

## 3.2 A Model-Based View on SARI Applications

Within the inherently generic Sense-and-Respond Infrastructure, SARI applications describe environment-specific integration approaches and problem-specific processing logic for concrete use cases. To provide manageability and usability also for large-scale solutions, SARI splits the overall definition of an application into a collection of smaller, decoupled sub-models. Each of these sub-models describes a certain aspect of a solution, beginning with the structure of all possible event data and ending with the orchestration of self-contained event-processing units such as event adapters and event services.<sup>2</sup> Figure 3.3 shows the various sub-models of the SARI application model, along with the relationships between them.

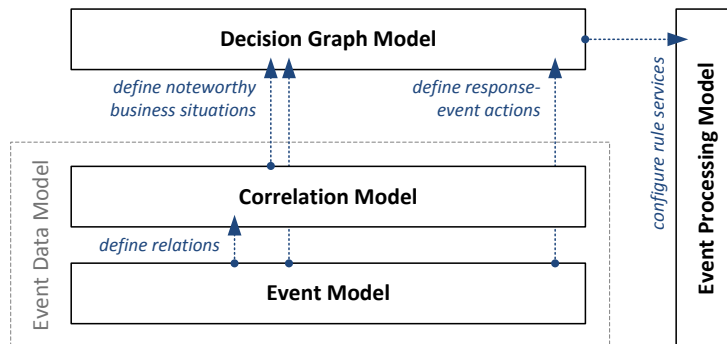


Figure 3.3. SARI Application Model

<sup>2</sup> A comparable, also model-driven approach has recently been used by Etzion and Niblett [37] for describing event processing in a platform-independent manner.

### Event Model

The event model provides abstract descriptions of all kinds of events that may occur within a SARI application, i.e., that may emerge directly from underlying source systems or be created virtually during the event processing. These descriptions – referred to as *event types* in the remainder of this thesis – declare all relevant characteristics of the represented occurrence itself and the context in which it occurs. For example, in the logistics domain, an event model would typically define event types such as “Order Placed”, with properties such as the corresponding user account and the kinds and quantities of goods, “Order Shipped”, etc. Event types form the foundation of any SARI application and allow all higher-level models to be defined in a type-safe manner.

The event model is discussed in greater detail in Section 3.3.

### Correlation Model

The correlation model defines how events of different event types relate to each other, i.e., whether they belong to a coherent sequence of real-world business occurrences such as a business process. For example, given an event model with two event types “Order Placed” and “Shipment Started”, a correlation relationship “Order Process” could link pairs of “Order Placed” and “Shipment Started” events by their order IDs. At run time, so-defined classes of event situations are then used to partition the overall set of events and handle these partitions separately within the SARI application’s event-processing logic.

The correlation model directly builds upon the *event model* to define relationships between events in an abstract manner. By itself, the correlation model serves as a basis for the *decision graph model*, through which application designers define classes of “noteworthy” event situations by imposing additional constraints on sets of correlated events.

The correlation model is discussed in greater detail in Section 3.4.

### Event Data Model

A SARI application’s event model and correlation model together form the application’s event data model, providing an abstract description of the underlying business environment on the level of individual occurrences (event model) as well as event situations (correlation model). While generally decoupled from the overlaying event-processing logic, it is obvious that a well-designed, targeted event data model “as thin as possible, as rich as necessary” is crucial for efficient application development.



### Decision Graph Model

The decision graph model defines event-processing logic of the form “if a noteworthy event situation – a so-called *event pattern* – occurs in an event stream, then execute respective reaction logic” in so-called *decision graphs*. Decision graphs are directly interpretable to SARI and form the basis for any rule-based event processing within the system.

In its basic form, the decision graph model builds upon the *event data model* for pattern detection purposes and the *event model* for defining reaction logic.<sup>3</sup> An event pattern may be considered as an additional constraint on either a class of individual occurrences as defined in the event model or on a class of business situations as defined in the correlation model. For example, an event pattern “order delayed by  $x$  days” would select from the overall set of all “order processes” only those cases that are delayed by  $x$  days or longer. The action part of a decision graph allows generating response events as defined in the event model. By themselves, decision graphs are referenced in the *event processing model* to be executed as part of a specific processing path through a SARI application.

Due to its paramount importance for the presented research contributions, the decision graph model receives special attention in Chapter 4 of this thesis.

### Event Processing Model

As a fourth and final sub-model, the event processing model defines

- (i) how real-world actions and state changes shall be translated into events of respective event types,
- (ii) how these events shall be processed in order to deduce valuable knowledge and act properly on exceptional business situations, and
- (iii) how the system shall feed back into the underlying business environment as orchestrations of self-contained event-processing units, so-called *event processing maps*. The elements of an event processing map produce and/or consume events and are connected to each other through a set of *channels*.

The event processing model builds upon the *event data model*, which allows event processing maps to be defined in a type-safe manner. Decision graphs as defined in the *decision graph model* are mapped to *rule services*, special event services that evaluate sets of decision graphs on the incoming stream of events and publish possible response events.

The event processing model is discussed in greater detail in Section 3.5.

<sup>3</sup> Extensions to the base decision-graph model are discussed in Chapter 5, Chapter 6, and Chapter 7.

### 3.3 Event Model

Forming the bottom layer of the proposed architecture, the event model provides abstract descriptions of all kinds of events that may emerge from underlying source systems or be created virtually during the event processing.

SARI’s approach to event typing as fully described by Rozsnyai et al. [108] is oriented towards the type systems of modern object-oriented programming languages. Enhanced typing concepts such as *duck typing* and *virtual event types* contribute to the aim of a straightforward, end-user oriented approach to application development; these concepts are, however, outside the scope of this thesis.

#### 3.3.1 Meta Model

Figure 3.4 sketches the base meta-model for a SARI application’s event-type library. An event type

$$T = \{a_1, a_2, \dots, a_n \mid a_j = (i_j, t_j)\}$$

is defined by a set of event attributes, each having a unique identifier  $i$  and an attribute type  $t$ . SARI features three basic kinds of attribute types:<sup>4</sup>

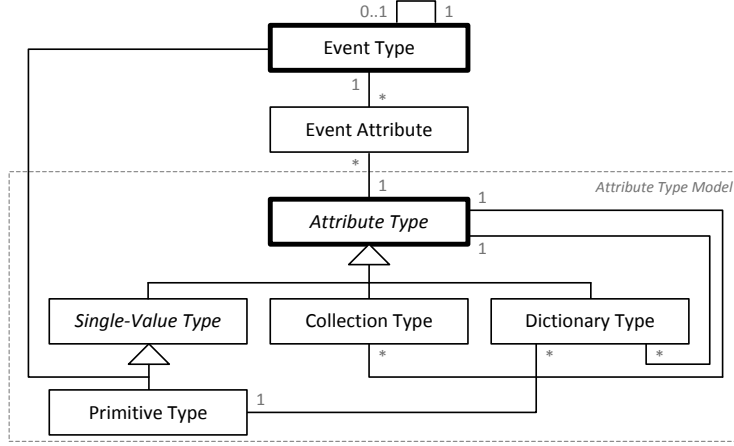
- *Single-value types* include primitive types (such as integers, strings, etc.) as well as event types (i.e., an event may hold auxiliary events as event attributes)
- *Collection types* are lists of attribute-typed elements
- *Dictionary types* associate attribute-typed values with primitive-typed keys

An event type  $T'$  may furthermore be in a subtype relationship with a base event type  $T$ ,  $T \text{ :> } T'$ . As usual, a sub event-type inherits all event attributes from the base type,  $T \subseteq T'$ . By definition, each event type in a SARI application must originate from a root event type “Base Event”. The “Base Event” type defines a time stamp “Creation Time” – holding an event’s time of occurrence – as well as a unique identifier “ID”.

Each event occurring in a SARI application must then be an instance of exactly one event type  $T$  as defined in the application’s event model, i.e., define a concrete event-attribute value for each event attribute defined in  $T$ . Formally, a SARI event  $e_T : T$  is therefore defined by a set of event-attribute/value tuples,

$$e_T = \{(a_1, v_1), (a_2, v_2), \dots, (a_n, v_n) \mid a_n = (i_j, v_j), \{a_1, a_2, \dots, a_n\} = T, v_j : t_j\}$$

<sup>4</sup> Unless otherwise stated, the described attribute-type model applies to all (non-event-) data types as discussed in the remainder of this thesis.



**Figure 3.4.** Event Type Meta-Model

where  $e_T$  contains a tuple for each event attribute in  $T$  and each event-attribute value conforms to the respective event-attribute type.

For the purpose of this work, we define the following methods on SARI events as defined above:

Let  $\mathbb{T}$  be the set of all event types and let  $\mathbb{E}$  be the set of all events. We define a method  $\text{typeof} : \mathbb{E} \rightarrow T$ , returning the runtime event type of a given SARI event  $e$ , as follows:

$$\text{typeof}(e) = T \mid e : T$$

Let  $e$  be a event of an event type  $T$ ,  $e : T$ , and let  $a = (i, t)$  be an event attribute in  $T$ ,  $a \in T$ . We define a method  $\text{value}_e : a \rightarrow t$ , returning the value of  $a$  in  $e$ , as follows:

$$\text{value}_e(a) = v \mid (a, v) \in e$$

### 3.3.2 Example

Figure 3.5 shows an exemplary event type “Order Received”, signifying the real-world action of an order being handed over to its receiver. Apart from the ever-present “ID” and “Creation Time” attributes, the event type declares an integer-typed “Order ID”, an integer-typed “Customer ID”, and a list of “Order Positions” of type “Order Position”. An order position could, for instance, be defined by a string-typed “Item ID” and an integer-typed “Quantity”.

| Order Received         |                       |
|------------------------|-----------------------|
| <b>ID</b>              | [GUID]                |
| <b>Creation Time</b>   | [DateTime]            |
| <b>Order ID</b>        | [Integer]             |
| <b>Customer ID</b>     | [Integer]             |
| <b>Order Positions</b> | [List<OrderPosition>] |

Figure 3.5. Exemplary Event Type

## 3.4 Correlation Model

Setting up on the event model, the correlation model defines how events of different event types relate to each other, i.e., whether they belong to a coherent sequence of real-world business occurrences such as a business process. So-defined classes of event situations allow partitioning the overall set of events during event processing and form the basis for the decision graph model as described in Chapter 4.

In SARI, correlation information is defined in so-called *correlation sets* [77, 112], a declarative model that allows incorporating and combining different correlation approaches through *correlation bands*. Each correlation set thereby corresponds to one class of event situations: For instance, in the logistics domain, a correlation set “Shipment” might correlate the events of all shipment processes as emerging from the source system. A concrete event-situation instance – e.g., the events of the certain shipment process #42 – is referred to as *correlation session*.

### 3.4.1 Meta Model

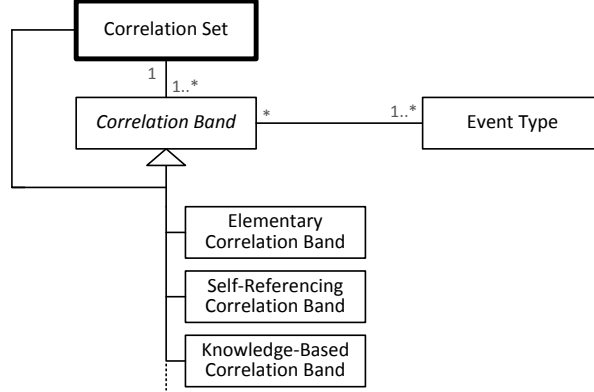
Figure 3.6 sketches the meta-model for correlation sets. A correlation set

$$c = \{b_1, b_2, \dots, b_n\}, n \geq 1$$

is defined by a non-empty collection of correlation bands. Each correlation band describes a specific correlation approach for events of one or more event types as defined in the SARI application’s event model, thereby defining a certain part of the overall event situation. Available correlation-band implementations are described in Section 3.4.2 below.

For the purpose of this work, we define the following methods on correlation bands and correlation sets:

Let  $b$  be a correlation band and let  $\mathbb{E}$  be the set of all events. We define a method  $\text{correlate}_b : \mathbb{E} \times \mathbb{E} \rightarrow \{0, 1\}$  as follows:



**Figure 3.6.** Correlation Set Meta-Model

$$\text{correlate}_b(e, f) = \begin{cases} 1 & e \text{ and } f \text{ are correlated with respect to } b \\ 0 & \text{otherwise} \end{cases}$$

Let  $c$  be a correlation set and let  $\mathbb{E}$  be the set of all events. We define a method  $\text{correlate}_c : \mathbb{E} \times \mathbb{E} \rightarrow \{0, 1\}$  as follows:

$$\text{correlate}_c(e, f) = \begin{cases} 1 & \exists b \in c : \text{correlate}_b(e, f) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Let  $c$  be a correlation set, let  $\mathbb{E}$  be the set of all events, and let  $S = \{s_1, s_2, \dots, s_n\}$ ,  $s_i = \{e_1, e_2, \dots, e_m \mid e_i \in \mathbb{E}\}$  be the collection of correlation sessions for  $c$ . We define a method  $\text{session}_s : \mathbb{E} \rightarrow \mathbb{E}^*$ , returning the correlation sessions for an event  $e$  with respect to  $c$ , as follows:

$$\text{session}_c(e) = \begin{cases} s \cup \{e\} & \exists s \in S : \exists f \in s : \text{correlate}_c(e, f) = 1 \\ \{e\} & \text{otherwise} \end{cases}$$

### 3.4.2 Correlation Band Implementations

At the time of writing, SARI features the following correlation bands:

**Elementary correlation bands** correlate events of different types based upon equal event-attribute values. Let  $T = \{T\}$  be an event-type library. An elementary correlation band is defined by  $e \subseteq \{(T, a) \mid T \in T, a \in T\}$ , i.e., a non-empty set of event types together with an event attribute per type. Given a correlation band  $e = \{(T_1, a_1), (T_2, a_2), \dots, (T_n, a_n)\}$ , two events  $e_i : T_i$  and  $e_j : T_j$  are correlated (and thus part of the same correlation session) if  $\text{value}_{e_i}(a_i) = \text{value}_{e_j}(a_j)$ . Note that a correlation band's event attributes do not necessarily have the same identifier.

**Self-referencing correlation bands** allow implementing scenarios where events explicitly define references to their (causal) predecessors. As with elementary correlation bands, a self-referencing correlation band  $s \subseteq \{(T, a) \mid T \in T, a \in T\}$  is defined by a nonempty set of event types, and, for each event type, an event attribute. Two events  $e$  and  $f$ ,  $f : T_i$ , are then correlated if  $\text{value}_e(\text{ID}) = \text{value}_f(a_i)$ , where ID signifies the unique identifier attribute of an event.

**Knowledge-based correlation bands** are similar to elementary correlation bands, however, for evaluating “equality” between event-attribute values, an external knowledge base is queried. For example, two string values “Vienna” and “Wien” could be detected as equivalent via an (online) dictionary. A knowledge-based correlation band  $k = (e, b)$  therefore extends an elementary correlation set  $e$  by a knowledge base  $b$ , offering methods for testing equality between two event-attribute values. Knowledge-based correlation bands have been investigated in great detail by Moser et al. [82].

**Correlation sets** may be (re-)used as correlation bands in higher-level correlation sets. Though rarely applied in practice, this enables the hierarchical modeling of event situations.

### 3.4.3 Example

Figure 3.7 shows an exemplary correlation set from the logistics domain. An elementary correlation band  $b_1$  describes the correlation approach for “Order Received”, “Shipment Ready” and “Order Shipped” events as being based upon equal order IDs. A second, self-referencing correlation band  $b_2$  describes the correlation approach for “Order Shipped” and “Shipment Received” events as being based upon an explicit reference from the “Shipment Received” events to the causally preceding “Order Shipped” event.

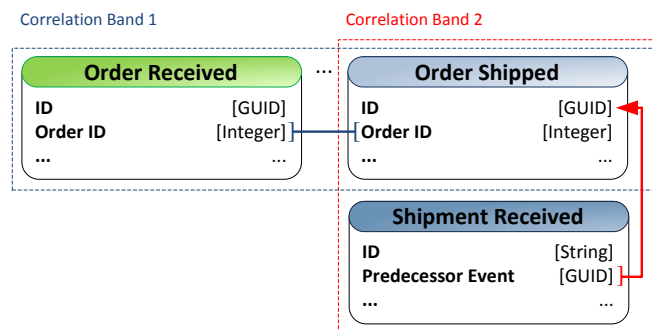


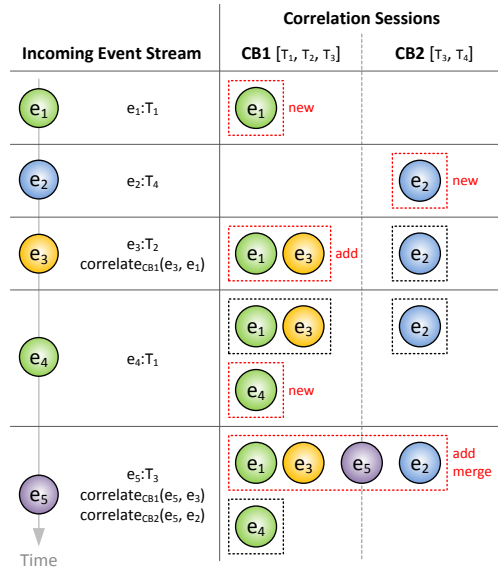
Figure 3.7. Exemplary Correlation Set

### 3.4.4 Correlating Events at Run Time

Within SARI, event correlation is made accessible to the user-defined event-processing logic of a SARI application through a system-wide *correlation service*. This service allows (i) correlating a given event based on the application’s correlation model, and (ii) retrieving all correlations sessions in which this event participates. SARI thereby follows an “on demand” approach to event correlation, meaning that an event must explicitly be correlated in order to be accessible as part of a correlation session in subsequent retrievals.

At run time, each correlation band of a SARI application’s correlation model is used independently for grouping incoming events into respective correlation sessions. Correlation sets finally serve as bridges between subordinate correlation bands: Whenever two or more correlation sessions for correlation bands of a common correlation set intersect, these correlation sessions are merged into a single, common correlation session. Incoming events are then added to the newly-merged correlation session when they correlate with an element of the merged session according to at least one of the associated correlation bands.

Figure 3.8 illustrates the described handling of correlation sessions for a simple correlation set  $s = (b_1, b_2)$ , where  $b_1$  defines an event-correlation approach for events of type  $T_1, T_2$  and  $T_3$ , and  $b_2$  defines an event-correlation approach for events of type  $T_3$  and  $T_4$ .



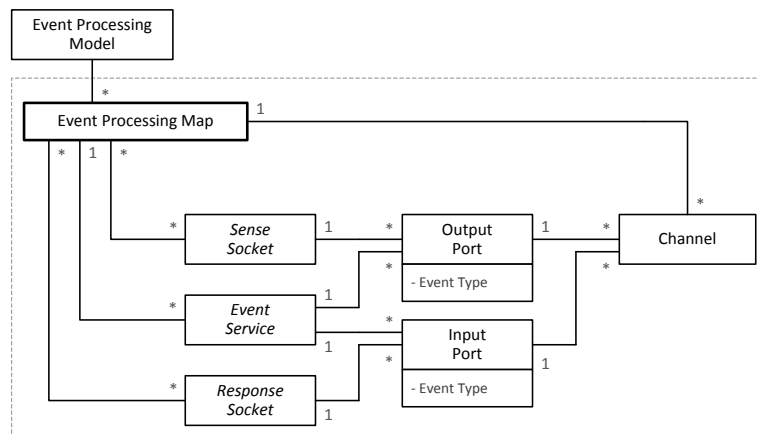
**Figure 3.8.** Correlating Events at Run Time

### 3.5 Event Processing Model

Setting up on the event data model and the decision graph model, the event processing model describes the overall functionality of a SARI application, as well as its integration with underlying source systems, by means of one or more *event processing maps*. Event processing maps are user-defined orchestrations of *sense sockets*, *event services*, and *response sockets*. Following the idea of a service-oriented approach to event processing, these elements operate as independent, self-contained processing units that may receive input events from other elements of a map, and/or publish output events. Output events produced by a map element are routed to other map elements according to a set of *channels* between these elements. Event processing maps are comparable to the concept of *event processing networks* [37, 120].

#### 3.5.1 Meta Model

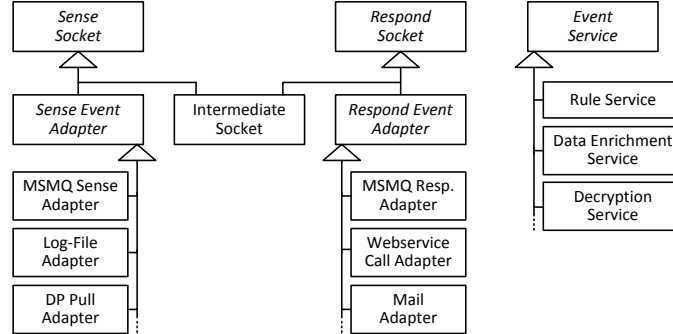
Figure 3.9 and Figure 3.10 show the meta-model for event processing maps.



**Figure 3.9.** Event Processing Meta-Model

Sense sockets introduce new events into an event processing map. Depending on the implementation of a socket, these events may emerge directly from the underlying business environment (in case of sense event adapters; see Section 3.5.2), or come from another event processing map (in case of intermediate sockets; see Section 3.5.4). Sense sockets are associated with event processing maps in an *m-to-n* relationship, meaning that a single socket can be used in several maps. In such a case, separate instances of an event are published to all event processing maps in which a socket is used.





**Figure 3.10.** Map Element Hierarchy

Event services receive events from sockets or other event services, process them based on implementation-specific logic and, where necessary, feed back to the event processing map through output events. Event services are associated with event processing maps in an 1-to- $n$  relationship, meaning that each event service instance belongs to exactly one event processing map.

Respond sockets consume events as generated in, or routed through, the orchestration of event services. Depending on the implementation, outgoing event data may be routed back into a source system (in case of respond event adapters; see Section 3.5.2), or be forwarded to other event processing maps (in case of intermediate sockets). As with sense sockets, response sockets are associated with event processing maps in an  $m$ -to- $n$  relationship

Sense sockets, event services and response services publish and/or receive events through collections of typed input ports and output ports, each associated with an event type as defined in the SARI application's event model. Depending on the implementation of a map element, these collections may either be user-defined (for rule services and intermediate sockets), or predetermined by the implementation (for any other map element). At run time, whenever a map element produces an event  $e$ ,  $e$  is published via all output ports of the respective element with an event type  $T_{out}$  conforming to  $e$ , i.e., where  $T_{out}$  is a super-type relationship to the implementation type of  $e$ ,  $T_{out} :> \text{typeof}(e)$ . When there is no output port fulfilling these conditions,  $e$  is discarded and not considered for further event processing steps.

Channels associate an output port  $out$  of a map component  $c_i$  with an input port  $i$  of another map component  $c_j$ , indicating that events as published via  $out$  shall be routed to  $i$ . Building upon typed ports rather than on map components directly, channels allow type-safe communications between the elements of an event processing map: In the proposed model, channels can only be established between pairs of output- and input ports that are compatible to each other, i.e., where the event type  $T_{out}$  of the output port is in a subtype relationship to the event type  $T_{in}$  of the input port,  $T_{in} :> T_{out}$ .

### 3.5.2 Event Adapters

Event adapters represent the technical interface between SARI and underlying business environments. An event adapter may be considered as a data gateway that, on one end, plugs into a source system in order to receive or publish data in a technology-specific format, and, on the other end, exchanges event data with the various event-processing maps of the SARI application. *Sense event adapters* detect actions and state-changes in the source system and translate these data into events of respective event types as defined in the SARI application's event model: A sense event-adapter could, for example, pull data from a message queue of "new orders" as being placed on an online-shopping platform, and based on these data, publish events of type "New Order" with event attributes describing the kind and quantity of goods and a customer identifier. *Response event adapters* retrieve events from the various event-processing maps to which they belong and transform these events into respective actions to be triggered in the source system: A sense adapter could, for instance, insert data into a database or message queue, call an API, or send an email.

Event adapters can be implemented as custom .NET assemblies by extending a SARI-provided base class, which enables application developers to perfectly align their event adapters with the given IT infrastructure. SARI distributions typically ship with a collection of predefined, configurable event adapters for the most common integration points, e.g., for diverse messaging systems, web services, or log files. For a more detailed discussion on event adapters and possible implementation approaches, the interested reader may refer to Roth [103].

### 3.5.3 Event Services

Operating between "sense" and "respond", orchestrations of event services describe the problem-specific event-processing logic of a SARI application. Depending on its implementation, each event service thereby represents a particular unit of work to be applied on incoming events. Most prominently, this includes the filtering of events, their transformation (including their enrichment by external data), event-data aggregation, or situation detection; still, any other event-triggered activity is possible as well. The ultimate outcome of an event processing map's network of event services is routed to the map's collection of respond sockets, through which they feed back into the business environment.

As with event adapters, event services can be implemented as .NET assemblies by extending a SARI-provided base class. Still, custom event-service implementations not only require a development environment and programming skills, but also are cumbersome to change during a SARI application's lifecycle. It is

therefore advisable to use a rule-based approach to event processing, incorporating *rule services* and decision graphs as described in Chapter 4, wherever applicable.

Rule services are special event services that allow evaluating a number of decision graphs on the incoming event stream. In their basic form, the action part of a decision graph always describes the generation of one or more response events; such response events are then published via the rule service's output ports as discussed above. As a consequence, a rule service not only provides the evaluation environment for decision graphs but also defines the specific context(s) in which a decision graph operates.

#### 3.5.4 Intermediate Sockets

Intermediate sockets are special map elements that may be used in event processing maps both as sense sockets and as respond sockets, where incoming events are immediately and without changes re-published to the event processing model. Intermediate sockets hence provide a communication interface between the various event-processing maps of a SARI application, allowing application designers to split a complex event-processing flow into a set of smaller, easier-to-understand event processing maps.

#### 3.5.5 Example

Figure 3.11 shows a simple event processing map for integrating and processing events from a workload automation system. On its input side, the map captures task events as well as different kinds of resource-tick events through tailored sense adapters; while task events indicate the completion of a task in the workload automation system, resource tick events provide information about the current resource utilization at regular time intervals. In the "Enrichment" service, incoming task events are enriched to prepare the event data for later rule processing. A typical example would be the attachment of additional task data; consider, for instance, scenarios where a source event only holds a task ID while the task's estimated runtime is required for the downstream decision-making logic. The "Unification" service creates from the different kinds of resource-tick events a single, unified resource-update event, eventually publishing the most recent state of a IT landscape whenever some kind of update is signified. The central rule service processes the incoming event according to a collection of decision graphs and publishes response events to the map's response-event adapters. These adapters transmit the response events to external systems by sending emails or invoking a script in the underlying workload automation system.

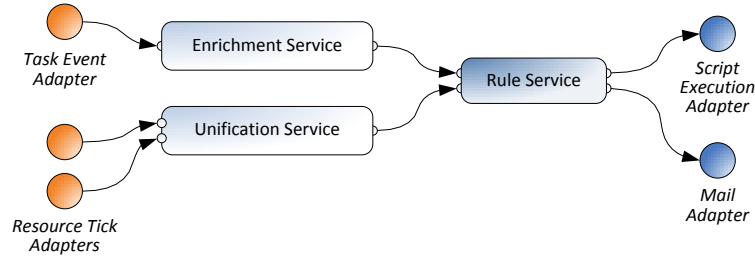


Figure 3.11. Exemplary Event-Processing Map

### 3.6 System Architecture

SARI was designed to provide efficient and reliable event processing across arbitrary business domains. To meet these requirements also in large-scale environments, SARI is implemented as a distributed system of self-contained event-processing nodes, enabling companies to balance the overall event-processing workload of their SARI applications across a network of virtual or physical hosts. SARI application development, the distribution of SARI applications over the network of event-processing nodes, as well as the ex-post analysis of historic event data is made accessible to technically versed end users through a suite of graphical front-end tools.

Figure 3.12 provides an overview of the basic SARI architecture as it stands at the time of writing. This architecture can be separated both vertically – into a data layer, a back-end layer and a front-end layer – and horizontally, into components concerned with *application management* and components concerned with *application execution and monitoring*.

#### 3.6.1 Data Layer

Forming the bottom layer of the presented architecture, the data layer is concerned with the persistent management of *system-level* as well as *application-level* data in a collection of relational databases. At the time of writing, SARI distributions come with native support for recent versions of Microsoft SQL Server, Oracle, and DB2.

The *administration database* (Figure 3.12a.) contains all system-level data, i.e., all data that are not created as a result of executing a particular SARI application. Most notably, these data include application descriptions for all available SARI applications, deployment groups, as well as data from the installation's user management system. An *application description* (b.) is an XML representation of the static structure of a SARI application, including all event types, correlation sets, decision graphs, and event processing maps as described in

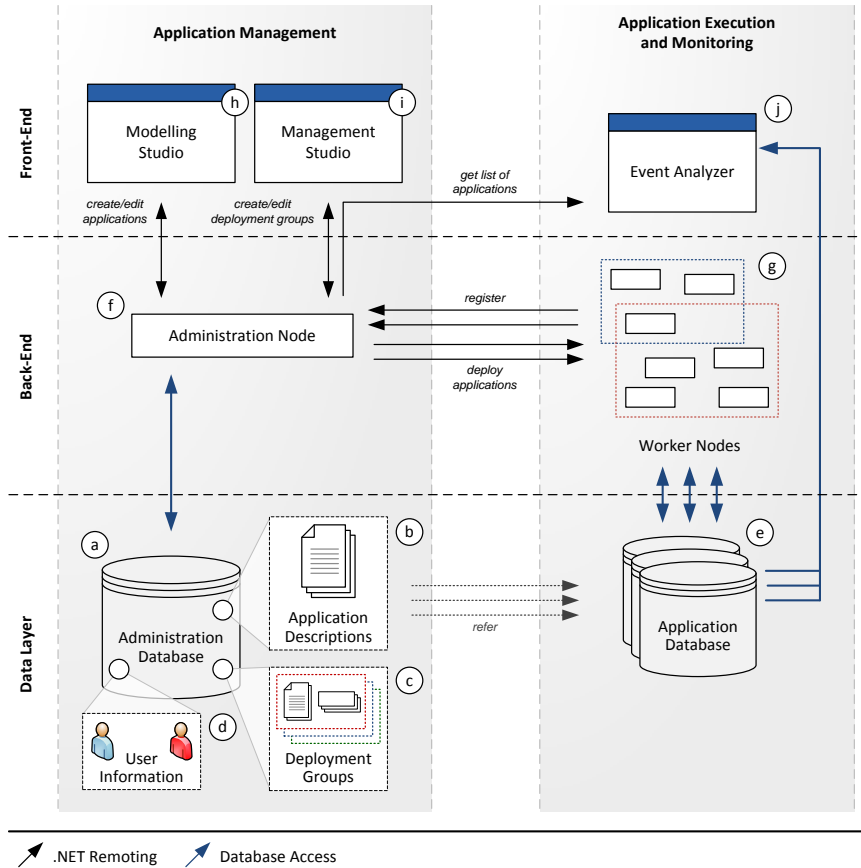


Figure 3.12. Implementation Architecture

the above sections. *Deployment groups* (c.) describe the distribution of SARI applications across the network of *worker nodes* and are discussed in greater detail in Section 3.6.2 below. *User data* (d.) include login information as well as respective user roles and are typically queried whenever attempts to log on to a front-end tool. The administration database is directly accessed by the *administration node* only.

*Application databases* (e.) exist per SARI application and contain all data that are produced during the execution of the corresponding applications. Application-level data include, among others, current event and correlation data (for the synchronization of multiple worker nodes) and historic event data (for the ex-post analysis of a system). Application databases are referenced in the application description of the corresponding SARI application and continuously accessed by the different *worker nodes* of the system. In

the front-end layer, the *Event Analyzer* may query an application database in order to retrieve (and, in further consequence, visualize) historic event data.

### 3.6.2 Back-End Layer

SARI's back-end layer is responsible for executing the system's SARI applications in a scalable and failsafe environment. It includes a central coordination unit – the so-called administration node – as well as a network of event-processing nodes, the so-called worker nodes. SARI's back-end components are implemented as .NET assemblies that can be executed as console applications or Windows services. For interacting with each other and/or SARI's front-end layer, .NET Remoting is used.

The *administration node* (f.) is the key element of the SARI architecture and a prerequisite for creating, managing, and deploying SARI applications. Most basically, the administration node serves as an interface to the different SARI applications of a system: As depicted in Figure 3.12, the administration node is the only element of the SARI architecture that has direct access to the administration database and the current set of application descriptions. All other components must connect with the administration node in order to query, create, or update SARI applications, and/or to register as listeners to be notified in case of changes.

For their actual execution in the SARI system, SARI applications are distributed from the administration node to a network of *worker nodes* (g.). Worker nodes are generally independent, self-contained execution units which are able to run in parallel an arbitrary number of application instances. The collection of applications to be hosted by a given worker node is defined using above-mentioned *deployment groups*. These associate in a *m-to-n* relationship

- (i) a set of SARI applications, and
- (ii) a set of worker nodes these applications shall be executed on.

It follows from the *m-to-n* relationship that several instances of a SARI application may exist in parallel, distributed across multiple worker nodes. These application instances operate generally independent from each other, meaning that sense event adapters, event services, and respond event adapters run as separate instances on each worker node. To avoid duplicate processing of events, sense-adaptor implementations must ensure that real-world occurrences are transformed by exactly one instance. Synchronization between multiple worker nodes is based on the concept of correlation sessions, which are shared among all instances of an application. For more details on the use of correlation sessions for synchronization purposes, the interested reader may refer to the work of McGregor and Schiefer [77].

### 3.6.3 Front-End Layer

SARI distributions come with a suite of graphical front-end tools for creating and administrating SARI applications, controlling their deployment on the network of worker nodes, and analyzing their performance based on historic event data. All these tools are .NET-based Windows applications and connect with the administration node via .NET Remoting.

The *Modeling Studio* (h.) is an extensive, power-user-oriented IDE that allows creating, editing, and deleting SARI applications. Graphical editors are provided for all logical parts of a SARI application as described in the above sections, including event types, correlation sets, decision graphs, and event processing maps. A new or updated SARI application is forwarded to the administration node, which persists the respective application description and triggers its (re-)deployment on the system's set of worker nodes.

The *Management Studio* (i.) allows authorized users to administrate deployment groups, as well as to manually deploy or un-deploy SARI applications on the current set of worker nodes.

The *Event Analyzer* (j.) enables business analysts to investigate the behavior of a SARI application based on historic event data, using tailored event-data visualizations and data-mining tools. The key visualization techniques of the Event Analyzer build upon the Event Tunnel metaphor as discussed in great detail by Suntinger et al. [123]. Other visualizations include a text view (rendering events and correlation sessions as a chronologically ordered, pageable list similar to the renderings of modern WWW search engines) and an Excel-based spreadsheet view. Event-based similarity searching [92] operates in full integration with the Event Analyzer's visualization techniques, allowing analysts to highlight those correlation sessions that are most similar to a reference event sequence.

## 3.7 Event Access Expressions

Across all parts of SARI's application model and implementation architecture, access to event instances is unified through a tailored, business-user-oriented event-access language, so-called *Event Access (EA) Expressions*. EAExpressions can generally be separated into (i) expressions on individual events of specified event types, and (ii) expressions on sequences of events, typically containing events of multiple event types. In both forms, EAExpressions play a crucial role in the detection of noteworthy event situations using *decision graphs* as discussed in the following chapter. In the following, we give an introduction to the basic syntax and functionality of EAExpressions; for more detailed discussions, the interested reader may refer to Rozsnyai et al. [105, 106].

### 3.7.1 Basic Operators

EAEExpression statements can be composed from lower-level expressions through a collection of basic operators well-known from common programming languages.

**Relational operators** include “equal”, “not equal”, “greater than”, “greater than or equal”, “smaller than”, as well as “smaller than or equal”, all in Pascal-like syntax. While equality and non-equality can be applied to expressions of any type, “greater than”, “greater than or equal”, “smaller than”, and “smaller than or equal” are restricted to expressions of continuous types, i.e., numeric values, timestamps, and time spans.

**Boolean operators** include the three basic binary operators “AND”, “OR” and “XOR” as well as the unary “NOT” operator. Boolean operators can be applied to (pairs of) Boolean values only.

**Arithmetic operators** include addition, subtraction, multiplication, division, and modulo, all in C-like syntax. Being typically applicable to numeric operands, EAEExpressions allows using “addition” and “subtraction” to perform selected calculations on time stamps and time spans: Two time stamps may, for instance, be subtracted from each other, resulting in a time span representing the difference between these time stamps. Additions of two time stamps, by contrast, would be semantically wrong. When at least one operand is a string, the “+” symbol results in string concatenation.

### 3.7.2 Handling Collection Types

The SARI data model provides two collection types, namely *lists* and *dictionaries*. EAEExpressions provide access to the elements of both data structures in C-like brace notation; while collections must be provided the desired index, dictionaries must be provided the element’s key. EAEExpressions moreover provide diverse binary operators for collections, including, among others, “CONTAINS” (returning whether an element or a list of elements is contained in a collection) and “CONTAINSANY” (returning whether any element of a list of elements is contained in a collection).

### 3.7.3 Accessing Single Events

When defined on single events of a specified event type, EAEExpressions provide access to the various event attributes of an event through the attributes’ identifiers, which are available as respectively-typed literals; when the EAEExpression is evaluated on a certain event  $e$ , the concrete event-attribute values of  $e$  are retrieved.



An exemplary EAExpression on events of type “Order Retrieved” as discussed in Section 3.3.2, testing whether the event’s customer ID lies in a certain range, could thus be defined as follows:

$$(\text{CustomerID} \geq 1000) \text{ AND } (\text{CustomerID} < 2000)$$

### 3.7.4 Accessing Sequences of Events

Being strongly-typed, EAExpressions on sequences of events must restrict the overall set of events to events of a certain event type before particular event attributes can be accessed. Such a reduction is achieved through (generally *type-based*) filtering, which forms the key element of most practical event-sequence expressions. Having selected events of a certain event type, sets of event-attribute values may be established in common dot-notation.

The most basic filtering approach is called *implicit filtering*, where an event-attribute identifier is simply prefixed with the name of the concerned event type. Applied on a given sequence of events, such an expression then accesses the most recent instance of that type in the sequence, i.e., the instance with the latest time of occurrence. An exemplary expression, accessing the customer ID of the most recent “Order Received” event, could thus be defined as follows:

$$\text{OrderShipped.CustomerID}$$

Apart from implicit filtering, filtering can be separated into *content-based filtering*, which is based on event-attribute values, and *index-based filtering*, which is based on the particular position of an event instance in the event sequence.

For content-based filtering, an EAExpression specifies the name of a concerned event type  $T$  along with a Boolean expression on events of type  $T$  in round brackets. Evaluated on a given stream of events, the expression then returns all  $T$  events that conform to the specified expression. An exemplary expression, retrieving all “Order Received” events with a customer ID lying in a certain range, could be defined as follows:

$$\text{OrderShipped}((\text{CustomerID} \geq 1000) \text{ AND } (\text{CustomerID} < 2000))$$

For index-based filtering, an EAExpression specifies the name of a concerned event type  $T$  along with an index or a range of indexes in square brackets: Ranges of indexes must be specified in Python-like “ $i..j$ ” syntax; while events are generally ordered by the time of their occurrence, an index  $i$  may be prefixed with “&” to refer to the  $i^{\text{th}}$  event in reverse order. Evaluated on a given stream of events, the expression then returns all  $T$  events on the specified index or within the specified range. An exemplary expression, retrieving the four most recent “Order Received” events, could thus be defined as follows:

OrderShipped [3..0]

Given an event type  $T$ , content-based filtering and index-based filtering can be combined in a common expression. Filters are thereby evaluated “from left to right”; consequently, the following expressions would yield different results:

OrderShipped(CustomerID > 1000) [3..0]

OrderShipped [3..0] (CustomerID > 1000)

Given a filtering on  $T$ -events as discussed above, an expression could eventually access the event-attribute values of all resulting  $T$ -events in common dot-notation. For example, the following expression would return the order IDs of all “Order Received” events with a customer ID greater than 1000:

OrderShipped(CustomerID > 1000).OrderID

### 3.7.5 Functions

EAEExpressions provide a whole range of functions on single as well as lists of values; for instance, the average of a list of numeric values could be calculated as follows:

Avg(OrderPlaced(TotalCosts > 1000).TotalCosts)

For a detailed listing of functions, the interested reader may refer to the work of Rozsnyai [105].



## Decision Graphs

**Abstract** Sense-and-Respond Infrastructure (SARI) is a general-purpose Complex Event Processing framework that has proved useful in a variety of business domains. In order to provide manageability and usability also for large-scale solutions, SARI splits the overall definition of an application into a collection of smaller, decoupled sub-models. This chapter presents the central *decision graph model* of a SARI application. Describing event-processing logic of the form “if a noteworthy event situation occurs in the incoming stream of event, then trigger respective actions”, it forms the foundation for any rule-based event processing in SARI. We present a meta model for decision graphs and discuss the relationship between decision graphs and event correlation at design time as well as run time. We introduce a library of *rule components*, which encapsulate easy-to-understand pieces of event-processing logic and form the nodes of a decision graph.<sup>1</sup>

### 4.1 Introduction

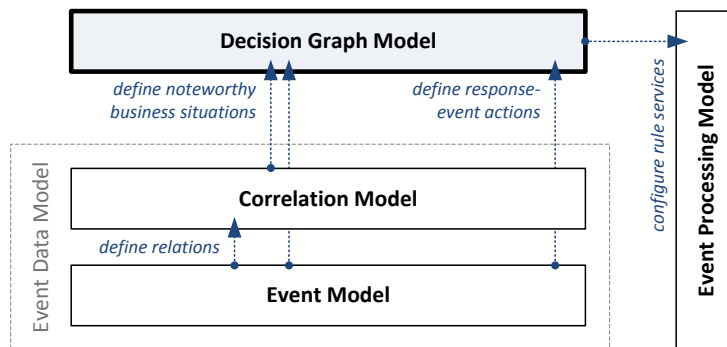
Sense-and-Respond Infrastructure (SARI) as originally proposed by Schiefer and Seufert [114] is a general-purpose Complex Event Processing framework that has proved useful in a variety of business domains such as online gambling, logistics, and automated product recommendation. In order to provide manageability and usability also for large-scale solutions, SARI splits the overall definition of an application into a collection of smaller, decoupled sub-models. Each of these sub-models then describes a certain aspect of a solution, beginning with the structure of all possible event data and ending with the orchestration of self-contained event-processing units such as event adapters and event services.

---

<sup>1</sup> This chapter is based on the work of Obweiger et al. [89].

In this chapter, we present the central *decision graph model*, which may be considered the key element of a model-driven view on SARI applications. Decision graphs describe in an integrated model (i) a noteworthy event situation – a so-called *event pattern* – to be fulfilled by one or more events of an event stream, and (ii) one or more actions to be executed when such a pattern is detected. Decision graphs are directly interpretable to SARI using a special event-service implementation and form the foundation for any rule-based event processing logic in the system. As such, the decision graph model directly underlies our approach to user-oriented rule management as presented in Chapter 5 of this thesis. Extensions to the base decision-graph model are presented in Chapter 6 and Chapter 7, where we use decision graphs for updating and monitoring *business entities* as well as *hierarchical pattern modeling*.

Figure 4.1 shows the decision graph model in the context of the overall SARI application model.



**Figure 4.1.** Decision Graphs in the SARI Application Model

The decision graph model builds upon the *event data model* for pattern detection purposes and the *event model* for defining reaction logic. An event pattern may be considered as an additional constraint on either a class of single business occurrences as defined in the event model or on a class of business situations as defined in the correlation model. For instance, an event pattern “order delayed by  $x$  days” would select from the overall set of all “order processes” only those cases that are delayed by  $x$  days or longer. The action part of a decision graph allows generating response events as defined in the event model. By themselves, decision graphs are referenced in the *event processing model* to be executed as part of a particular processing path through a SARI application.

#### 4.1.1 Key Characteristics

Schiefer et al. [112, 113] identified the following key characteristics of decision graphs, among others:

**Building rules with Divide and Conquer.** Powerful event-pattern rules are key to successful applications of CEP. Still, describing classes of noteworthy event situations in an abstract manner may place heavy demands on users. SARI aims to simplify this process by employing a “divide-and-conquer”-like approach to modeling rules, where application developers compose complex decisions from easy-to-understand pieces of event-processing logic such as “the occurrence of an event of type  $T$  with certain attribute values” or “the generation of a response event of type  $U$ ” These pieces – encapsulated in so-called *rule components* – are connected to each other in a directed, acyclic decision graph. At run time, the so-defined predecessors in the decision graph are then considered as preconditions in the event-processing logic: To activate a component  $c$  – and thus bring it to play into the evaluation process – a concrete event situation must conform to (at least) one valid path through the decision graph. Depending on the evaluation result of  $c$ , further parts of the decision graph are activated, and so forth.

**Decoupling event correlation and event-pattern detection.** In many CEP solutions, relationships between events as well as conditional restrictions (e.g., on the quantity, ordering, or attribute values of events) are defined in a single, integrated event-pattern model. SARI allows for a strict decoupling of *event correlation* on the one side and *event-pattern detection* on the other: An application’s correlation model defines how events generally relate to each other, e.g., whether or not two “Transport Update” events belong to the same delivery process. Decision graphs set up on a so-defined, “common” class of event situations and define those characteristics of a concrete situation instance that make it noteworthy in a certain sense. The proposed decoupling not only simplifies the creation of decision graphs, but also enables the reuse of correlation information across SARI, e.g., for analysis purposes.

**Event-triggered rule evaluation.** The main value of CEP systems consists in responding to noteworthy business situation in near real time, on the occurrence of a decisive business event. The evaluation of decision graphs is entirely event triggered, meaning that any evaluation step is directly or indirectly caused by the occurrence of an event. As a consequence, detections of noteworthy event situations along with possible reactions to these occurrences can be carried out with minimal latency.

**Graphical rule modeling.** The described, graph-based composition of generally self-contained components is in full accordance with a graphical approach to rule modeling, enabling both a comprehensive view of the overall decision graph and quick and easy access to the various rule components. SARI features a graphical decision-graph editor, allowing users to add, configure, and

connect graphical representations of rule components. Possible renderings of rule components are presented in Section 4.3.

**Service-oriented rule processing.** Decision graphs are assigned to and executed by so-called rule services as described in Section 3.5, special event services that retrieve the correct decision-graph evaluation state for an incoming event and adapt this state based upon the represented event-processing logic.

#### 4.1.2 Decision Graphs and Rules

One may argue that due to their general semantics – “if event pattern, then action” – decision graphs would directly and without restrictions qualify for modeling event-pattern rules. Such an approach has been applied in earlier research on rule-based event processing in SARI [113]. However, while well-suited for technically versed users, it proved to be inappropriate for domain experts with little or no technical expertise.

In the remainder of this thesis, we introduce several abstraction layers that set up on decision graphs in order to ease the creation of meaningful event-pattern rules. Independent from these extensions, decision graphs continue to be the base model for rule-based event processing: Rule services execute a collection of decision graphs, no matter which abstraction layers may exist beyond them.

#### 4.1.3 Outlook

The remainder of this section is structured as follows: In Section 4.2, we present a meta model for decision graphs. Rule components, which encapsulate easy-to-understand pieces of event-processing logics and form the key elements of a decision graph, are presented in Section 4.3. Section 4.4 discusses an exemplary decision-graph from the fraud-detection domain. The management of decision-graph states at run time is discussed in Section 4.5.

### 4.2 Meta Model

Figure 4.2 shows the meta-model for decision graphs. A decision graph

$$g = (C, P, c, \Delta t)$$

is defined by a non-empty set of *rule components*  $C$ , a non-empty set of *precondition relationships*  $P$ , a *correlation set*  $c$ , as well as a *time window*  $\Delta t$ . The correlation set  $c$  and the time window  $\Delta t$  together form the decision graph’s *correlation configuration*, which is optional.

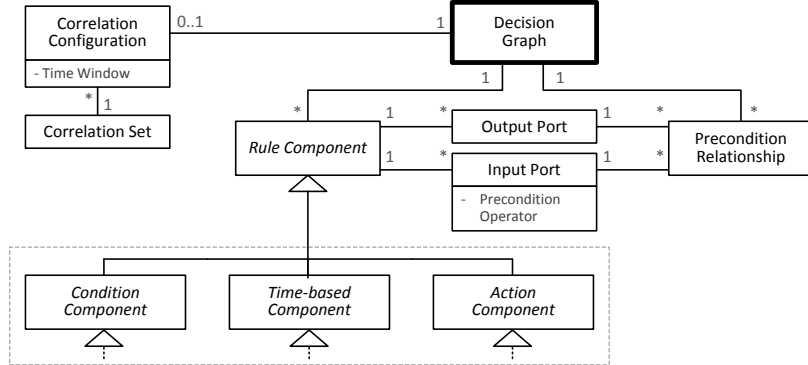


Figure 4.2. Decision Graph Meta-Model

**Rule components.** Encapsulating easy-to-understand pieces of pattern-detection or reaction logic, rule components form the nodes and basic elements of any decision graph. Depending on its implementation, each rule component  $c \in C$  has a collection of *input ports*  $IN$  and a collection of *output ports*  $OUT$ . While input ports allow generally activating a rule component, output ports represent possible results of the encapsulated logic. Dependencies between components are modeled as precondition relationships between output ports and input ports. To allow multiple preconditions, a binary *precondition operator*  $p_{in} \in \{\text{AND}, \text{OR}, \text{XOR}\}$  specifies whether all (AND), at least one (OR), or exactly one (XOR) precondition must be fulfilled in order to activate an input port  $in$ .

Rule components can generally be separated into *condition components*, *time-based components*, and *action components*. We discuss these categories in greater detail in Section 4.3 below.

**Precondition relationships.** Given the set of rule components, precondition relationships define under which conditions – i.e., under which state of the decision graph – a rule component is *activated*. Depending on the rule-component implementation, this activation may bring it to play in the pattern-detection process or cause it to trigger a described action. A precondition relationship  $p = (in, out)$  associates an output port  $out$  of a rule component  $c_i \in C$  with an input port  $in$  of another rule component  $c_j \in C$ . Cyclic dependencies are forbidden.

**Correlation set.** The proposed, model-driven approach to rule composition builds upon a strict decoupling of *event correlation* and *event-pattern detection*: While event correlation defines classes of event situations as a common level based on relationships between events, event-pattern detection defines for a given correlation set those characteristics of an event situation that make it noteworthy in a specific context. A decision graph’s correlation set consequently defines the class of event situations upon which a decision graph shall



be evaluated; given a correlation set  $c$ , the decision graph is evaluated separately for each correlation session of  $c$ . When no correlation set is defined, a decision graph is evaluated independently for each incoming event.

A more detailed discussion on the interplay between correlation sessions and decision-graph states is presented in Section 4.5 below.

**Time window.** Given a correlation set  $c$ , the time window  $\Delta t$  allows restricting the events of a correlation session that are considered for pattern detection at a certain point in time. Consider a decision graph that acts on the occurrence of an event  $e$ ; for performing calculation on the underlying event situation, a preceding event  $f$  of the correlation session  $s_{c,e} = \text{session}_c(e)$  is then considered only if  $\text{value}_e(\text{CreationTime}) - \text{value}_f(\text{CreationTime}) \leq \Delta t$ . By default,  $\Delta t = \infty$  is assumed.

### 4.3 Rule Components

Rule components are the basic elements of any decision graph and make the proposed model a flexible and extendible toolkit for defining rule-based event-processing logic. In its current version, SARI provides a predefined library of rule components for commonly required pattern-detection and reaction tasks. These rule components can be adjusted by the user to perfectly suit the given business needs. Depending on their general function in a decision graph, rule components can thereby be separated into

- condition components,
- time-based components, and
- action components.

In the following, we discuss these categories in greater detail and introduce the respective elements of SARI’s basic rule-component library. In the course of this thesis, additional rule components are introduced.

#### 4.3.1 Common Characteristics

Independent from their concrete implementation, rule components have access to – and, thus, may apply the encapsulated logic on – the concrete event situation that underlies the current evaluation step: *Event conditions* (Section 4.3.2), for example, evaluate Boolean functions against user-defined characteristics of event situations in order to decide whether to activate a “true” port or a “false” port. For a decision graph  $g = (C, P, c, \Delta t)$  and a most recent incoming event  $e$  – i.e., the event that has triggered the current evaluation step – the underlying event situation  $s_{g,e}$  is thereby defined as follows:

- for decision graphs that set up on a correlation set ( $c \neq \varepsilon$ ),  $s_{g,e}$  is constituted by those events of the correlation session  $s_{c,e} = \text{session}_c(e)$  of  $e$  according to  $c$  which lie in the specified time window  $\Delta t$ , i.e., with  $t_v$  indicating the time of occurrence of an event  $v$ ,

$$s_{g,e} = (f_1, f_2, \dots, f_n \mid f_i \in \text{session}_c(e), t_e - t_{f_i} \leq \Delta t)$$

- for so-called *stateless* decision graphs ( $c = \varepsilon$ ),  $s_{g,e} = (e)$ , i.e., the triggering event only. Stateless decision graphs are discussed in greater detail in Section 4.5.

Unless otherwise stated, we assume that user-definable properties of a rule component are declared as *Event Access (EA) Expressions* on the underlying event situation. Note, however, that user-defined expressions can easily be erroneous, e.g., due to divisions by zero or null-value errors. For all rule components that use user-defined expressions for accessing the underlying event situations, we therefore provide a Boolean “Ignore evaluation errors” property, indicating whether to simply ignore evaluation errors or to trigger a special *exception event*. Exception events are published via the respective rule service’s special *exception port* and can be handled just like any other event in downstream event-processing logic. Further, implementation-specific configurations are introduced below.

### 4.3.2 Condition Components

Condition components include all components that evaluate user-defined Boolean expressions on the underlying event situation and activate their output ports depending on the results of this evaluation. Condition components therewith allow modeling those characteristics of a class of event situations that make it *noteworthy* in a certain sense, and typically cover major parts of the pattern-detection side of a decision graph. For enhanced, time-related aspects, condition components must be combined with time-based components as described in Section 4.3.3 below.

**Input and output ports.** On their input side, all condition components feature a single *activator* port: Only if all required preconditions are fulfilled, the described pattern-detection logic is considered in the evaluation process. Otherwise, if the preconditions are not fulfilled, all output ports are de-activated and the component is ignored for event processing. On their output side, a set of output ports represent the possible results of an evaluation. In all of the following descriptions, we assume that a component is *active*, i.e., that all its preconditions are fulfilled.

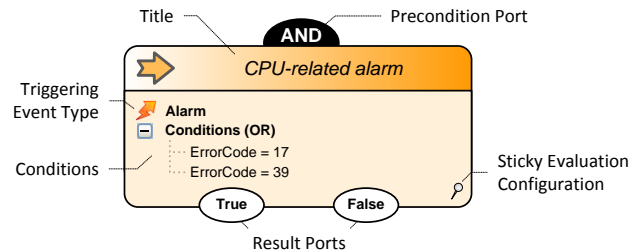
**Sticky evaluation.** In addition to implementation-specific configurations, all condition components provide a Boolean “Sticky evaluation” property, indicating whether or not activated output ports shall retain their activation across

subsequent evaluations of the encapsulated pattern-detection logic. Consider a condition component with two output ports “true” and “false”, with pattern-detection logic evaluating to *true* at a time stamp  $t_1$  and to *false* at a time stamp  $t_2$ , where  $t_1 < t_2$ . While the true port would in any case be activated at  $t_1$ , the further behavior depends on the chosen evaluation mode: When a component is defined to be sticky, the false port would be activated *in addition* to the previously-activated true port. Otherwise, when the sticky-evaluation flag is set to false, activation would switch from the true port to the false port, causing only the false port to be active after  $t_2$ .

### Event Conditions

An event condition  $c = (T, x_b)$  evaluates a Boolean expression  $x_b$  on the underlying event situation whenever an event of a user-defined, “triggering” event type  $T$  occurs in the incoming event stream. Depending on the result of this evaluation, a “true” output port or a “false” output port is activated. In many cases – although, not necessarily – the result of an event condition depends on the most recent event of an event situation, which is the triggering event itself. A so-defined event condition then describes the occurrence of a certain kind of event and may be considered as the core element of most decision graphs.

Figure 4.3 shows the shape rendering of an exemplary event condition “CPU-related alarm”, describing the occurrence of an event of type “Alarm” with an error code of 17 or 39.<sup>2</sup>



**Figure 4.3.** Event Condition Component

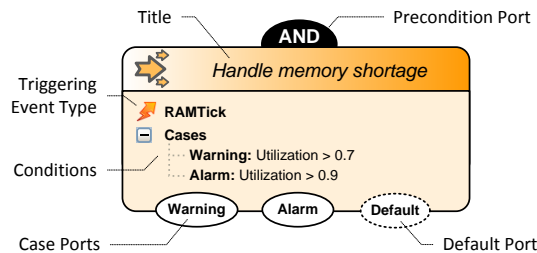
To ensure correct pattern detection also when several event conditions for a given “triggering” event type  $T$  are involved, event conditions are evaluated “bottom up”, i.e., for an incoming event  $e : T$ , a component is always evaluated prior to its predecessors in the decision graph. Otherwise, an (originally active) component could be deactivated by a predecessor before being evaluated itself.

<sup>2</sup> Note that in the presented example, the condition’s Boolean expression is defined as an EAExpression on a single event of the triggering event type  $T$ ; when this approach is used, the expression is always evaluated on the triggering event  $e : T$ .

## Event Cases

Event cases are similar to event conditions, however, they allow users to group sets of Boolean expressions in a single rule component. An event case  $c = (T, C)$  is defined by a triggering event type  $T$  and a collection of *cases*  $C$ , where each case  $c_i \in C$ ,  $c_i = (i, x_b)$  is defined by an identifier  $i$  and a Boolean expression  $x_b$ . When for an incoming event  $e : T$ , a case  $x_b$  evaluates to true, a corresponding output port  $out_{c_i}$  is activated. When all cases evaluate to false, an (optional) default port  $out_{default}$  is activated. As with event conditions, event cases are evaluated “bottom-up”.

Figure 4.4 shows an exemplary case “Handle memory shortage”, activating its output ports “Warning” and “Alarm” depending on the current memory utilization of a supervised device.



**Figure 4.4.** Event Case Component

### 4.3.3 Time-Based Components

Condition components as described in the above section always depend on a trigger to actually evaluate the encapsulated pattern-detection logic. With the set of rule components as described by now, such a trigger is – directly or indirectly – constituted by the occurrence of a concrete event, which may either trigger a component by itself or fulfill the component’s preconditions. In many real-world use cases, however, actions may depend on temporal conditions rather than on the immediate occurrence of event. Consider, for instance, typical use cases from the logistics domain, where actions shall be triggered in case that a transport is delayed for more than a specified time interval: An action should then not be executed with the eventual occurrence of the (delayed) “Transport Ended” event, but with the expiration of a use-case-specific time interval beginning with the “Transport Started” event, if the corresponding “Transport Ended” event does not occur within this time span.

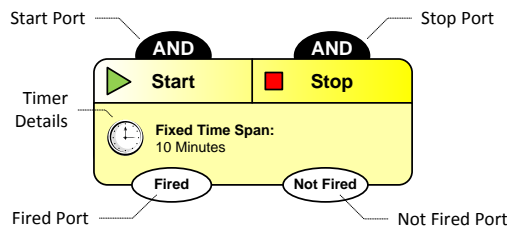
Time-based components complement the above-described, basic pattern-detection facilities by using internal clocks for activating their output ports.

As with condition components, all time-based components feature a Boolean “Sticky evaluation” property, indicating whether existing output-port activations shall be reset with new evaluations.

## Timers

Timer components have two precondition ports for starting and stopping an internal timer. When a running timer is stopped within a user-defined time span  $\Delta t$ , an output port “Not Fired” is activated; otherwise, after the expiration of  $\Delta t$ , an output port “Fired” is activated. Timer components are typically used to actively respond to the delayed occurrence of an event as illustrated in the above example.

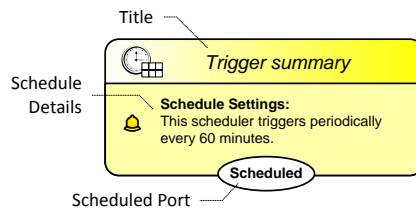
Figure 4.5 shows an exemplary timer with a constant time span of 10 minutes.



**Figure 4.5.** Timer Component

## Schedulers

Schedulers facilitate recurring activations of downstream rule components. Schedulers do not have an input port; an output port “Scheduled” triggers regularly at user-defined intervals  $\Delta t$ . Figure 4.6 shows an exemplary scheduler shape with a constant time interval of 60 minutes.



**Figure 4.6.** Scheduler Component

#### 4.3.4 Action Components

Action components encapsulate reaction logic to be executed whenever a component is activated, i.e., whenever an event situation conforms to a component's precondition. Unlike the pattern-detection part of a decision graph, such logic may be destructive, i.e., may affect the state of the SARI application and/or the underlying business environment. Typically serving as the end nodes of a decision graph, action components provide a single output port that simply loops through the activation state of the component's precondition port.

For all action components, SARI allows for the following base configuration:

- **Reset Rule State:** SARI's rule engine allows resetting the state of a decision graph for a given correlation set. When the "Reset Rule State" flag is set for an action component, the decision graph is reset automatically each time the component is activated, i.e., all output ports/dependencies are set to inactive.
- **Silence Interval:** Silence intervals enable application developers to prevent cascading actions in case of high-frequent activations of a respective action component: Beginning with the first successful execution of the encapsulated action, a so-configured component then suppresses all further executions for a user-defined time span.

#### Response Event Actions

Response-event components generate a response event based on a user-defined *response-event template*. The response-event template is defined by an event type  $T$ , and for each event attribute  $(i, t)$  in  $T$ , an expression on the underlying event situation returning a value of type  $t$ . On activate, these expressions are applied on the given event situation in order to calculate the concrete event-attribute values.<sup>3</sup> After finishing the current decision-graph evaluation step, the so-defined response event is published to the SARI application's processing model, where it may be subject to further event processing and/or cause concrete state changes or actions in the underlying business environment.

Figure 4.7 shows an exemplary response-event action component "Server crashed or shut down", which could, for example, be activated whenever the time between two events coming from a server is longer than expected. As a result of the component's activation, a response event of type "Server Down", with event attributes signifying the concerned server name and time stamp of the most recent "Resource Tick" event, is generated.

<sup>3</sup> The default attributes of an event – i.e., its time of occurrence and unique ID – are implicitly generated and need not to be defined.

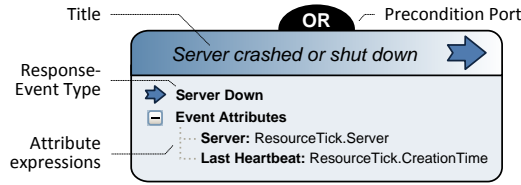


Figure 4.7. Response-Event Action Component

### 4.4 Example

Figure 4.8 shows the rendering of a simple decision graph from the fraud-detection domain, assembled from three *event conditions* and one *response event action*. In the example, a “Suspicious Bet Placed” event is generated whenever a high-stake bet (with a bet amount greater than 100\$) is followed by an external warning from at least one of two global early-warning systems. Via its event attributes, the response event carries the involved account ID, the bet ID, and a warning message. Note that correlation information – i.e., that all events must belong to the same sports event – is not defined in the decision graph directly, but would be modeled in a separate correlation set.

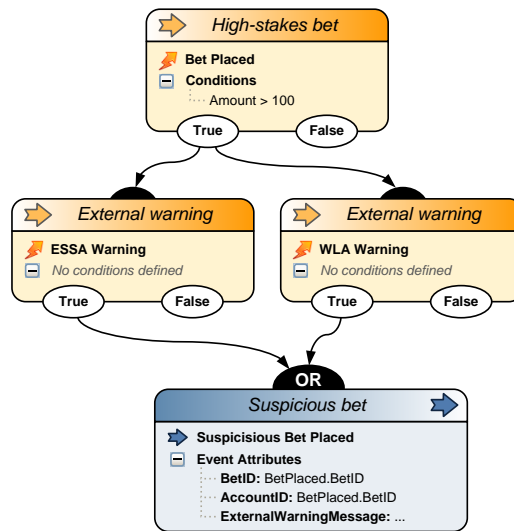


Figure 4.8. Exemplary Decision Graph

## 4.5 Managing Decision Graph State

As discussed in Section 4.1, SARI provides for a general separation between *event correlation* and *event-pattern detection*: Whereas event correlation groups events that are generally related to each other, event-pattern detection selects from these groupings those that are *noteworthy* in a certain context. In SARI's application model, this separation reflects in a decoupling between the *correlation model* as discussed in Section 3.4 and the *decision graph model*.

During run time, SARI implements the described separation by managing the state of a decision graph – i.e., the set of active dependencies between rule components as well as running timers and schedulers – separately for each correlation session of the decision graph's correlation set  $c$ . For an incoming event  $e$ , a rule service implicitly correlates  $e$  according to  $c$  as described in Section 3.4.4 and retrieves the specific correlation session  $s_{c,e} = \text{session}_c(e)$  in which  $e$  participates. Based upon this correlation session, the rule service retrieves the decision-graph state from a (rule-service level) state manager, or, if such state is not yet available, creates a new one. The decision-graph state then serves as the basis for processing  $e$  and is adapted depending on the described event-processing logic. After the processing of  $e$ , the adapted decision-graph state is updated at the state manager for subsequent event processing. As with correlation sessions, decision-graph states can be managed in memory or persistently in a database.

Figure 4.9 illustrates the management of decision-graph state by the example of a simple decision graph  $g = (C, P, c, \Delta t)$ . For the sake of simplicity, we assume that  $\Delta t = \infty$ .

An incoming event  $e_1$  is correlated based on  $c$  and establishes a new correlation session  $s_{c,e_1}$ . This new correlation session is not yet associated with a decision-graph state. As a consequence, a new state is created and attached to the correlation session.  $e_1$  is processed based on this initial decision-graph state and activates the left-hand output port of the decision graph's initial rule component  $r_1$ ; all other rule components are *inactive* for the given decision-graph state.  $e_2$  also establishes a new correlation session and associated decision-graph state. The third and final event  $e_3$  is correlated to  $e_1$ , meaning that  $e_3$  is added to the pre-existing correlation session  $s_{c,e_1}$ .  $e_3$  is then evaluated based on the decision graph state attached to  $s_{c,e_1}$ , in which the left-hand output for of the initial rule component has been activated based on the occurrence of  $e_1$ .  $e_3$  finally activates the output port of the – now *active* – rule component  $r_2$ .



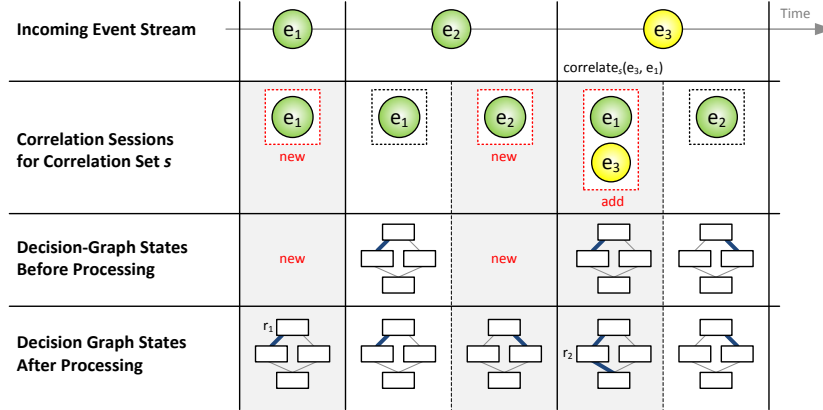


Figure 4.9. Handling Decision-Graph State at Run Time

#### 4.5.1 Merging Decision-Graph States

Whenever a correlation set is composed from more than one correlation band, merges between correlation sessions may occur. With decision-graph state being associated with correlation sessions in a 1-to-1 relationship, this inevitably leads to situations where two decision-graph states – one for each pre-merge correlation session – needs to be combined into one. For performance reasons, SARI follows what we call an *activation-based* merging approach, where output ports are activated in the merged decision-graph state if they are activated in *at least* one of the original decision-graph states. Running timers are adopted; if a timer is running for both decision-graph states, the earlier start time is chosen.

#### 4.5.2 Stateless Decision Graphs

Given a correlation configuration, decision graphs further characterize classes of event situations as defined in the correlation model. In certain cases, however, decisions can be taken on the level of individual events, independently from possible higher-level processes to which these events may relate.

In SARI, such scenarios can be modeled as decision graphs by simply omitting the correlation configuration: As the evaluation state of a decision graph is managed per correlation session, a so-defined decision graph is then evaluated independently for each incoming event; in other words, for each incoming event the decision graph is set to its initial evaluation state, with all output ports/dependencies inactive. We refer to these kinds of decision graphs as *stateless* in the remainder of this thesis.

Figure 4.10 shows a typical stateless decision graph from the fraud-detection domain. For each incoming event of type “Alarm” with an error code of 42, an “Email” event with respective attribute values is generated.

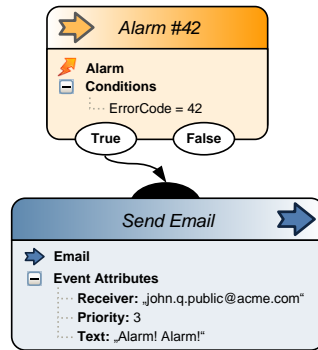


Figure 4.10. Exemplary Stateless Decision Graph



## A Framework for User-Oriented Rule Management

**Abstract** Event-pattern rules are “the foundation for successful applications of Complex Event Processing” [71] and find use on different conceptual layers of a event-based system. Today, when more and more people are involved in the setup and maintenance of event-based applications, it hence becomes increasingly important for companies that the different event-pattern rules of an application can be created, deployed and administrated by responsible and qualified personnel. This chapter presents a novel approach to user-oriented rule management for Sense-and-Respond Infrastructure (SARI). It caters to the needs of IT experts as well as business users, for which complementary, yet clearly decoupled workflows are presented. We present the conceptual foundations of our framework and present in great detail or approaches to *infrastructural rule management* and *sense-and-respond rule management*. We discuss user rights management for these approaches and illustrate the implementation of our framework as an extension to the base architecture of SARI. A particular focus is placed on extensions of the front-end layer of SARI, namely, an extended IDE for power users and a simplified web interface for business users.<sup>1</sup>

### 5.1 Introduction

Event-pattern rules of the form

**if** an event pattern  $p$  is detected, **then** execute action(s)  $A$

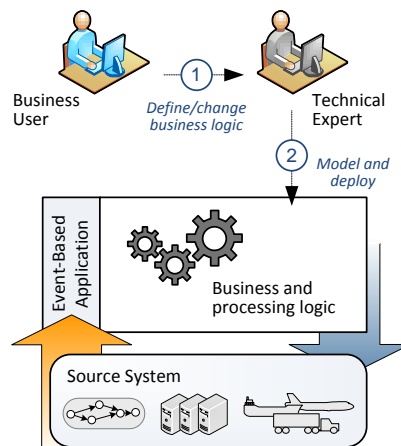
are a key element of Complex Event Processing (CEP) and proved suitable for describing high-level business logic as well as low-level (pre-)processing and integration steps. Depending on their function, event-pattern rules are typically associated with different user groups of an enterprise: While processing logic will typically be managed by IT experts, business logic will typically be in the responsibility of technically inexperienced domain experts. An approach to *rule management* for Complex Event Processing systems – i.e., the overall

---

<sup>1</sup> This chapter is based on the work of Obwegger et al. [90] and Kavelar et al. [62].

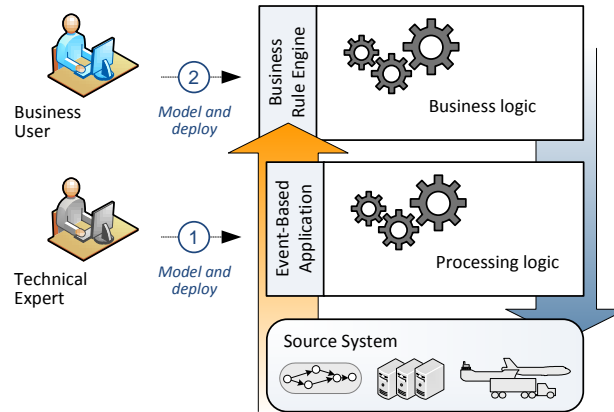
set of tools and workflows provided for the creation, application, and administration of event-pattern rules – must therefore cater to the needs of IT experts as well as business users. IT experts must be provided with facilities to define event-pattern rules in an effective manner with little administrative overhead. Business users, by contrast, must be provided with facilities to define rules in a non-technical, business-oriented manner. Following from these highly divergent and partly opposing requirements, approaches that focus on only one group of users will inevitably fail to support the work of the other.

In cases where a rule-management system caters to the needs of IT experts but neglects the needs of business users, business users are either required to have extensive technical skills or rely on technical experts to implement changes in the business logic. The former option may appear favorable at first sight; it is, however, infeasible in practice and will raise security concerns if a CEP framework gains unrestricted access to an event-based application. The latter option is generally practicable, however, imposes significant overhead on both the IT department – for implementing changes in the business logic as requested by business users in a timely manner – and business users, for documenting and communicating the requested business logic in a form that is clear and complete enough to be usable by the IT department. Figure 5.1 sketches the described workflow.



**Figure 5.1.** Rule Management in Power-User-Oriented CEP Systems

In cases where a rule-management system caters to the needs of business users but neglects the needs of IT experts, IT experts are required to cope with restrictions and/or unnecessary abstractions for implementing low-level processing logic. If required functionality is not available at all, a separate integration layer must be introduced for pre-processing issues. Such approach has been proposed for the integration of CEP and business rule engines (BREs) [14];



**Figure 5.2.** Rule Management in CEP/BRE Architectures

however, besides resulting in additional efforts for purchasing, maintaining, and getting acquainted with two separate products, it neglects the importance of temporal aspects for high-level business logic. These are not well supported by state-of-the-art BREs. Figure 5.2 sketches the described workflow.

In this chapter, we present a novel approach to rule management for Sense-and-Respond Infrastructure. It caters to the needs of IT experts as well as business users, for which complementary, yet clearly decoupled workflows are presented. IT experts define event-pattern rules in a single, comprehensive model, in parallel and fully integrated with the other elements of an application’s event-processing infrastructure. Business users assemble event-pattern rules from prepared, configurable building blocks of pattern-detection and reaction logic in a simplified, wizard-based interface. The resulting change in the rule management workflow is depicted in Figure 5.3.

## Outlook

The remainder of this chapter is structured as follows: Section 5.2 the conceptual foundations of our approach. In Section 5.3, related work is discussed. In Section 5.4 and Section 5.5, we present our approaches to *infrastructural rule management* and *sense-and-respond rule management* in greater detail. User rights management for these approaches is discussed in Section 5.6. Section 5.7 provides an overview of the implementation architecture of our approach, which extends SARI’s base architecture as presented in Section 3.6 across all layers of the system. The two major extensions on the front-end layer of the architecture – namely, an extended version of the Modeling Studio and a novel, web-based rule-management tool for business users – are discussed in Section 5.8 and Section 5.9, respectively.

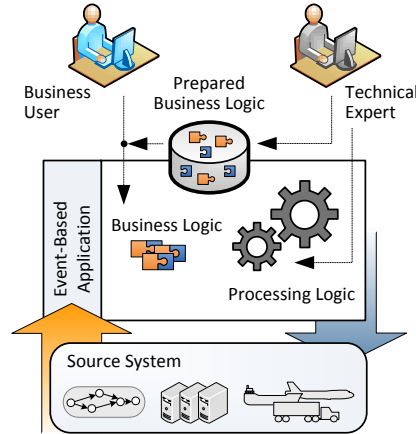


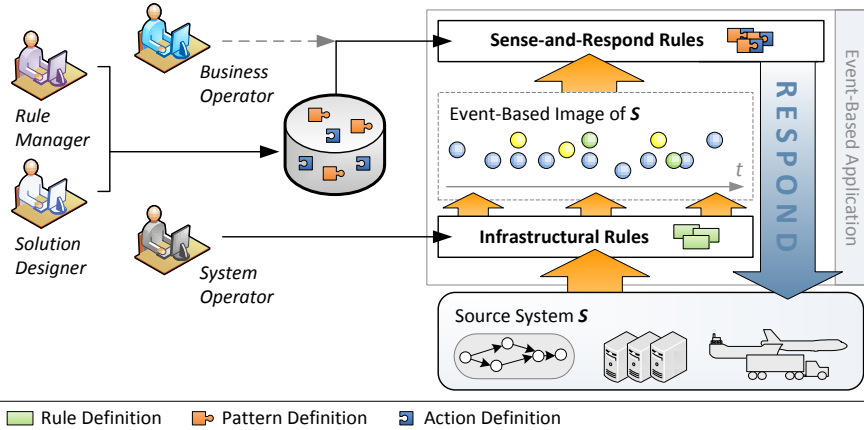
Figure 5.3. Rule Management in SARI

## 5.2 Conceptual Foundations

The presented approach to user-oriented rule management is rooted in a conceptual differentiation of event-pattern rules by their general function within an event-based application, into *infrastructural rules* and *sense-and-respond rules*. Both kinds of rules are fully equivalent regarding their basic semantics: In either case, actions are triggered in response to detections of event patterns in an underlying stream of events. They differ, however, in the way they are created, applied, and administrated in the proposed rule-management system. Figure 5.4 illustrates the roles of infrastructural rules and sense-and-respond rules within an event-based application.

**Infrastructural rules**, on the one hand, include all rules that provide input for other parts of an event-based application, but not by themselves respond to the underlying source system. Along with the event model, correlation model, and event processing model, the collection of infrastructural rules of an application may therefore be considered the *event-processing infrastructure* for creating an event-based, near real-time image of the underlying business environment: All relevant real-world actions and state changes are then accessible at a proper level of granularity, via accordingly pre-processed events.

As part of a so-defined integration layer between the real-world business environment and high-level decision making, effective and properly orchestrated infrastructural rules are obviously critical to the general functioning and performance of an event-based application. In the proposed framework, infrastructural rules are therefore managed by well-trained power users of a system – so-called *system operators* – in parallel and fully integrated with the other elements of an event-processing infrastructure. System operators model infrastructural rules in a single, comprehensive model, as so-called *rule definitions*.



**Figure 5.4.** Infrastructural Rules vs. Sense-and-Respond Rules

These rule definitions are immediately interpretable to SARI and can be deployed by directly assigning them to one or more of the application’s rule services. The proposed workflow especially focuses on *efficiency*, *transparency*, and *immediacy*, and strives to minimize administrative overhead as would emerge from a more abstracted approach.

**Sense-and-respond rules**, on the other hand, include all rules that do not serve as an input for other parts of an event-based application, but directly or indirectly respond to the underlying business environment. Setting up on an up-and-running *event-processing infrastructure* (including the infrastructural rules of an application), sense-and-respond rules continuously monitoring the provided event-based image for relevant business situations and trigger respective actions in response.

Implementing the high-level business logic of an event-based application, sense-and-respond rules are critical to the proper monitoring and controlling of a business environment. Still, creating and applying such rules “from scratch” would not only require the domain-specific expertise of customer-side representatives, but would also force these users to get acquainted with the diverse mechanisms for creating and applying event-pattern rules as part of an event-based application. We therefore propose a three-step workflow for sense-and-respond rule management:

In the first step of the workflow, well-trained power users of the employed CEP platform – so-called *solution designers* – define a catalog of configurable building blocks of (i) encapsulated pattern-detection logic (*pattern definitions*), and (ii) encapsulated reaction logic (*action definitions*) based on the general monitoring and steering requirements of the given business scenario. With the focus on *reusability* across different use cases and *information hiding*, these *event-level building blocks* expose a collection of named, typed, and documented in-



put parameters, which allows them to be configured without having to change and/or understand the low-level event-processing logic.

In the second step, senior domain experts – so-called *rule managers* – refine the prepared building blocks with respect to the concrete application scenarios in which they are to be used. Focusing on *easy of use*, rule managers simplify the instantiation of a building block by further specifying the encapsulated event-processing logic (through setting or restricting the domain of input parameters) and providing for each so-defined refinement a high-level, textual representation with placeholders for all unset input parameters. The resulting *business-level building blocks* fully abstract from underlying complexity: From an end-user point of view, the prepared catalog of event-processing logic appears as a collection of relevant business situations (e.g., “suspicious user behavior”) and possible actions (e.g., “notify fraud-prevention department”).

In the third and final step, appropriately configured buildings blocks are assembled to concrete sense-and-respond rules of the form “if pattern, then action(s)” depending on the current requirements of an enterprise. The creation and maintenance of the concrete event-pattern rules still requires domain-specific, detailed knowledge of the source system; however, it fully abstracts from the event-base foundations of an application. Also, due to sense-and-respond rules’ “read only” access to the underlying event stream of real-world actions and state changes, they may be added, changed, and removed without having to consider any side-effects to other parts of the application. The process of instantiating concrete sense-and-respond logic from prepared building blocks can therefore be performed by so-called *business operators*, which usually are domain experts with little or no technical expertise.

Covering both processing logic and business logic, the overall process of creating a full-fledged, up-and-running event-based application can be summarized as follows:

- **System operators** establish an event-processing infrastructure, including all infrastructural rules of an application. This event-processing infrastructure establishes a near real-time, event-based image of the underlying business environment, including event representations for all relevant actions and state changes at a proper level of granularity.
- **Solution designers** create a catalog of *event-level building blocks* of encapsulated pattern-detection logic and reaction logic based on the general monitoring requirements of the given business scenario.
- **Rule managers** refine the prepared building blocks to so-called *business-level building blocks*, by further specifying the encapsulated event-processing logic and providing a high-level, textual representation.
- **Business operators** assemble these business-level building blocks to concrete sense-and-respond rule logic.

## 5.3 Related Work

In the following, we present existing work related to rule management in Complex Event Processing systems. Beginning with work on the particular issue of rule management in Section 5.3.1, we address the current discussion on CEP for business users in Section 5.3.2 and show existing applications of layered event-processing models in Section 5.3.3. We discuss existing differentiations between event-pattern rules in Section 5.3.4 and elaborate on possible similarities and differences between the proposed approach and Business Rule Management Systems in Section 5.3.5.

### 5.3.1 Rule Management for Event-Based Systems

For many years, CEP-related research has primarily focused on technical qualities such as expressiveness and performance. While some efforts have been devoted to the development of easy-to-use EPLs and less programming-centered approaches to rule creation, there is little work on the broader issue of rule management in event-based systems.

Luckham [72] sees rule management as one of the challenges for future Complex Event Processing. He defines rules management as follows:

“Rules management is about (1) writing correct rules – that say what you mean, (2) organizing rule sets for efficient execution, so the rules engine tries only the rules that might apply at any time, (3) making changes correctly, which involves knowing how a new rule will interact with existing rules, and of course (4) ensuring logical consistency and absence of redundancies.”

Our approach to rule management addresses point 1 of the definition – writing correct rules – by providing rule-authoring facilities tailored to the different user groups in a company. Point 2 – organizing rule sets for efficient execution – is addressed through *rule spaces*, which group end-user defined rules and map them to rule services so that only relevant events are considered for rule evaluation. Point 3 – making changes correctly, in a way that is aware of other rules and possible side effects – is supported by the proposed workflows for both power users and business users: Whereas power users have full access to a system and the various infrastructural rules therein, sense-and-respond rules set up on the event-based image of a business environment in a read-only manner and thus can be created, changed, and removed without side-effects. Point 4 – ensuring logical consistency and absence of redundancies – is not currently addressed in the presented approach, but is subject to future work.

Sen and Stojanovic present GRUve [118], a four-phase methodology for managing complex event patterns – so-called CEPATs – throughout their life cycle.

It is based upon the idea to “enable non-technicians to search incrementally for the most suitable form of requested CEPATs and to continually improve their quality taking into account changes that might happen in the internal or external world”. The goal of the initial *generation* phase of the methodology is to create a first version of the requested event pattern and to represent it in an RDFS-based format.<sup>2</sup> The authors list several approaches to event-pattern creation, namely creation from scratch (for power users), querying an event-pattern repository, as well as data mining on system data such as log files. The RDFS representation is then used in the *refinement* phase, where based on semantic relationships between the event pattern at hand and existing event patterns, similar event patterns are presented to the user. In the *usage* phase, the event pattern is deployed in the CEP framework. It is continuously monitored and statistics such as the number of matches per time unit are collected. In the final *evolution* phase, improvements to the evaluated CEPATs are suggested based on the usage statistics. The authors present a web application that implements the generation and the refinement phase; usage and evolution are left for future work. We agree with the authors that reuse of event-processing logic is of paramount importance for a user-friendly rule or event-pattern management framework. We believe, however, that a common workflow for power users and business users will in almost any case fail to support the work of at least one group of users. Our framework supports the reuse of event-processing logic through the separation of abstracted “building blocks” and instantiations thereof. Reuse on the level of abstract pattern definitions will be discussed in great detail in Chapter 7.

### 5.3.2 Complex Event Processing for Business Users

While little work addresses the general issue of rule-management in event-based systems, there is an active discussion on how to make (complex) event processing accessible to business users with restricted technical skills. The proposed rule-management system defines a workflow tailored to the needs of business users. Equally important, a parallel workflow enables IT experts to operate with minimal administrative overhead.

Event processing for business users has been discussed prominently in recent academically-oriented monographs on event processing, by Etzion and Niblett [37] and by Chandy and Schulte [27]. Etzion and Niblett list the development from programming-centered to semi-technical development tools as one of the emerging directions in event processing. They see “an increasing trend towards allowing business users and business analytics, who might not

---

<sup>2</sup> RDF Schema [139] is an extensible knowledge representation language that allows defining ontologies. Represented in an RDFS-based format, relationships such as *disjoint from*, *sub/superset of*, or *instance of* can be evaluated on pairs of event patterns.

have deep programming skills, to compose all or part of an event processing application”, through higher levels of abstraction. The authors identify IBM WebSphere Business Events [56] as an early example of this trend. Chandy and Schulte identify the ability to “enable business users to tailor systems to their needs” as a major criterion for the relevance of an event-based system and claim that a one-size-fits-all specification of events and responses doesn’t work. On the contrary, the authors point towards increasing efforts of such an approach, remarking that several roles in a company must – to some extent – be trained to the event-processing system:

“IT staff in the enterprise learn event specification notations provided by vendors and set up business-oriented templates for end users; power business users create their own macros; and, finally, each business user spends time learning how to use tools to tailor the system to that user’s individual needs.”

Note that the discussed assignment of responsibilities to the various user groups within an enterprise largely conforms to the architecture proposed in Section 5.2. Chandy and Schulte furthermore point out the relevance of systematic event(-pattern) management for successful implementations of event processing. They remark, however, that such features are not yet available in any commercial registry or repository. Through a repository of building blocks, describing noteworthy event situations and possible reactions in an abstract manner, our approach introduces pattern-management capabilities to SARI.

Event-processing frameworks suitable to business users have furthermore been requested by actual and potential adopters of CEP. ebizQ [33] presents the results of an online customer survey on event processing in companies, indicating that 84% businesses would like to have event rules defined by “business specialists” or “business analysts”, and only 16% want to have them defined by IT developers.

Mismatches between the complexity of event-processing logic on the one side and the abilities of potential adopters on the other side have been highlighted by von Ammon et al. [8], which consider the resulting problems a major reason why future CEP applications will “delay to be set up”. The authors suggest the definition of domain-specific reference models for event processing, thereby orienting themselves on the theory of design patterns [5] as well known, for instance, from software engineering (e.g., [20, 41, 116]). Design patterns for event processing have been discussed in academic contexts [12, 24, 97] as well as in the industry [30]. We fully agree that a comprehensive and widely accepted pattern language for CEP would be a significant step in the development of the discipline. We believe, however, that design patterns will primarily address technically versed users, and that business users are not willing and/or capable to work through collections of design patterns and transfer them to the concrete event-processing software at hand. Design patterns may therefore be an important source of knowledge for *system operators* and *solution designers*

at the bottom layers of our architecture; business operators, by contrast, are not concerned with the “how” of event processing but instead may focus on the “what” only.

Turchin et al. [129] claim that independent from the definition of concrete event-processing logic in a CEP framework, domain experts will often fail to specify the exact parameters and thresholds required to optimally monitor a system: “While it is reasonable to expect that domain experts will be able to provide a partial rules specification, providing all the required details is a hard task, even for domain experts”. The authors propose an approach to *rule parameter prediction and correction*, where the rule parameters are initialized and/or adapted dynamically based on statistical methods. While such an approach is reasonable in domains such as intrusion detection, we believe that companies will typically demand highest transparency about the rules at play. In our system, any changes in the event-processing logic are therefore induced by explicit user actions and logged in so-called *rule histories*.

### 5.3.3 Layered Event-Processing Models

The presented approach to rule management builds upon a differentiation of event-pattern rules into infrastructural rules and sense-and-respond rules. Conceptually, this separation results in a two-layered architecture for event-processing applications. While seldom associated with different workflows and user groups, layered application models have a long tradition in CEP.

In his widely cited work on “The Power of Events” [71], Luckham presents a layered architecture for event-processing networks. At the adapter layer, external events are transformed into a format that is interpretable to the CEP framework at hand. At the filtering layer, irrelevant events are eliminated from further processing. At the map layer, aggregation rules and situation-detection rules implement the concrete event-processing logic. Via the adapter layer, response events are routed back to external systems.

Paschke and Vincent [99] present a general reference architecture for event processing. They refer as *event processing medium* to a platform that, on the bottom end, receives low-level “atomic events” from event sources, processes these events stepwise and, on the upper end, sends high-level “business events” to event consumers. The authors separate the event processing medium into event selection, event aggregation, and event handling. Event selection and event aggregation largely conform to the processing tasks of *filtering* and *aggregation* as described in Section 1.2. Event handling includes the detection of noteworthy situations, the rating of events within the contexts in which they occur, and event prediction. Thus, it may be considered as the business logic of an application. According to the purpose of a high-level reference architecture, the authors do not go into further detail on how these phases shall be

implemented and how respective event-processing logic shall be defined and administrated.

In contrast to the above architectures we do not draw a distinction between filtering and transformation from an architectural point of view. Instead, *infrastructural rules* include all rules that produce input for other parts of an event-based application. While aggregation and situation-detection issues are consolidated into a common layer by Luckham, we consider situation detection as a separate layer in accordance to Paschke and Vincent.

In their extensive work on stream-data processing, Chakravarthy and Jiang [26] present *MavEStream*, a layered architecture that allows integrating continuous query processing and Complex Event Processing. At the bottom layer, a stream-processing system is responsible for low-level data aggregations and filtering. At the second layer, events are generated. At the third layer, CEP is used for detecting complex events from raw events as emerging from the second layer. At the fourth and final layer, actions are associated with triggering events in event-pattern rules. The proposed separation into infrastructural rules and sense-and-respond rules is comparable to Stage 3 and Stage 4 of MavEStream, which address the CEP part of the architecture.

Kellner and Fiege [63] present the separation of two viewpoints in a CEP application, which facilitates a business-oriented, top-down approach to event processing based on the concept of Key Performance Indicators (KPIs). In the *derivation* viewpoint, KPIs (e.g., the number of items produced in the last hour) are derived from lower-level events (e.g., “Item Produced”) through filtering and aggregation rules. In the *interpretation* viewpoint, target values are specified for the various KPIs and relevant business situations are defined to be reported when the actual values fall below respective thresholds (e.g., the opportunity for maintenance is detected when a utilization rate drops below 50%). The authors claim that “changes in situations to be detected can be handled without affecting the derivation of values for KPIs that are more stable”. Similarly, we argue that infrastructural rules are more stable while sense-and-respond rules enable quick adaptations of the business logic. While there is no explicit notion of KPIs in basic SARI as discussed in Chapter 3, we introduce respective mechanisms in Chapter 6 and align them with our approach to rule management.

#### 5.3.4 Differentiating Rules in Event Processing

Bry and Eckert present XChange<sup>EQ</sup> [16, 17, 34], a high-level EPL that builds upon a differentiation of event-pattern rules into *deductive rules* and *reactive rules* depending on their function in an event-based application [34]:

“Deductive Rules define new events based on event queries; they are comparable with views in databases and have no side-effects. We em-

phasize that these deductive rules operate on events, not on facts (like ‘traditional’ deductive rules from logic programming and deductive databases). Reactive rules specify how to react to (complex) events, e.g., with database updates or procedure calls.”

The presented distinction is primarily a conceptual one since both kinds of rules conform to the same, Event/Condition/Action-based style of event processing – “While it is possible to ‘abuse’ reactive rules to simulate or implement deductive rules [...], this is undesirable” [16]. Yet, in contrast to most other EPLs and CEP frameworks, XChange<sup>EQ</sup> prescribes different (although very similar) syntaxes for the different kinds of rules.

The notion of deductive rules and reactive rules is comparable to the notion of infrastructural rules and sense-and-respond rules, respectively. It is, however, necessary to take a closer look at the issue of side effects. While the authors consider (lower-level) deductive rules as free from side effects, we state this for (higher-level) sense-and-respond rules. This difference is apparently rooted in different understandings of side effects: Whereas Eckert and Bry locate side effects outside the actual event-processing environment, side effects in our understanding concern the event-processing environment itself and therefore are given if – and only if – a rule produces input for other parts of an application.

### 5.3.5 Business Rule Management Systems

In the field of business rules, rule management has long been recognized as an important component for practical deployments. Business Rule Management (BRM; e.g., [51, 81, 102]), which complements business rule engines with rule repositories and rule authorizing tools, has developed into an industry and research field in its own right. Similar to our work, Business Rule Management Systems (BRMSs) especially aim to provide tailored workflows for the different user groups in a company. In the following, we give a brief introduction to the most prominent concepts in BRM<sup>3</sup> and discuss how they relate to CEP in common and our work in specific.

The rule repository is typically considered the key element of a BRMS. Features provided by modern rule repositories include versioning, role-based permissions, collaboration among remote users and synchronization of concurrent updates, hot rule deployment (i.e., the possibility to add, change and remove rules without having to stop the system) as well as scheduled rule activation and deactivation. Full-fledged rule repositories are seldom provided by CEP frameworks, which may be owed to the relative youth of the discipline and the focus on operational issues in research so far. In the proposed rule-management

---

<sup>3</sup> This introduction is largely based on Graham’s review of the most prominent commercial BRMSs [47]. Further information is available at the various companies’ websites and in respective secondary literature [11].

framework, we opted against a dedicated repository for infrastructural rules. Infrastructural rules are instead considered an integral part of an application description, which is subject to a repository at the SARI architecture’s administration database (see Section 3.6.1) as a whole. A rule repository is, however, provided for sense-and-respond rules, which are changed on a regular basis by multiple – often remote – users. Our repository supports role-based permissions, collaboration among remote users, hot deployment, as well as rule scheduling. Versioning and synchronization are provided in basic forms, with further improvements planned for future work.

In addition to a rule repository, BRMSs provide rule-authoring facilities for both technically versed users and business users. Most BRMSs aim to provide a rule syntax that is close to natural language, often in addition to a more technical syntax for developers. Prominent examples are Blaze Advisor’s Structured Rule Language [136] or ILOG JRules’ Business Authoring Language [57]. While some of these languages are restricted to syntactic sugar (such as “client’s age is 17” instead of “client.age = 17”), others allow defining “verbalizations” (in terms of ILOG JRules) of the underlying object model to achieve a more natural look and feel. Domain Specific Languages ([32]) are an extension to natural-looking, yet general purpose rule languages. Here, qualified users may define a language that is particularly suitable to the given application domain. Prominent examples are JRules’ Business Rule Language Definition Framework (BRLDF) and the domain-specific language support in JBoss Drools [58]. Haley Expert Rules [50] allows its users to start with rules defined in a custom – yet, English-grammar based – language and derive a respective object model based on these rules. To our best knowledge, there is no EPL that seriously claims to be close to natural language; neither have DSLs in the described sense<sup>4</sup> been used in event processing. We believe that this is because of the increased complexity of event-pattern rules in comparison to business rules.

Rule templates are partially defined rules with placeholders that can be populated later on by a business user. While in some languages the original syntax is presented to the end user, others allow to define custom text (or HTML) to be shown with the placeholders – see Blaze Advisor’s Rule Maintenance Applications for a prominent example. Allowing users to configure and instantiate arbitrarily complex rule logic without requiring deeper knowledge of the underlying rule language, rule templates are well-suited for CEP. Rule templates have, for instance, been used with AMiT (cf. [3]). The proposed rule-management workflow for business users builds upon a concept that is closely related to rule templates. However, template-like structures are pro-

---

<sup>4</sup> As most EPLs provide good support for accessing event attributes and temporal reasoning, EPLs are often considered as “domain specific” with respect to event processing in general (e.g. [137]). In the course of this discussion, we use the term DSL for languages that are specific to a certain application domain.



vided on the level of patterns and actions instead of complete rules (although templates for complete rules are presented as an extended concept). Also, we separate the logic part of a template from the presentation part, which enables reuse of rule logic across different contexts and multi-language support. Finally, in contrast to the vast majority of BRMSs, we consider templates as self-contained, identifiable entities. This enables us to maintain a connection between concrete rules and the building blocks from which they were created, which in turn leads to a consistent and up-to-date rule base.

Decision tables allow users to define business policies in tabular form, associating pattern parameters (such as the age of a customer) with action parameters (such as the resulting insurance rating). Decision tables are available with selected CEP frameworks such as TIBCO's BusinessEvents [128].

A recent development in BRM is the integration of BRMSs with office suites. Several frameworks allow decision tables to be defined in spreadsheet software such as Microsoft Excel. ILOG JRules also integrates with Microsoft Word, which enables business users to write rules as part of normal text documents. We are not aware of any comparable approach in CEP.

## 5.4 Infrastructural Rule Management

In Section 5.1 we have introduced the notion of infrastructural rules for all rules that prepare data for other parts of an application, but do not by themselves respond to the source system. Along with event types, correlation sets, and event processing maps, infrastructural rules form the event-processing infrastructure of an application and contribute to a near real-time, event-based image of the underlying business environment.

In the following, we discuss the proposed approach to infrastructural rule management. It is based on the idea of letting power users of an event-processing framework – so-called *system operators* – define infrastructural event-processing logic in a single, comprehensive model, and directly apply it at respective event services.

### 5.4.1 Requirements

From the basic differentiation into infrastructural rules and sense-and-respond rules we derived the following requirements for an approach to infrastructural rule management:

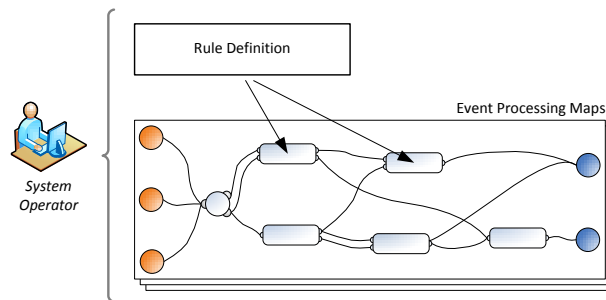
- **Expressiveness:** Rule-based event processing has proved useful for a wide range of infrastructural issues, including the filtering of irrelevant and erroneous data, event-data enrichment, and the aggregation of low-level events

to higher-level composite events. Infrastructural rules must therefore be expressive enough to enable application developers to establish an appropriate event-based image of the given source system.

- **Efficiency of use:** Infrastructural rules are created and applied by power users with a deep understanding of the given event-processing framework. To facilitate the efficient creation of an event-processing infrastructure, an approach to rule management should therefore make the creation and deployment of infrastructural rules as *immediate*, *clear* and *transparent* as possible. These aspects should be emphasized in preference to concepts such as decoupling of pattern-detection logic and reaction logic or a however-defined abstraction from underlying complexity. Infrastructural rule management should furthermore minimize any overhead that may arise from administrating rules and their assignment to rule services.
- **Full and system-wide access:** Being part of the event-processing infrastructure of an application, adding, changing, or removing infrastructural rules may affect calculations in diverse parts of an event processing map. An approach to infrastructural rule management should therefore provide full access to the application, enabling users to investigate and handle possible side effects.

#### 5.4.2 Model Overview

Figure 5.5 provides an overview of the proposed approach to infrastructural rule management.



**Figure 5.5.** Overview of Infrastructural Rule Management

Given the specific requirements for the event-processing infrastructure of an application, power users of a CEP framework – so-called *system operators* – model infrastructural event-processing logic as application-wide *rule definitions* in parallel and fully integrated with the other elements of an application. Rule definitions encapsulate event-processing logic of the form “if pattern, then action(s)” in a single, integrated model that is directly interpretable to SARI. As a consequence, rule definitions can be applied to incoming event streams without further instantiation steps.

For their enactment as part of a SARI application, system operators assign rule definitions to one or more *rule services* across the event processing maps of an application. During run time, each rule service then evaluates an independent instance of the encapsulated event-processing logic on the incoming event stream and publishes possible response events to downstream event-processing units. As both event-processing maps and rule definitions are defined and administrated by system operators, rule services can be associated with appropriate pre- and post-processing steps according to the hosted set of rule definitions.

### 5.4.3 Rule Definitions

In the proposed architecture, rule definitions allow modeling infrastructural event-processing logic such that it can directly and without further instantiation steps be applied as part of an event-processing infrastructure. Rule definitions describe such logic in the form of *reactive decision graphs*, decision graphs which, by themselves, cover the execution of reaction logic in response to the detection of noteworthy event situations. The decision graph of a rule definition is thereby assembled from

- a non-empty collection of *condition components* and/or *time-based components* as described in Section 4.3.2 and Section 4.3.3, and
- a non-empty collection of *action components* as described in Section 4.3.4. While basic SARI as presented in Section 3 and Section 4 features only one kind of action component – namely, *response action components* – further components are introduced in Chapter 6.<sup>5</sup>

A so-defined rule definition clearly implies a tight coupling of pattern-detection logic and response logic, preventing users from reusing event-processing logic in other contexts. Note, however, that infrastructural rules typically encompass highly purpose-specific logic that is required exactly once in a system and makes sense only “as is”. Decoupling the pattern-detection part and the

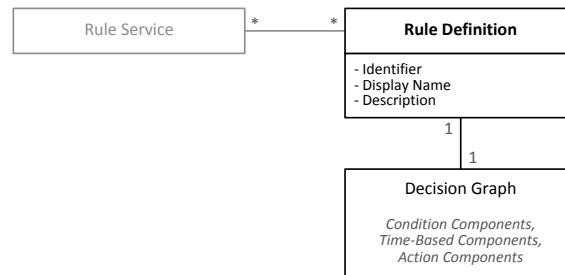
---

<sup>5</sup> *Signals* as are used in sense-and-respond rule management are special action components and are not available with rule definitions.

reaction part of an infrastructural rule would therefore cause considerable effort not only for the actual decoupling, but also for the administration of the so-created sub-entities, for little or no gain. Also, using a single decision graph provides a comprehensive view on a rule. This simplifies the creation of complex rule logic as well as bug-fixing.

### Meta Model

Figure 5.6 shows the meta model for rule definitions. A rule definition  $r = (i, n, d, g)$  is defined by an application-wide identifier  $i$ , a display name  $n$ , a human-readable description  $d$ , and a reactive decision graph  $d$ . Rule definitions are associated with rule services in an  $n$ -to- $m$  relationship.



**Figure 5.6.** Rule Definition Meta-Model

### Example

Figure 5.7 shows an exemplary rule definition “Short-term betting transaction” from the fraud-detection domain. Deployed on a rule service, it generates an event of type “Short-term betting transaction” whenever a user pays into a (near-) empty account, places and wins a single high-risk bet, and immediately pays off in full. For both pay-in and cash-out, an impreciseness of 5% is considered. A so-defined user behavior – especially when observed repeatedly – is highly suspicious and may indicate fraud.

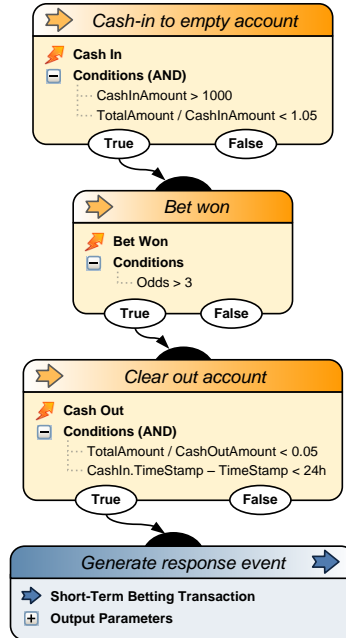


Figure 5.7. Exemplary Rule Definition

## 5.5 Sense-and-Respond Rule Management

In Section 5.1 we have introduced the notion of sense-and-respond rules for all rules that do not prepare data for other parts of an application, but directly or indirectly respond to the source system. Setting up on an up-and-running event-processing infrastructure, sense-and-respond rules continuously monitor the event-based image of a business environment for relevant business situations and trigger respective actions in response.

In this section, we discuss the proposed approach to sense-and-respond rule management. It is based on the idea of (i) technically versed *solution designers* and senior domain experts – so-called *rule managers* – preparing building blocks of pattern-detection logic and reaction logic, and (ii) *business operators* assembling these building blocks to concrete rule logic of the form “if pattern, then action(s)”.

The remainder of this section is structured as follows: In Section 5.5.1, we summarize the requirements for a sense-and-respond rule-management system. In Section 5.5.2, we present an overview of our approach. The various artifacts of this approach, namely *pattern definitions*, *action definitions*, *business patterns*, *business actions*, *sense-and-respond rules*, and *rule spaces*, are discussed in Section 5.5.3 to Section 5.5.8.

In the following, unless otherwise stated, we assume that all presented entities, as well as their various sub-entities, have a unique identifier, a display name, and an optional, human-readable description. For concrete examples, we resort to a real-world application for *event-based service assurance* in the workload automation domain, where SARI is employed as an add-on to the *UC4 Automation Engine* [130] with the goal of detecting fault patterns and ensuring the reliable, high-performing operation of a system landscape. The presented application is discussed in greater detail in Chapter 8.

### 5.5.1 Requirements

From the basic separation into infrastructural rules and sense-and-respond rules and the idea of easy-to-use building blocks of event-processing logic, we derived the following requirements for an approach to sense-and-respond rule management:

- **Decoupling** of pattern-detection logic and reaction logic: Allowing business users to assemble rules from building blocks of pattern-detection and reaction logic first of all requires having these parts available as separate entities, in a way that allows combining and integrating them in an arbitrary manner.
- **Reusability** of building blocks across different application scenarios, i.e., *configurability*: While building blocks need to be prepared by technically versed users according to the basic monitoring requirements of a business environment, it is not applicable to predefine tailored event-processing logic for any possible use case in that environment. For instance, one would usually refrain from predefining email actions for all employees of a company. Building blocks therefore need to be configurable by business operators, in a way that abstracts from the low-level event-processing logic that is represented by a building block. A building block then serves as a template which, when all required data are provided by a business operator, is instantiated to concrete event-processing logic as part of a rule.
- **Ease of use**: Sense-and-respond rules are typically administrated by business users with little or no technical expertise. Creating and deploying these rules shall be as straightforward and fail-safe as possible and fully abstract from the event-based foundation of decision making. In other words, business operators shall neither have to care about the implementation of building blocks, nor about the execution of concrete sense-and-respond rules as part of an event-based application.
- **Personalized rule management**: In many business scenarios – although, not necessarily – sense-and-respond rules may be defined within a business operator’s particular area of responsibility rather than as application-wide steering logic. For instance, a system administrator could demand receiving

an email whenever an alarm occurs on a server for which he or she is responsible (while another administrator may want to receive a short message for severe errors only and a third is on vacation anyway).<sup>6</sup> To adequately support a user-centered approach to sense-and-respond rule management, a framework needs to support a notion of *rule ownership* so that each rule can be associated with a specific user. This, in turn, enables restricting the visibility or accessibility of rules on a user basis.

- **Rule activation and scheduling:** In contrast to infrastructural rules which typically shall run 24/7, high-level business logic as realized through sense-and-respond rules may often be subject to temporal restrictions, such as, for instance, being suspended on weekends and holidays. An approach to sense-and-respond rule management shall therefore provide mechanisms that allow business operators to pause and resume the execution of event-processing logic easily, without having to remove and re-create a rule in its entirety. Moreover, the approach shall provide mechanisms for the scheduled execution of sense-and-respond rules – e.g., based on a calendar – where the activation state of a rule is adapted automatically.
- **Hot deployment:** Given a running event-processing infrastructure and a collection of predefined building blocks, business operators shall be able to work autonomously, generally independent from other users of an application or any kind of temporal restrictions. As a consequence, an approach to sense-and-respond rule management requires a possibility for “hot deployment”, i.e., to enact, change, and remove event-processing logic without having to stop and restart an event-based application as a whole.
- **Security:** An approach to sense-and-respond rule management shall eventually enable rule managers to clearly define the competences of a business operator, e.g., through specifying what building blocks are available to the business operator for creating sense-and-respond ruling logic. Business operators shall not have access to the event-processing infrastructure of an application at any point in time.

### 5.5.2 Model Overview

Figure 5.8 provides an overview of the proposed approach to sense-and-respond rule management.

The key concept of the model is that of configurable “building blocks” of event-processing logic, which can be assembled to sense-and-respond logic without having to understand the event-based foundations of decision making. Building blocks may thereby be considered as templates, which typically must be

---

<sup>6</sup> The role of personalized information delivery in Complex Event Processing is, for instance, highlighted by Vidačković et al. [134].

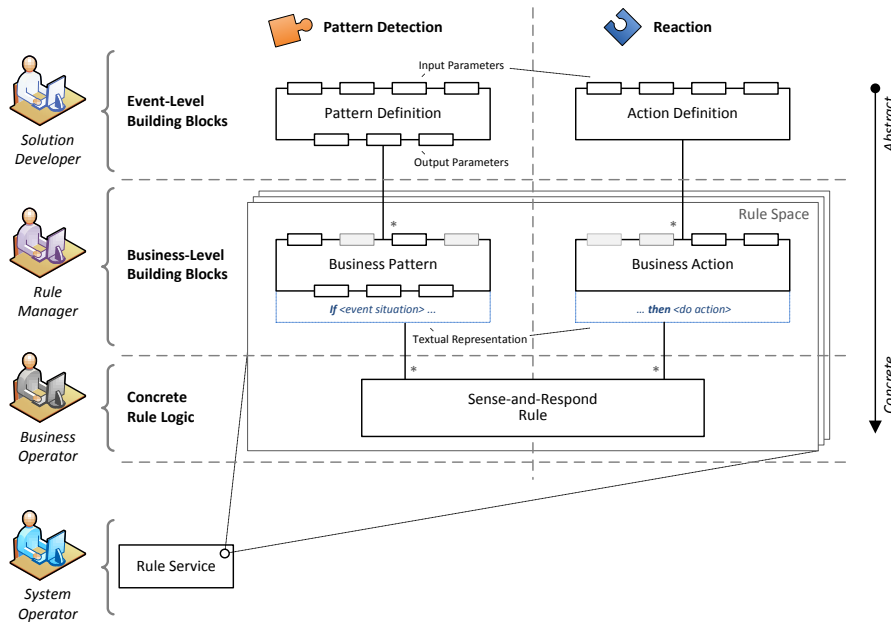


Figure 5.8. Overview of Sense-and-Respond Rule Management

provided with additional information in order to be instantiated and used as part of concrete rule logic.

In the presented model, building blocks are separated in vertical direction, into *pattern-detection logic* and *reaction logic*, and in horizontal direction, into *event-level building blocks* and *business-level building blocks*. The vertical separation follows naturally from a decoupling of pattern-detection logic and reaction logic, which we identified as a key requirement to the proposed approach in Section 5.5.1. In horizontal direction, a separation of more generic event-level building blocks and more specific business-level building blocks facilitates both the reusability of building blocks and their ease of use from a business operator’s point of view.

Rule spaces organize building blocks and sense-and-respond rules based on basic organizational tasks for which they are relevant. Besides structuring the overall event-processing logic of an application, rule spaces form the point of integration between sense-and-respond rules and the event-processing infrastructure of an application.



### Event-Level Building Blocks

Event-level building blocks encapsulate low-level event-processing logic in the form of *decision graphs* and *response-event templates*, respectively. While not directly visible to business operators, event-level building blocks form the ultimate foundation of our approach to rule management; any sense-and-respond rule is, in fact, assembled from instantiations of so-defined pieces of event-processing logic. In the proposed workflow, event-level building blocks are designed by so-called *solution designers* based on the general monitoring and steering requirements of the given business scenario, and organized in *rule spaces* based on the organizational tasks for which they are relevant. An event-level building block may thereby be used in several rule spaces, resulting in an *n-to-m* relationship.

Typically describing event-processing logic of high complexity, the focus with event-level building blocks is (i) on *reusability* across different application scenarios, and (ii) on *information hiding*. Reusability is achieved through a set of typed input parameters for all points of variability; to apply the event-processing logic in a specific use case, values for these input parameters must be specified. Information hiding is achieved through the set of input parameters on the “input side” of an event-level building block. On the “output side”, pattern definitions specify a collection of typed output parameters, which provide insights to a triggering event situation in an abstracted and controller manner. Multiple signals allow distinguishing between different manifestations of a business situation. Action definitions, by contrast, do not expose a so-defined output interface.

Pattern definitions and action definitions are presented in greater detail in Section 5.5.3 and Section 5.5.4, respectively. As several concepts apply to both kinds of entities, our discussion primarily focuses on pattern definitions. The action part is addressed more briefly, with the spotlight on possible differences to the pattern part.

### Business-Level Building Blocks

Event-level building blocks as discussed above form the ultimate foundation of any concrete sense-and-respond rule. However, due to their high level of generality, business users might easily be overwhelmed by using them directly.

Serving as an intermediate layer between the low-level event-processing logic of a rule space and business operators, the basic aim of *business-level building blocks* is to simplify the instantiation of an underlying event-level building block with respect to the specific use cases in which it is to be used. This is achieved through a further specification of the underlying event-level building

block: While a business-level building block would typically remain configurable to some extent (although, not necessarily), it may further specify the underlying event-level building block by setting input parameters or restricting their input domains. In case of pattern-detection logic, a business-level building block may furthermore restrict the set of signals that are relevant for the described class of event situations. Business-level building blocks eventually define a textual representation of the encapsulated event-processing logic, with placeholders for all unset input parameters. These textual representations are then presented to business operators, which assemble sense-and-respond logic as natural-language sentences from so-defined clauses. Underlying event-level building blocks are not visible to business operators at any point in time.

Business-level building blocks set up on the event-level building blocks of a rule space in a 1-to- $n$  relationship, meaning that one event-level building block may form the basis for an arbitrary collection of business-level building blocks. Given a rule space with a collection of event-level building blocks as defined by a solution designer, business-level building blocks are defined by senior business users – so-called *rule managers* – based on the specific problems business operators are going to face in their daily work. Simply “wrapping” event-level building blocks, the creation, modification, and deletion of business-level building blocks does not require adaptations of the underlying event-processing infrastructure and, thus, can be performed after the basic set up of an application.

Business patterns and business actions are presented in greater detail in Section 5.5.5 and Section 5.5.6, respectively. As with event-level building blocks, our discussion primarily focuses on the pattern part of the model.

### Sense-and-Respond Rules

In the final step of the proposed workflow, domain experts with little or no knowledge of the event-based foundations of a SARI application – so-called *business operators* – use business-level building blocks to assemble and instantiate concrete sense-and-respond rules of the form “if pattern, then action(s)”. Setting up on the event-based image of a business environment and directly or indirectly feeding back to this business environment, sense-and-respond rules eventually represent the actual decision-making logic of an application.

Having selected a rule space, business operators choose a business pattern and associate it with one or more business actions according to the monitoring task they want to implement. In parallel, business operators define concrete input-parameter values for all previously unset input parameters of the incorporated building blocks: In case of business actions, input-parameter values may be calculated from output parameters of the associated business pattern. The described approach therefore enables reaction logic to dynamically adapt to

the triggering event-situation instance. In case of business patterns, input-parameter values are necessarily constant.

As both kinds of business-level business entities abstract from the represented event-processing logic through a high-level textual representation, creating a sense-and-respond rule can be presented to the user as assembling a natural-language sentence from prepared clauses and replacing input-parameter placeholders in that sentence by concrete values.

Sense-and-respond rules are presented in greater detail in Section 5.5.7.

## Rule Spaces

Event-level building blocks (in an  $n$ -to- $m$  relationship) as well as business-level building blocks (in a 1-to- $n$  relationship)<sup>7</sup> are organized in so-called *rule spaces* based on the organizational tasks for which they are relevant. Structuring the overall set of sense-and-respond event-processing logic of an application, rule spaces play a crucial role for both (i) the creation of sense-and-respond rules through business operators, and (ii) the integration of sense-and-respond rules with the event-processing infrastructure of an application in a way that is transparent for business users.

**Creating sense-and-respond rules.** Business operators always create sense-and-respond rules *within* a rule space, based on the business-level building blocks of this rule space. With rule spaces grouping event-processing logic that makes sense concerning a certain organizational task, business operators are thereby confronted with task-relevant building blocks only, which facilitates a quick, secure, and fail-safe rule creation process. Rule spaces eventually serve as the primary unit for user rights management, as they can be assigned to business operators depending on their specific skills and functions within a company. User rights management in sense-and-respond rule management is discussed in greater detail in Section 5.6.

**Integrating sense-and-respond rules.** In the proposed approach, all sense-and-respond rules of a rule space are guaranteed to be assembled from instantiations of the event-level building blocks of this rule space. Based on the collection of event-level building blocks of a rule space – however, independent from both business-level building blocks and the concrete sense-and-respond rules in that rule space – it is therefore possible to deduce

- (i) what kinds of *input events* must be fed to a rule space, and

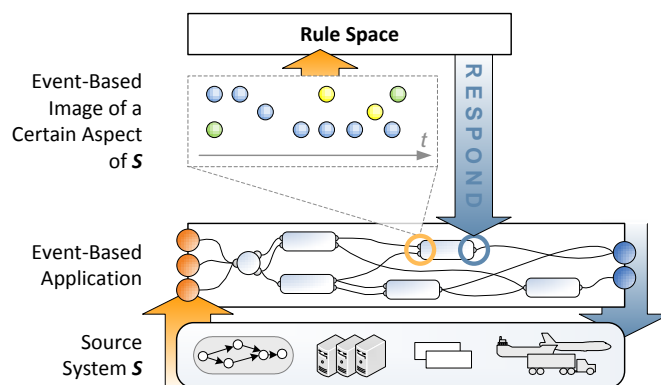
---

<sup>7</sup> A 1-to- $n$  relationship between rule spaces and business-level building blocks is not a technical necessity; instead, a business-level building block could be part of multiple rule spaces as long as the underlying event-level business pattern is available with all these rule spaces. In the proposed architecture, we opted for the 1-to- $n$  relationship to reduce the number of application-wide entities.

- (ii) what event-processing logic must be applied to the *response events* of a rule space,

such that the however-defined sense-and-respond rules of a rule space actually work as part of an event-based application. By decoupling the pre- and post-processing from the concrete decision-making logic, rule spaces allow bridging the gap between solution designers and system operators in their (often parallel and inherently cooperative) work on an application that is ready to use for rule managers and business operators: Given a rule space with a set of event-level building blocks as defined by a solution designer, a system operator maps this rule space to one or more *rule services*, which are then said to “host” the rule space. Depending on the used event types on both the input and the output side of the rule space, such a rule service must then be integrated into the event-processing infrastructure of an application as part of an event processing map. During run time, all sense-and-respond rules that are created within a rule space are assigned to the hosting rule services implicitly; i.e., whenever a business operator creates a rule, this rule is automatically and transparently applied in appropriate parts of an underlying event processing map.

In accordance with the overall role of sense-and-respond rules as sketched in Figure 5.4 – setting up on an event-based image of the underlying business environment – a rule space may now be considered as setting up on an event-based image of a certain “aspect” of the underlying business environment. Which parts of a source system actually belong to such an aspect is defined by system operators, through the specific event-processing logic that precedes the concerned rule service(s) in the application’s event processing maps. Figure 5.9 illustrates the described role of rule spaces in a SARI application.



**Figure 5.9.** Rule Spaces in a SARI Application

Rule spaces are presented in greater detail in Section 5.5.8.

### 5.5.3 Pattern Definitions

Pattern definitions represent pattern-detection logic in a form that can be interpreted by SARI for the automated analysis of incoming event streams. Together with their counterpart for reaction logic – so-called *action definitions* – pattern definitions form the base elements of the proposed approach to sense-and-respond rule management: Any sense-and-respond rule is, eventually, based on an instantiation of a pattern definition. When an event situation matches the pattern, the associated reaction logic is triggered.

Pattern definitions are based on an abstract decision graph, which allows instantiations of the represented pattern-detection logic to be evaluated on common rule services. Focusing on reusability and information hiding, pattern definitions abstract from the encapsulated pattern-detection logic through collections of *input parameters* and *output parameters*, respectively. Input parameters allow configuring pattern-detection logic based on the specific context in which it is used without having to understand and/or change the underlying decision graph. Output parameters provide access to selected characteristics of a triggering event situation in a controlled and abstracted manner. *Signals* allow distinguishing between different manifestations of an event situation.

In the proposed workflow, pattern definitions are created and assigned to rule spaces by technically versed *solution designers* based on the general monitoring requirements of the given business scenario. Pattern definitions are then used by rule managers to create business patterns, which are typically less generic and can be used by business operators for assembling concrete sense-and-respond rules.

#### Meta Model

Figure 5.10 shows the meta model for pattern definitions. A pattern definition  $p = (IN, OUT, d)$  is defined by a collection of *input parameters*  $IN$ , a collection of *output parameters*  $OUT$ , and a *passive decision graph*  $d$ . A decision graph is said to be passive when it is assembled from condition components, time-based components, and so-called *signals* only.

**Input parameters.** Input parameters allow configuring the pattern-detection logic of a pattern definition based on the specific use case in which it is used, without having to change or understand the encapsulated low-level pattern-detection logic. Within the decision graph of a pattern definition, input parameters may be used as typed placeholders – so-called *decision graph variables* – in the various expressions of the used rule components; for instance, given an input parameter “ServerID” of type String, a condition component could be configured to evaluate the following condition on a triggering “Alarm” event:

$$\text{Alarm.Server} > \$\text{ServerID}$$

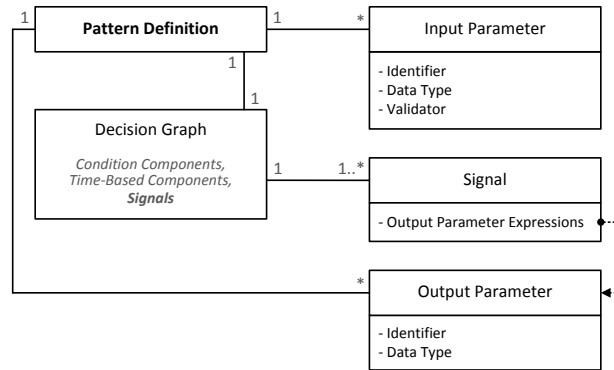


Figure 5.10. Pattern Definition Meta-Model

In an instantiation of the encapsulated pattern-detection logic, the component would then activate its “true” port only with those events that come from the specified server.

An input parameter  $in = (i, n, d, t, validate)$  is defined by an identifier  $i$ , a display name  $n$ , a description  $d$ , a data type  $t$ , and an optional validator  $validate : t \rightarrow \{0, 1\}$ . If specified, a validator allows further restricting the set of possible input-parameter values for  $in$ ; given an input-parameter value  $v : t$ ,  $validate(v) = 1$  must hold. On the level of pattern definitions (as well as action definitions),<sup>8</sup> validators would typically be used to ensure the syntactic consistency of the encapsulated event-processing logic, e.g., to avoid division by zero.

**Output parameters.** In almost any use case, the action part of an event-pattern rule needs access to selected characteristics of the triggering event situation. The extraction of relevant data from an event situation requires, however, detailed knowledge of both the triggering event situation as a whole and the events it is made of. Pattern definitions provide for an abstraction of the triggering event situation through the use of output parameters, which allow rule managers to specify those aspects of a triggering event situation that are supposed to be relevant when using the pattern definition in a concrete sense-and-respond rule. For integrating the action part with the pattern part of a rule, business operators may then resort to a plain list of typed, named, and documented data fields, with the event-based source of the data being hidden entirely. In the proposed model, an output parameter  $out = (i, n, d, t)$  is defined by an identifier  $i$ , a display name  $n$ , a description  $d$ , and a data

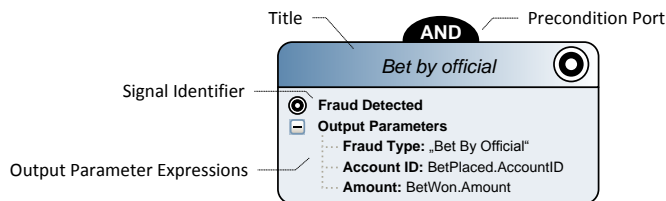
<sup>8</sup> Validators can also be defined as part of an input-parameter configuration on the level of business patterns and business actions as described in Section 5.5.5 and Section 5.5.6, respectively. Here, validators are used to restrict the scope of a pattern definition or action definition with respect to the specific context in which it is to be used.

type  $t$ . The actual value of an output parameter is calculated in the various signals of the pattern definition’s decision graph, where expressions for each output parameter are specified.

**Decision graph.** The actual pattern-detection logic of a pattern definition is defined as a decision graph, which allows evaluating instantiated pattern-detection logic on a usual rule service. The decision graph of a pattern definition is required to be *passive*, which means that it must not contain *action components* in the common sense as discussed in Section 4.3.4. Instead, the specific actions to be executed when an event situation is detected are specified in sense-and-respond rules by business operators. A decision graph still requires, however, a dedicated class of rule components to signify the end nodes of the described pattern-detection logic.

We therefore introduce *signals*, special action components that abstract from concrete reaction logic and simply notify the detection of an event situation to arbitrary signal listeners. A signal  $s = (i, X_{OUT})$  is defined by an identifier  $i$  and a collection of *output-parameter expressions*  $X_{OUT}$ , with an appropriately-typed expression on the triggering event situation and the collection of input parameters for each output parameter in  $OUT$ . When a signal is activated, the output-parameter expressions are evaluated and concrete output-parameter values are calculated. In a sense-and-respond rule, the so-calculated output parameter values are then passed over to the action part of the rule.

Figure 5.11 shows an exemplary signal “Fraud detected”, with output-parameter expressions for three output parameters “Fraud Type”, “Account ID” and “Amount”. While the first expression simply passes a constant, the second and the third extract the demanded information from the triggering event situation.



**Figure 5.11.** Signal Component

By definition, the decision graph of a pattern definition must contain at least one signal, and a single signal would be the right choice in a majority of use cases. In certain scenarios, however, a business situation may occur in different *manifestations*: Consider a pattern definition that detects security violations to a IT systems; here, although equivalent regarding their basic semantics, one could distinguish between an “Anomaly”, a “Warning”, and an “Alarm”

based on the exact sequence of events and/or their specific attribute values. In such cases, the decision graph of a pattern definition may contain multiple signals, each with its separate collection of output-parameter expressions. The collection of signals then describes possible output states of a pattern definition.

**Example**

Table 5.1 shows a simple pattern definition “DB transaction duration check”. Based on an incoming event of type “DB Log Message”, it checks whether the represented database transaction outruns a user-defined maximum duration. As output parameters, it provides access to the actual transaction duration, the maximum duration (which is simply passed through from the input parameter), as well as the transaction type. Being based on the occurrence of a single event, the pattern definition sets up on a stateless decision graph, meaning that it does not have an underlying correlation set.

| Input Parameters  | ID  | Type      | Validator  |
|-------------------|---|-----------|------------|
|                   | Accepted Transaction Duration   | Time span | $x \geq 0$ |
| Decision Graph    | <p>The decision graph consists of two main components. The first is a yellow box titled "Test incoming DB log message" with a lightning bolt icon. It contains a "DB Log Message" event and a "Conditions" section with the expression "TransactionTime &gt; \$AcceptedTransactionDuration". Below the conditions are two circular nodes labeled "True" and "False". An arrow points from the "True" node to a second box. The second box is blue and titled "Long running transaction detected..." with a target icon. It contains a signal "S1" and an "Output Parameters" section with three entries: "Actual Transaction Duration: DbLogMessage.TransactionTime", "Accepted Transaction Duration: \$AcceptedTransactionDuration", and "Transaction Type: DbLogMessage.TransactionType".</p> |           |            |
| Output Parameters | ID  | Type      |            |
|                   | Actual Transaction Duration   | Time span |            |
|                   | Accepted Transaction Duration   | Time span |            |
|                   | Transaction Type  | String    |            |

**Table 5.1.** Exemplary Pattern Definition

**5.5.4 Action Definitions**

Action definitions form the counterpart to pattern definitions on the level of *event-level building blocks* as sketched in Section 5.8. An action definition



encapsulates an abstract response-event template, which may be considered as a blueprint for concrete response events to be generated when instantiated reaction logic is executed as part of a sense-and-respond rule. In turn, such a response event would actually trigger the represented action in a downstream event service, or in the underlying source system.

### Meta Model

Figure 5.12 shows the meta model for action definitions. An action definition  $a = (IN, r)$  is defined by a collection of *input parameters*  $IN$  and a *response-event template*  $r$ . As with pattern definitions, input parameters allow configuring the encapsulated event-processing logic depending on the specific context in which it is used. Within an action definition, an input parameter may now be used as a typed placeholder in the diverse event-attribute expressions of the *response-event template*. Event-attribute values (and, thus, the basic semantics) of a response event then depend on the concrete input-parameter values as eventually specified in a business action or a sense-and-respond rule. The response-event template describes the general structure of the action definition's event representation. A response-event template  $r = (t, X) \mid X = \{x_1, x_2, \dots, x_n\}$  is defined by an event type  $t$ , and, for each event attribute  $a_i = (i_i, t_i) \in t$ , an expression  $x_i$  on the collection of input parameters  $IN$  returning a value of type  $t_i$ .

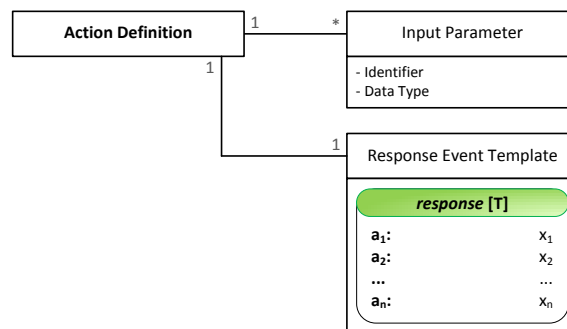


Figure 5.12. Action Definition Meta-Model

### Example

In workload automation, overload situations may often be resolved automatically through additional resources or rescheduling of tasks. The example application for event-based service assurance therefore provides a variety of so-called *system actions*: Based on a web-service call, they directly feed back into

the automation platform, where they may request machines, pause or cancel running tasks, or delay the execution of scheduled ones. Table 5.2 shows an exemplary action definition “Start automation platform task”. Exposing a string-typed input parameter “Task”, it eventually results in a respectively-configured “Web Service Action” event, where the “Arguments” attribute is calculated from the provided task name. While the method name is provided as a constant, the exact path to the web service’s WSDL is read from an application-wide resource string.

| Input Parameters        | ID                                  | Type       | Validator                     |
|-------------------------|-------------------------------------|------------|-------------------------------|
|                         | Task                                | String     | -                             |
| Response-Event Template | Response-Event Type:                |            |                               |
|                         | com.example.common.WebServiceAction |            |                               |
|                         | Event Attribute                     | Expression |                               |
|                         | ID                                  | Type       |                               |
|                         | WSDL                                | String     | \$\$WSDLPath                  |
|                         | Method                              | String     | “executeObject”               |
|                         | Arguments                           | List       | { \$Task, “UTC”, Now(), ... } |
|                         | ...                                 | ...        | ...                           |

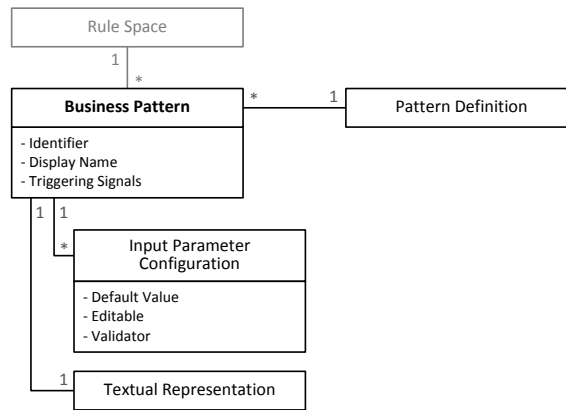
**Table 5.2.** Exemplary Action Definition

### 5.5.5 Business Patterns

In the proposed architecture, business patterns represent pattern-detection logic in a way that is understandable and usable to business users with little or no knowledge of the event-based foundations of decision making. A business pattern further specifies the (in many cases, relatively generic) class of event situations described by an underlying pattern definition, which enables rule managers to optimize the instantiation of the represented pattern-detection logic with respect to the specific contexts in which it is used. This specification is limited to the *interface* of the pattern definition – to *input parameters* on the input side of the pattern definition, to *signals* on the output side – and has no effect on the actual pattern-detection logic. In addition, business patterns provide a textual representation of the described class of event situations, with placeholders for all previously unset input parameters. Together with their counterparts for reaction logic – so-called *business actions* – business patterns and their textual representations are eventually presented to business operators, which assemble concrete sense-and-respond logic as a natural-language sentence from the so-defined clauses.

### Meta Model

Figure 5.13 shows the meta model for business patterns. Let  $p$  be a pattern definition, and let  $S$  denote the signals in  $p$ . A business pattern  $bp = (p, C, S_{triggering}, t)$  is then defined by an underlying pattern definition  $p$ , a collection of *input parameter configurations*  $C$  for all input parameters of  $p$ , a non-empty collection of *triggering signals*  $S_{triggering} \subseteq S$ , and a textual representation  $t$ . While input-parameter configurations and the set of triggering signals allow specifying the class of event situations described by a business pattern, the textual representation serves as a layer of abstraction between the encapsulated pattern-detection logic on the one hand and the business operator on the other.



**Figure 5.13.** Business Pattern Meta-Model

**Input parameter configurations.** Input-parameter configurations allow further specifying the underlying pattern definition by setting or restricting the input domain of its input parameters. Given an input parameter  $in = (i, n, d, t, validate_{base})$ , an input-parameter configuration  $c_{in} = (v, d, validate) \in C$  for  $in$  is defined by an optional input-parameter value  $v : t$ , an *editable* flag  $d \in \{0, 1\}$ , and an optional validator  $validate : t \rightarrow \{0, 1\}$ . If the editable flag is set,  $v$  can still be adapted during the instantiation of sense-and-respond rules; in such a case,  $v$  may be considered a default value for  $in$ . If no input-parameter value is specified,  $d$  is set by definition. The validator  $validate$  allows further restricting the set of possible values for  $in$ . If a validator  $validate_{base}$  is specified for the concerned input parameter in the underlying pattern definition,  $validate$  must be a specialization of its base validator, i.e.,  $validate_{base}(x) = 1 \rightarrow validate(x) = 1$ . Note that a validator is of practical relevance for editable parameters only; predefined values are not subject to validation.

**Triggering signals.** In contrast to pattern definitions, business patterns do not distinguish between different manifestations of a business situation on the level of signals; in general, a business pattern triggers if *any* signal is activated in the underlying decision graph. A rule manager may, however, restrict the set of signals that are generally relevant for a business pattern. For instance, having a pattern definition “Security violation” with three signals “Anomaly”, “Warning”, and “Alarm”, a first, more general business pattern “Security violation” could trigger based on all three signals. A second, more specific business pattern “Security alarm” could be restricted to the “Alarm” signal so that detections of less severe situations in the underlying decision graph would not affect the business pattern. In the proposed meta model, the collection of *triggering signals*  $S_{triggering} \subseteq S$  indicates which signals of the underlying pattern definition shall be considered.

**Textual representation.** The textual representation  $t$  of a business pattern provides a natural-language description of the represented class of event situations, with placeholders for all *editable* input parameters.<sup>9</sup> To facilitate a high-level, business-oriented approach to rule creation, the textual representation would typically describe the real-world business situation that originally caused a matching event situation, rather than describing the exact event sequence by itself. For instance, given a fraud pattern, one would speak of a “*fraud attempt in league x*” rather than a specific sequence of “Cash In”, “Bet Placed”, and “Cash Out” events. During the assembling of business-level building blocks to concrete sense-and-respond rules, the placeholders of a textual representation are eventually replaced by the desired input-parameter values. By definition, a textual representation for business patterns is required to begin with the term “if” or its equivalent in another language.

### Example

Table 5.3 shows an exemplary business pattern “2 seconds DB transaction” based on the pattern definition “DB transaction duration check” as presented in Section 5.5.3. Setting a non-editable expression of two seconds –  $TimeSpan(0, 0, 2)$  in Event Access Expression (see Section 3.7) syntax – for input parameter “Accepted Transaction Duration”, the business pattern can be used by business operators directly, without having to specify further properties. As a consequence, the business pattern’s textual representation does not specify any input-parameter placeholders.

---

<sup>9</sup> The textual representation of a business-level building blocks is not to be confused with the common *description* of rule-management entities. While the former is directly used to instantiate event-processing logic and thus must conform to a specified structure, the latter is purely descriptive and may be omitted if no further information is required.

|                               |   |                   |                   |
|-------------------------------|---|-------------------|-------------------|
| <b>Pattern Definition</b>     | DB transaction duration check (Section 5.5.3)           |                   |                   |
| <b>Textual Representation</b> | (if) a database transaction takes longer than 2 seconds |                   |                   |
| <b>Input Parameter</b>        | <b>ID</b>   | <b>Expression</b> | <b>Editable</b>   |
| <b>Configurations</b>         | Accepted Transaction Duration                           | TimeSpan(0, 0, 2) | ×                 |
| <b>Triggering Signals</b>     | <b>Signal ID</b>  |                   | <b>Triggering</b> |
|                               | S1  |                   | ✓                 |

Table 5.3. Exemplary Business Pattern

### 5.5.6 Business Actions

Business actions form the counterpart to business patterns on the level of *business-level building blocks* as sketched in Section 5.8. As with business patterns, business actions further restrict the event-processing logic of an underlying event-level building block – in this case, of an *action definition* – and provide a natural-language description with placeholders for all previously unset input parameters.

#### Meta Model

Figure 5.14 shows the meta model for business actions. A business action  $ba = (a, C, t)$  is defined by an underlying action definition  $a$ , a collection of input parameter configurations  $C$  for all input parameters of  $a$ , and a textual representation  $t$ . Input parameter configurations for business actions are generally equivalent to input parameter configurations for business patterns as discussed above. The textual representation of a business action would typically describe its ultimate result (as it would be visible to a business operator) rather than its immediate, technical implications; for instance, a business action that results in an “Email” event would be described as “*sending an email*”. By definition, a textual representation for business actions begins with the term “then” or its equivalent in another language.

#### Example

Figure 5.4 shows an exemplary business action “Start task” based on the action definition “Start automation platform task” as presented in Section 5.5.4. Unlike the exemplary business pattern as discussed before, it leaves the underlying event-level building block’s input parameter editable, however, specifies a *validator* in accordance with the specific context – i.e., *rule space* – in which it is used. The editable input parameter is available as a placeholder in the business action’s textual representation. In the remainder of this thesis, we show placeholders colored and in italic font.

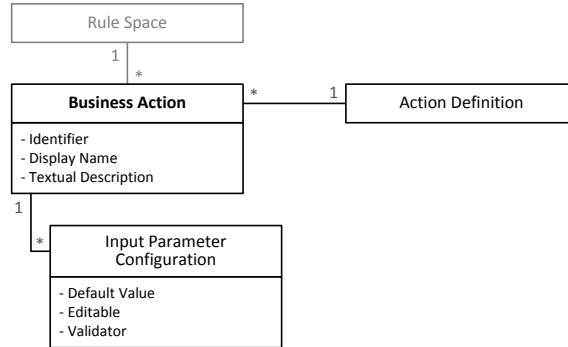


Figure 5.14. Business Action Meta-Model

| <b>Action Definition</b>              | Start automation platform task (Section 5.5.4)  |          |   |          |           |        |  |   |   |
|---------------------------------------|---|----------|---|----------|-----------|--------|--|---|---|
| <b>Textual Representation</b>         | (then) start <i>Task</i>  |          |   |          |           |        |  |   |   |
| <b>Input Parameter Configurations</b> | <table border="1"> <thead> <tr> <th>ID</th> <th>Expression</th> <th>Editable</th> <th>Validator</th> </tr> </thead> <tbody> <tr> <td>Task -</td> <td></td> <td>✓</td> <td><math>x \in \{\text{JOBP.INFSTR.PROVISIONHOST}, \dots\}</math></td> </tr> </tbody> </table> | ID       | Expression  | Editable | Validator | Task - |  | ✓ | $x \in \{\text{JOBP.INFSTR.PROVISIONHOST}, \dots\}$ |
| ID                                    | Expression  | Editable | Validator   |          |           |        |  |   |   |
| Task -                                |   | ✓        | $x \in \{\text{JOBP.INFSTR.PROVISIONHOST}, \dots\}$ |          |           |        |  |   |   |

Table 5.4. Exemplary Business Action

### 5.5.7 Sense-and-Respond Rules

In the proposed architecture, business patterns and business actions are eventually presented to business operators to create concrete event-processing logic in the form of sense-and-respond rules. As both kinds of business-level building blocks abstract from the underlying complexity through a high-level textual representation of the event-processing logic, the process of assembling a sense-and-respond rule can be presented to the user as assembling a natural-language sentence in the form “*if real-world situation, then real-world action(s)*” from prepared clauses. Input parameters are seamlessly integrated into the textual representations and can successively be replaced by concrete values during the instantiation process.

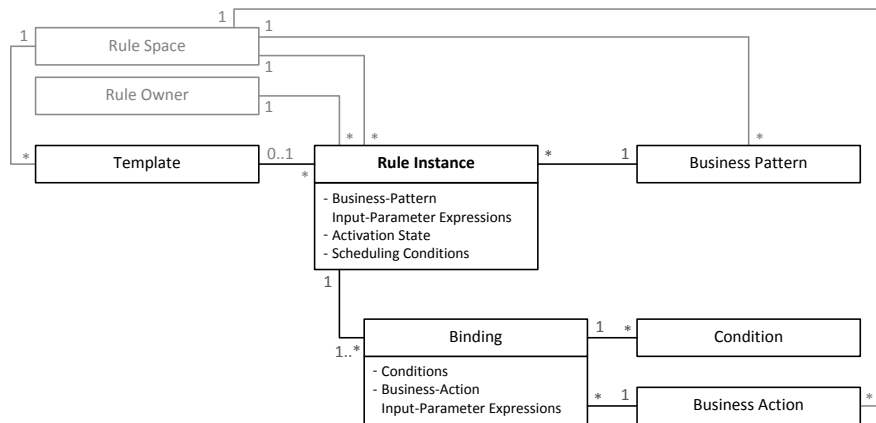
In addition to the basic association of an instantiated business pattern with one or more instantiated business actions, sense-and-respond rules provide a mechanism for setting the *activation state* of a rule: Only if a sense-and-respond rule is *active* it is considered for event processing. In accordance with the basic requirements as discussed in Section 5.5.1, sense-and-respond rules may either be activated and paused manually, or be executed in a scheduled execution mode. In the latter case, the activation state of a rule is adapted automatically based on predefined time conditions.

Sense-and-respond rules are created and exist within *rule spaces*, which group building blocks and allow mapping sense-and-respond rules to rule services in a way that is transparent for business users. Each sense-and-respond rule

is furthermore associated with a *rule owner*, who is typically, although not necessarily, its creator. The role of rule owners in user rights management is discussed in greater detail in Section 5.6.

## Meta Model

Figure 5.15 shows the meta model for sense-and-respond rules. A sense-and-respond rule  $r = (bp, X_{bp}, B, a, S, t)$  is defined by a business pattern  $bp$ , a collection of *input parameter expressions*  $X_{bp}$  for all input parameters of  $bp$ , a collection of *bindings*  $B$ , an *activation state*  $a$ , a collection of *scheduling conditions*  $S$ , and an optional *template*  $t$ . Bindings associate the business pattern of a rule with business actions. A binding  $b = (ba, X_{ba}, C) \in B$  is defined by a business action  $ba$ , a collection of input-parameter expressions  $X_{ba}$  for all input parameters of  $ba$ , and a collection of *conditions*  $C$ .<sup>10</sup>



**Figure 5.15.** Sense-and-Respond Rule Meta-Model

**Input parameter expressions.** Input parameter expressions define concrete values for the diverse input parameters of the rule’s business pattern and its business actions, respectively. In case of business actions, input parameter expressions may calculate such a value from the output parameters of the business pattern. This enables rule managers to adapt reaction logic dynamically based on the triggering event-situation instance. Business-pattern input-parameter expressions, by contrast, are necessarily constant.

<sup>10</sup> Serving as an auxiliary construct for associating business patterns with business actions rather than being visible as an independent entity to business operators, bindings do not have a display name and human-readable description.

**Conditions.** In the proposed approach to sense-and-respond rule management, the role of business operators in defining relevant business situations is deliberately focused on choosing an appropriate business pattern and providing concrete values for its input parameters. Conditions provide a simple mechanism for business operators to further specify the so-defined pattern-detection logic based on the characteristics of a triggering event-situation instance, in a way that abstracts from the event-based foundations of decision making. A condition may basically be considered as a Boolean expression involving one or more output parameters of the business pattern, which is evaluated when the business pattern triggers. Only if all conditions of a binding evaluate to *true*, the associated business action is executed.

In the proposed meta model, a condition  $c = (IN, x, t, X_{IN}) \in C$  is defined by a collection of typed input parameters  $IN$ , a Boolean expression  $x$  on the output parameters of the business pattern and the collection of input parameters, a human-readable textual representation with placeholders for all input parameters, and a collection of input-parameter expressions  $X_{IN}$  for all input parameters in  $c$ .<sup>11</sup> Although conceptually equivalent, we distinguish between condition and so-called *exceptions* in the web-based rule creation interface as presented in Section 5.9. Exceptions must evaluate to *false* and thus could be defined in a negated form as conditions.

**Activation State.** The activation state  $a \in \{\text{active, paused, scheduled}\}$  of a sense-and-respond rule indicates whether the rule is manually paused, manually activated, or its activation state shall be calculated from the set of *scheduling conditions* as discussed below. If no scheduling conditions are defined – i.e.,  $S = \emptyset$  – the latter option is unavailable.

**Scheduling conditions.** Scheduling conditions specify the automated activation of a sense-and-respond rule when executed in scheduled execution mode. Let  $\mathcal{T}$  be the set of all time stamps. A scheduling condition  $s : \mathcal{T} \rightarrow \{0, 1\} \in S$  is then defined as a Boolean expressions on time stamps, indicating whether the rule is active at given point in time or not. Given an incoming event  $e$ ,  $e$  is processed if  $s(\text{value}_t(e)) = 1 \forall s \in S$ .

**Template.** The template  $t$  signifies whether the sense-and-respond rule was created “from scratch” or based on a *template*. Templates are partially-defined rules for commonly needed event-processing logic and are discussed in greater detail with rule spaces in Section 5.5.8.

---

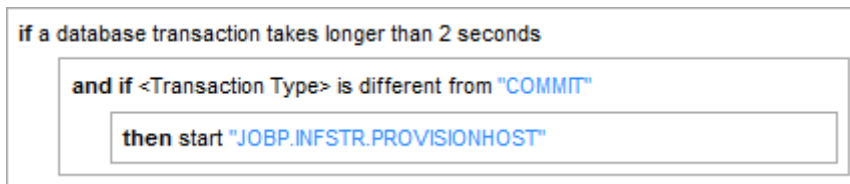
<sup>11</sup> Albeit not a technical necessity on the level of sense-and-respond rules, the concept of input parameters and input parameter expressions enables preparing partially-defined conditions, where only input-parameter values must be specified by a business operator. Such constructs are used in the web interface to ease the creation of sense-and-respond rules and also are an important part of templates as discussed in Section 5.5.8. Similar, textual representations allow rendering conditions in a way that is understandable to business users.



**Example**

Figure 5.16 shows an exemplary sense-and-respond rule “Host Provisioning HP1 - Transactions > 2 seconds”, assembled from the above-presented business-level building blocks “2 seconds DB transaction” and “Start task”. A condition ensures that the action is *not* triggered for database transactions of type `COMMIT`, which in the presented scenario are known to take longer than two seconds even if sufficient resources are available. Running in scheduled execution mode, the rule is configured to be deactivated during a defined maintenance interval through the following scheduling condition  $s_1$ :

$$s_1(t) = \begin{cases} 0 & t \in [(\text{Feb 29, 2012, 04:08 pm}), (\text{Feb 29, 2012, 11:07 pm})] \\ 1 & \text{otherwise} \end{cases}$$



**Figure 5.16.** Exemplary Sense-and-Respond Rule

**5.5.8 Rule Spaces**

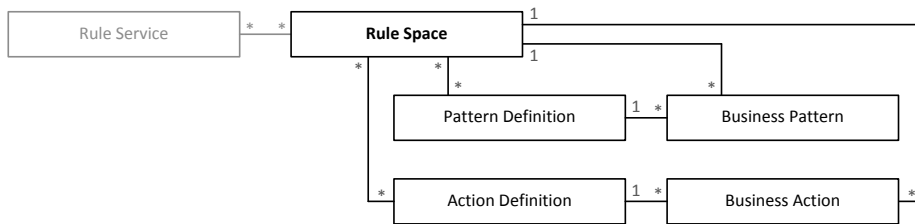
Rules spaces group the event-level building blocks (in an  $n$ -to- $m$  relationship) and the business-level building blocks (in a 1-to- $n$  relationship) of a SARI application based on organizational tasks to which they belong; for instance, a rule space “Fraud Detection” could contain event-processing logic for detecting different kinds of fraudulent user behavior, notifying the fraud department, and blocking a user account automatically. In the proposed architecture, rule spaces therewith form the basic workspaces for business operators: Each sense-and-respond rule is created *within* a rule space, based on the business-level building blocks of this rule space. In addition, rule spaces serve as the key point of integration between sense-and-respond rules and the event-processing infrastructure of an application: Each rule space is assigned to a collection of *rule services*, which then execute all sense-and-respond rules of that rule space. Whereas the creation of a rule space and the grouping of event-level building blocks is up to solution designers, the assignment of rule spaces to appropriately configured rule services lies in the responsibility of system developers.

Besides grouping building blocks that make sense regarding a certain aspect of an application, rule spaces provide additional mechanisms for guiding the

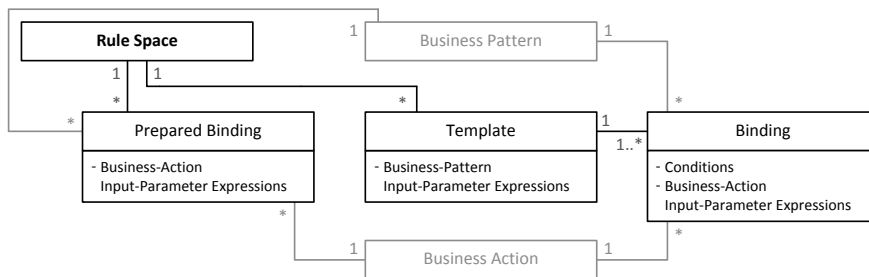
work of business operators: *Prepared bindings* allow defining combinations of business patterns and business actions which are particularly useful and thus shall be suggested to a business operator. *Templates* are partially-defined rules for commonly needed event-processing logic.

**Meta Model**

Figure 5.17 and Figure 5.18 show the meta model for rule spaces. Let  $P_{total}$  and  $A_{total}$  be the overall sets of pattern definitions and action definitions of an application. A rule space  $s = (P, A, BP, BA, B, T)$  is then defined by a collection of pattern definitions  $P \subseteq P_{total}$ , a collection of action definitions  $A \subseteq A_{total}$ , a collection of business patterns  $BP$  based on the pattern definitions in  $P$ , a collection of business actions  $BA$  based on the action definition in  $A$ , a collection of *prepared bindings*  $B$ , and a collection of *templates*  $T$ .



**Figure 5.17.** Rule Space Meta-Model



**Figure 5.18.** Rule Space Meta-Model: Prepared Bindings and Templates

**Prepared bindings.** Prepared bindings enable a rule manager to prepare associations between business patterns and business actions which are considered as meaningful and/or commonly requested in practical use cases. In the presented model, a prepared binding  $pb = (bp, ba, X_{ba})$  is defined by a business pattern  $bp$ , a business action  $ba$ , and a collection of input-parameter expressions  $X_{ba}$  for the input parameters of  $ba$ . When creating a sense-and-respond

rule based on  $bp$ , all “prepared” business actions – i.e., all business actions for which a prepared binding to  $bp$  exists in  $B$ <sup>12</sup> – are suggested to the business operator as being particularly suited in response to the given business situation.<sup>13</sup> When adding a prepared business action, possible input-parameter expressions of the prepared binding are set as default values. Note that for one tuple of business patterns and business actions, at most one prepared binding may be defined.

**Templates.** Templates are structurally equivalent to sense-and-respond rules, however, may leave open input-parameter expressions for the business pattern, for all conditions and for all business actions. Templates may therefore be considered not-yet-finished sense-and-respond rules: To create a full-fledged rule from a template, a user simply defines the still missing input-parameter expressions, if any. Rule managers typically provide templates for rule logic that is required frequently, with equal or similar configurations. In the proposed meta model, a template  $t = (bp, X_{bp}, B)$  is defined by a business pattern  $bp$ , a collection of input-parameter expressions  $X_{bp}$  for input parameters of  $bp$ , and a collection of bindings  $B$ . In contrast to sense-and-respond rules, collections of input-parameter expressions are not necessarily complete.

### Example

Our exemplary SARI application for event-based service assurance provides a total of six rule spaces, each of which is concerned with a different aspect of the underlying automation platform such as the runtime of individual tasks, exceptional log file entries, or access denials. The “Database protection monitoring” rule space is concerned with monitoring the automation platform’s data repository, which is used to keep track on the various actions of the platform and also serves as the basis for messaging in order to keep it reliable and persistent. On the pattern-detection side, the database-protection rule space provides diverse business patterns for testing the duration of transactions (e.g., “2 seconds DB transaction” as discussed in Section 5.5.5), detecting individual errors by their error code, and detecting accumulations of errors as well as critical database calls. On the reaction side, it provides different business actions for notifying responsible departments within a company as well as a whole range of business actions for starting, pausing, and canceling actions in the automation platform (e.g., “Start task” as discussed in Section 5.5.6).

<sup>12</sup> In the following, whenever speaking of *prepared business actions*, we assume that the respective business pattern is clear from the context.

<sup>13</sup> Whether or not *non-prepared* business actions are available depends on the specific authorizations of a business operator in the given rule space. A detailed discussion of user rights management are presented in Section 5.6.

## 5.6 User Rights Management

Sense-and-Respond Infrastructure follows a role-based approach to user rights management, where the specific authorizations of a user depend on the roles he or she is granted. From the various roles available, only a limited number concerns the aspect of rule management as presented in this thesis.<sup>14</sup> These roles conform closely – although, not entirely – to the conceptual user roles as introduced in Section 5.4 and Section 5.5 before.

In the following, we discuss rule-management-relevant user roles and their specific permissions beginning with the highest-level *super administrator* role and ending with the lowest-level *business operator* role. All these user roles are structured in a linear hierarchy, meaning that a higher-level user role inherits all authorizations from lower-level user roles.

### Super Administrators

Super administrator form the top-most layer of SARI’s overall role hierarchy, and, as such, have full and unrestricted access to a SARI installation. Having the exclusive competence to create new SARI applications and manage user rights, super administrators are necessarily involved in the set up of a rule-based SARI application. Super-administrator rights are not required for the actual definition of an event-processing infrastructure and high-level sense-and-respond logic.

### Application Developers

For any rule-management user role other than super administrators, SARI applications serve as a central “unit of authorization”: Each user is associated with a collection of SARI applications for which this user is then generally authorized. Otherwise, if a user is not authorized for a SARI application, this application is not at all visible to him or her.

Application developers have full and unrestricted access to the SARI applications for which they are authorized. However, they may not delete them or create new SARI applications. In the proposed rule management system, so-defined permissions would be granted to both *system operators* and *solution designers*. Since these actors typically work in close collaboration, no conflicts should occur.

---

<sup>14</sup> The majority of roles in fact concerns the ex-post analysis of event data as discussed by Suntinger et al. [123, 124], as well as the deployment of SARI applications.

### Rule Managers

At the bottom layers of the proposed user-role hierarchy – i.e., for *rule managers* and *business operators* – rule spaces serve as a second unit of authorization subordinate to SARI applications: Given an accessible SARI application, a user is associated with a collection of rule spaces of this application for which this user is then generally authorized. If a user is not authorized for a rule space, this rule space is not at all visible to him or her.

Rule managers are authorized to create, modify, and remove the *business-level building blocks*, *prepared bindings*, and *templates* of a rule space. Assuming that the various rule managers of a rule space operate in close collaboration, each rule manager has full access to all elements independent from an entity’s original creator.

### Business Operators

Business operators are generally authorized to administer sense-and-respond rules based on a predefined set of business-level building blocks, prepared bindings, and templates, but do not have access to any other part of a SARI application. Within this basic scope, the business operator role provides for the following detail settings:

- **Rule access mode:** Unlike infrastructural rule management, sense-and-respond rule management provides a notion of *rule ownership*; whereas per default, each rule is owned by its creator, the owner of a sense-and-respond rule may be changed by any user that has write access to this rule. In the proposed user-role hierarchy, business operators have full access to all sense-and-respond rules of which they are the owners. The *rule access mode* defines whether rules of another owner are (i) fully accessible, (ii) visible but not editable, or (iii) entirely hidden from the user.
- **Prepared bindings only:** Prepared bindings enable rule managers to declare business pattern/business action pairs that are considered particularly useful in the given business scenario. The *prepared bindings only* flag indicates whether a business operator is forced to create sense-and-respond rules from such prepared bindings or may define non-prepared bindings.
- **Editable templates:** Templates enable rule managers to predefine partially-defined rules for commonly needed event-processing logic. The *editable templates* flag indicates whether the structure of a template must always be instantiated *as is*, or instead may serve as a default structure that can be modified during rule creation. In the latter case, a template must explicitly be “broken” in order to be modified in structure. A resulting sense-and-respond rule then does not maintain any reference to the template from which it originates.

Provided that a user is generally authorized to view the rule space, any higher-level user role has full access to the sense-and-respond rules of a rule space.

### 5.7 Implementation Architecture

We fully implemented the proposed approach to rule management as an extension to the basic SARI architecture as discussed in Section 3.6. Figure 5.19 shows those parts of the extended architecture that directly and indirectly participate in the management and execution of infrastructural rules and sense-and-respond rules, respectively. Unless otherwise stated, all elements, mechanisms, and communication channels of the original architecture are preserved.

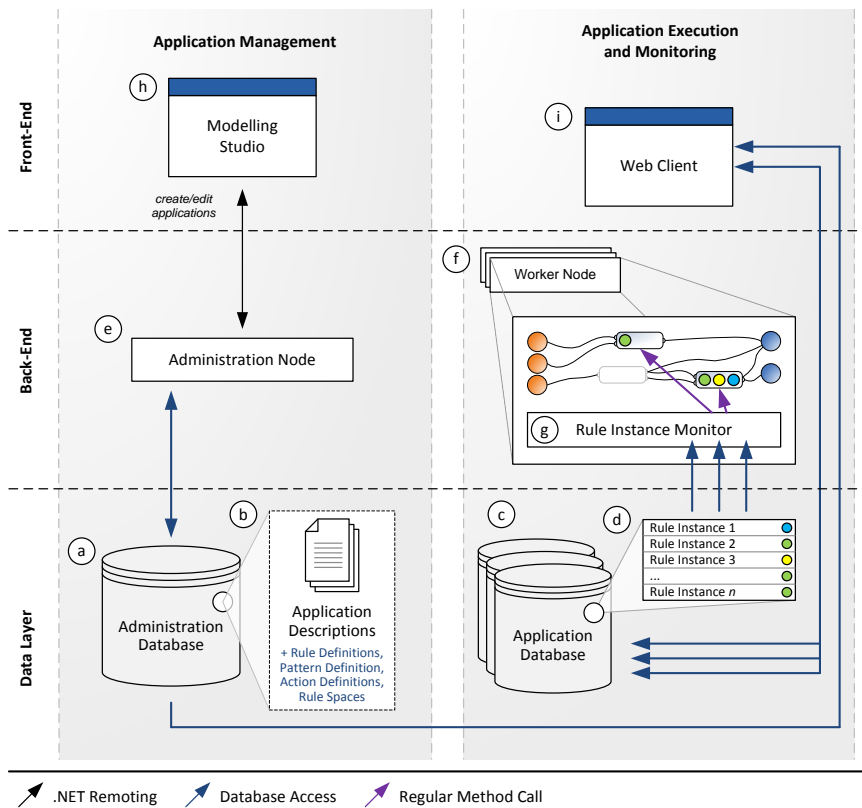


Figure 5.19. Implementation Architecture

### 5.7.1 Data Layer

On the data layer of SARI, a central *administration database* (Figure 5.19a.) and a collection of *application databases* (c.) contain all system-level and all application-specific data, respectively.

The key element of the administration database is the collection of *application descriptions* (b.), which are XML-formatted definitions of the static structure of all SARI applications of a system. From the proposed approaches to infrastructural and sense-and-respond rule management, all elements except of concrete sense-and-respond rules may be considered as static in that they are not changed during the run time of an application. In the proposed implementation, we therefore persist all *rule definitions* (along with the mappings to rule services), *pattern definitions*, *action definitions*, and *rule spaces* of a rule-management system as direct parts of the respective application description. As a consequence, changes in these elements require an entire redeployment of a SARI application. Consistency among the parts of an application description is guaranteed through higher-level elements such as the *administration node*.

For each SARI application of a system, a separate application database contains all data necessary for the execution of the application. In the extended architecture, we use the application database to store the various *sense-and-respond rules* of an application. Keeping sense-and-respond rules separate from the application description allows them to be updated at any point in time without having to redeploy the entire application. The key element of the extended schema is the “Rules” table (d.), which basically contains the most recent versions of all rules along with the rule spaces to which they belong. It is queried both by the various worker nodes of a system and the *web client*.

Figure 5.20 shows the full schema for sense-and-respond rules. The display name, description, owner, rule space, and activation mode of a rule are available as separate columns. The exact structure of a rule along with all input-parameter expressions, as well as the collection of *scheduling conditions*, are stored as XML-formatted texts. The version field is increased whenever a rule is updated and is used to detect (i) rule updates in the back-end-layer, and (ii) concurrent updates of multiple users. Similarly, a rule-space-level version field is updated at the “Rule Space” table whenever an update occurs among the rules of a rule space. This includes the creation of new rules as well as updates and deletions. The “Rule History” table eventually contains an entry for each update to a rule, including an XML-formatted description of the update, a reference to the responsible user, and a time stamp. s

The proposed schema refers to rule spaces, business-level building blocks, and templates via their identifiers to avoid redundancies among the application description and the application database. Listing 5.1 shows the XML-formatted definition of an exemplary rule “Host Provisioning HP1 - Transactions > 2 seconds” as discussed in Section 5.5.7.

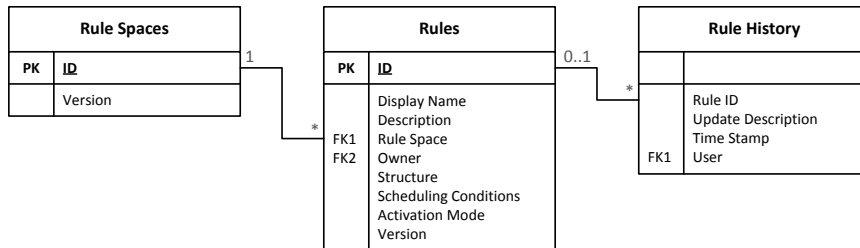


Figure 5.20. Sense-and-Respond Rule Schema

```

<RuleInstance xmlns="http://www.uc4.com/Decision/RuleInstance">
  <PatternActionInstance>
    <BusinessPattern identifier="DbLongRunningTransaction"/>
    <Bindings>
      <Binding identifier="1D7BE553">
        <BusinessPattern identifier="DbLongRunningTransaction">
          <Condition identifier="2E4FA01F" inverted="false">
            <Expression>
              <![CDATA[OUT_TransactionType <> $x]]>
            </Expression>
            <InputTemplate>
              <![CDATA[OUT_TransactionType} is different from ${x}]]>
            </InputTemplate>
            <InputParameters>
              <Parameter displayName="x" identifier="x">
                <Expression>
                  <![CDATA["COMMIT"]]>
                </Expression>
                <SingleValueType>
                  <RuntimeType type="System.String"/>
                </SingleValueType>
              </Parameter>
            </InputParameters>
          </Condition>
        </BusinessPattern>
        <BusinessAction identifier="StartUC4Object">
          <ParameterExpression identifier="ObjectName">
            <![CDATA["JOBP.INFSTR.PROVISIONHOST"]]>
          </ParameterExpression>
        </BusinessAction>
      </Binding>
    </Bindings>
  </PatternActionInstance>
</RuleInstance>
  
```

Listing 5.1. Exemplary Sense-and-Respond Rule Description

### 5.7.2 Back-End Layer

The back-end layer of the SARI architecture is constituted by a central *administration node* (e.) and a collection of *worker nodes* (f.).

The administration node serves as an interface between the application description of a SARI application and other elements of the architecture, including the various worker nodes and the Modeling Studio. When receiving an up-



dated version of an application description, it updates the SARI description in the application database and, depending on the defined deployment groups, deploys it on worker nodes. In the presented implementation architecture, updates to an application description have no immediate effect on the concrete sense-and-respond rules as stored in the respective application database, meaning that these data may become inconsistent. Table 5.5 summarizes possible sources of inconsistency.

| Inconsistency                       | Description  |
|-------------------------------------|--|
| Missing business actions            | One or more of a rule's business actions were deleted from the respective rule space.  |
| Missing business pattern            | A rule's business pattern was deleted from the respective rule space.  |
| Missing rule space                  | A rule's rule space was deleted entirely from the application description.   |
| Missing input-parameter expressions | One or more building blocks of a rule space were extended by unset input parameters.   |
| Illegal input-parameter expressions | One or more input-parameter expressions turned invalid, e.g., due to changes in the respective input parameter (to a differing data type), in the respective input-parameter configuration (to <i>non-editable</i> , with a differing default expression), or in the application's global resources (so that a referred constant is not available any more). |
| Deviations from template            | A (template-based) rule's template was changed so that the rule does not conform to the current version of the template.   |

**Table 5.5.** Sources of Sense-and-Respond Rule Inconsistency

In the present version of our rule-management system, we require inconsistencies to be explicitly resolved by end users. A detailed discussion on possible resolvment strategies is given in Section 5.9.

On the application-execution side of the SARI architecture, a collection of worker nodes is concerned with the execution of SARI applications in a distributed environment. While worker nodes extract the infrastructural rules of an application from the application description during deployment, sense-and-respond rules are retrieved from the underlying application database. During run time, worker nodes thereby need to support the “hot deployment” of sense-and-respond rules as discussed in Section 5.5.1: Whenever a rule is created, changed, or deleted, it must be retrieved from the application database and deployed/undeployed with little or no latency.

For each deployed SARI application, worker nodes therefore run a so-called *rule monitoring service* (g.), which continuously monitors the rule-related tables of the respective application database. If a rule is created or modified, the rule data are retrieved from the application database and a full-fledged run time object is assembled from the various building blocks as available at the application description. Invalid rules are ignored for event processing un-

til inconsistencies are resolved. On start up, the various rule services of an application register at the rule-monitoring service. After receiving the initial collection of sense-and-respond rules during registration, they are subsequently provided with updates in near real time.

Designed with configurability in mind, the proposed implementation allows arbitrary implementations of a rule-monitoring service: While the default implementations follow a pull-based approach by querying rule-related tables at regular time intervals, more sophisticated implementations could set up on database triggers or related concepts.

### 5.7.3 Front-End Layer

On the front-end layer of the extended architecture, two tools are involved in rule management: The *Modeling Studio* (h.) is used to define the static structures of a rule-management system. An easy-to-use web interface (i.) allows defining sense-and-respond rules in a business-user-friendly fashion.

The creation and management of rule definitions, action definitions, and pattern definitions, as well as the basic set up of rule spaces, is provided to system operators and solution designers as part of the already-known Modeling Studio. The Modeling Studio is currently also used by rule managers for the configuration of rule spaces, i.e., the creation of business-level building blocks, prepared bindings, and templates.<sup>15</sup> These users then have no access to any other part of a SARI application. Focusing on the aspects that are relevant for rule management, the Modeling Studio is discussed in greater detail in Section 5.8.

The *rule-management web client* enables business operators to administrate sense-and-respond rules in a way that fully abstracts from the event-based foundations of decision making. The web client retrieves application descriptions and user data directly from the administration database and retrieves the current set of sense-and-respond rules from the application database. Collaboration among remote business operators is supported through the above-described version field, which enables detecting parallel and potentially conflicting updates. The web client and its exact integration with the overall architecture is discussed in greater detail in Section 5.9.

## 5.8 Modeling Studio

In the proposed architecture, the Modeling Studio serves as a comprehensive, power-user oriented IDE for SARI application development. This includes the

<sup>15</sup> A tailored tool for rule managers is planned for future work.

creation and administration of rule definitions, pattern definitions, action definitions, and rule spaces. The Modeling Studio is used by system operators, solution designers as well as rule managers. While system operators and solution designers have full access to the various elements of a SARI application, system operators are authorized for the administration of business-level building blocks in predefined rule spaces only. In the following, we present the various editors provided for rule-management entities.

### 5.8.1 Pattern and Rule Definition Editor

Figure 5.21 shows the Modeling Studio’s pattern-definition editor, displaying an exemplary pattern definition “DB transaction duration check” as discussed in Section 5.5.3.

The pattern editor is structured as follows: In the *rule components toolbar* (Figure 5.21a.), action buttons allow adding condition components, time-based components, as well as signals to the pattern definition’s decision graph. The *input parameter control* (b.) and the *output parameter control* (c.) enable solution designers to define the input parameters and output parameters of a pattern definition. The *decision-graph panel* (d.) shows the actual decision graph of a rule definition. Here, users may position rule components and establish dependencies between them. In the *detail panel* (e.), tailored configuration panels are shown for all rule components currently selected in the above decision-graph panel. The ever-present “General” tab allows defining the correlation set and time window of a decision graph.

The Modeling Studio’s rule definition editor is similar to the pattern definition; however, it allows adding response-event actions instead of signals and does not provide any facilities for defining input and output parameters.

### 5.8.2 Action Definition Editor

Figure 5.22 shows the Modeling Studio’s action-definition editor, displaying an exemplary action definition “Start UC4 task” as discussed in Section 5.5.4. As with the afore-mentioned pattern editor, an *input-parameter control* (Figure 5.22a.) shows the collection of input parameters of the given action definition. In the *response-event template control* (b.), an event-type selector allows choosing the event type of the encapsulated response-event template, and event-attribute expressions can be defined.

### 5.8.3 Rule Space Editor

Figure 5.23 to Figure 5.25 show the Modeling Studio’s rule-space editor, displaying an exemplary rule space “Database protection monitoring” as dis-

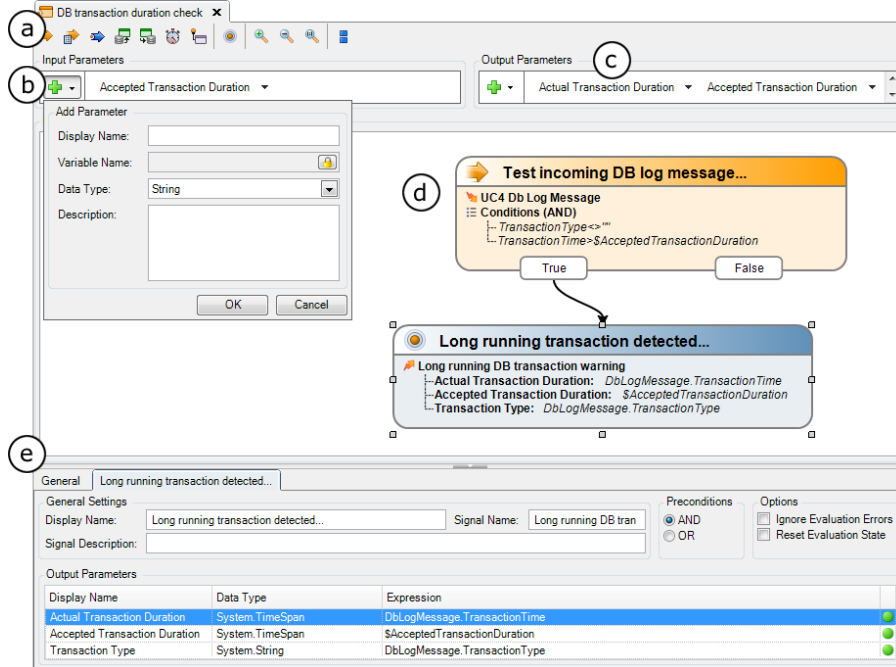


Figure 5.21. Pattern Definition Editor

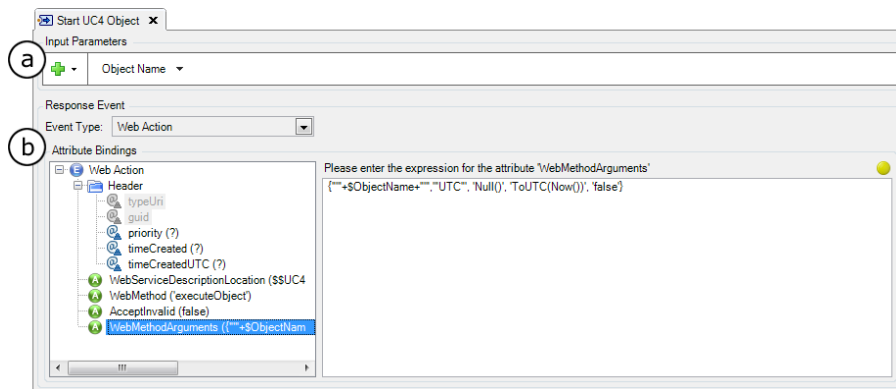


Figure 5.22. Action Definition Editor

cussed in Section 5.5.8. The editor is roughly separated by a central tabbed pane into pages for managing the pattern and action parts of a rule space, as well as its templates.

### Managing Pattern Definitions and Business Patterns

Figure 5.23 shows the pattern page of the rule-space editor. On the left-hand side, the *elements box* (Figure 5.23a.) shows all pattern definitions of the rule space along with the business patterns derived from them. Pattern definitions can be added to the rule space per drag and drop, or via a wizard-based dialog to be opened from the toolbar. The toolbar furthermore provides action buttons for creating new business patterns – again, opening a wizard-based dialog – or deleting the current selection. On the right-hand side of the editor, the *business pattern editor* allows editing the currently selected business pattern. The business pattern editor is structured as follows:

**Textual description editor (b.).** The user provides the textual description of the business pattern. In further consequence, this description is provided to business operators at the web client. Placeholders for input parameters are shown colored and can be added to the description via a drop-down list.

**Input-parameter configuration panel (c.).** The user defines the various input-parameter configurations of the business pattern. For each input parameter, he or she may specify a value, which can be marked as editable – i.e., as a default value – via the adjacent checkbox; if no value is specified, the input parameter is editable by definition. Additional controls allow specifying, editing, and removing the parameter’s validator, which is to be defined in a separate dialog depending on the input-parameters specific data type.

**Triggering signals editor (d.).** From a list of all signals of the underlying pattern definition, the user selects those signals that are relevant for the specified business pattern. By definition, at least one signal must be selected.

**Prepared bindings list (e.).** The user is provided all prepared bindings for the current business pattern, and may define new prepared bindings via a wizard-based dialog. Structured similar to the *template wizard* as discussed below, this dialog allows the user to select a business action and specify expressions for all input parameters of this business action.

### Managing Action Definitions and Business Action

Figure 5.24 shows the action page of the rule-space editor. It is generally equivalent to the pattern page, however, does not provide a triggering-signals editor. Prepared bindings are shown with both their business patterns and their business actions, meaning that a prepared binding  $pb = (ba, bp, \dots)$  is listed in the prepared bindings list of  $ba$  as well as  $bp$ .

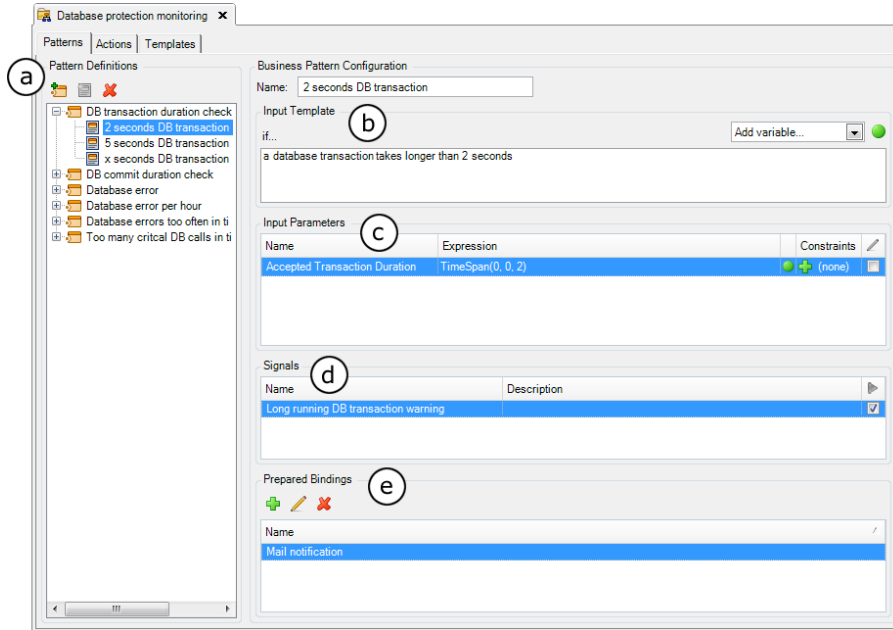


Figure 5.23. Pattern Page of the Rule Space Editor

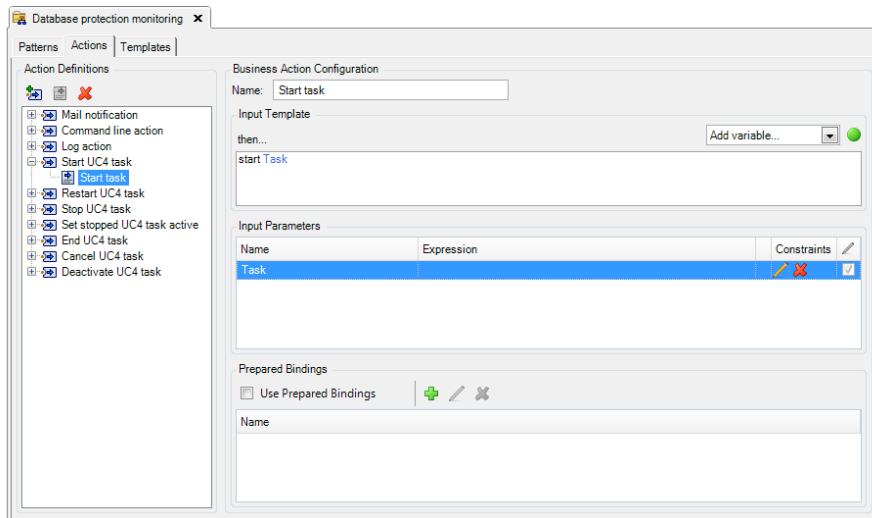


Figure 5.24. Action Page of the Rule Space Editor

## Managing Templates

Figure 5.25 shows the action page of the rule-space editor. On the left-hand side, the various templates of the rule space are shown in a list-based control. On the right-hand side, a preview of the currently selected template is provided. Users create and edit templates in a wizard-based dialog as shown in Figure 5.26: Based on a chosen business pattern, business actions, and conditions/exceptions may be added to a nesting of building blocks, and input-parameter expressions may be defined. For a detailed discussion of the used approach to rule assembling, the interested reader may refer to Section 5.9.4.

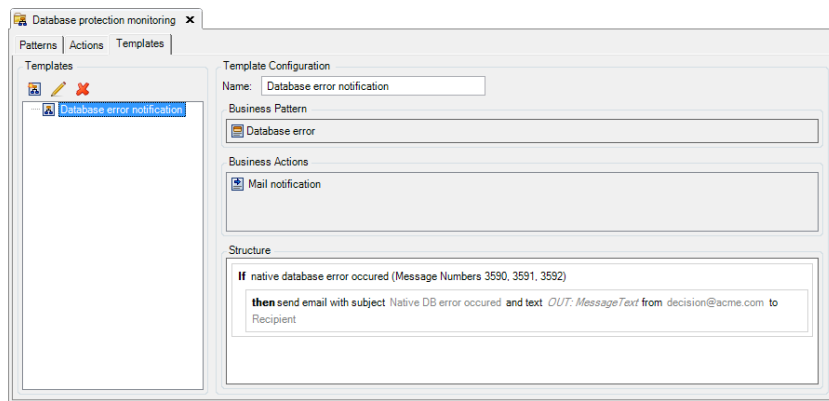


Figure 5.25. Templates Page of the Rule Space Editor

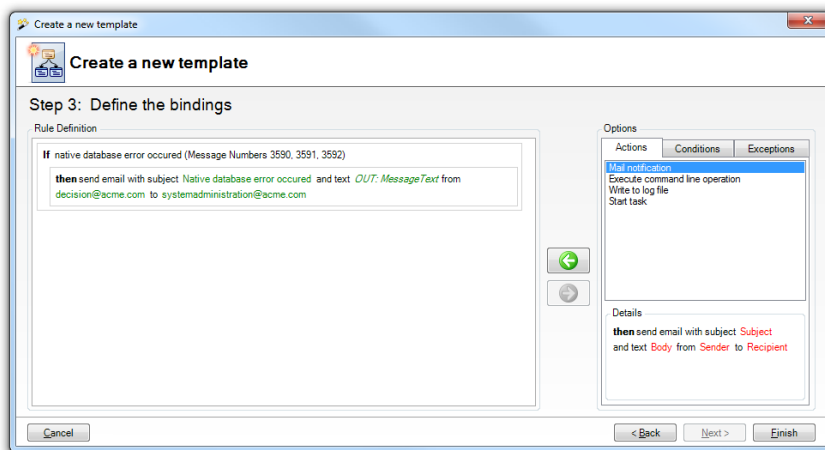


Figure 5.26. Template Creation Wizard

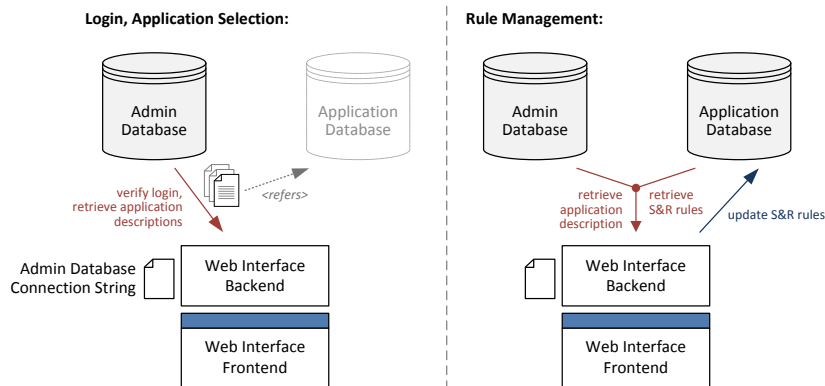
## 5.9 Web Client

The proposed architecture provides for a business-oriented, web-based user interface to enable sense-and-respond rule management in ways that fully abstract from the event-based foundations of decision making. In the course of our research, we fully implemented the *Decision Web Client* – shortly referred to as *web client* in the remainder of this thesis – as an extension to the existing collection of SARI front-end tools. It is based on the Google Web Toolkit (GWT) [46] and makes use of several other free-to-use third-party libraries both on the front-end side (e.g., to support “drag and drop”) and on the back-end side.

In the following, we present the integration of the web client with other parts of the overall SARI architecture (5.9.1) and discuss how the different rule-management tasks of a business operator can be accomplished. This includes rule monitoring (5.9.3) as well as different approaches to rule creation (5.9.4, 5.9.5) and resolving inconsistencies (5.9.6).

### 5.9.1 Integration

Figure 5.27 shows the integration of the web client with other elements of the overall SARI architecture.



**Figure 5.27.** Web Client Integration

When a user logs in to the system, the web interface connects to the system’s *administration database* with a connection string that is specified in a back-end-side configuration file. After validating the specified user credentials, the web interface retrieves all application descriptions and presents them to the user in a tree-based application-selection control. Having a SARI application



chosen by the user, the connection string for the respective *application database* is parsed from the respective application description.

In the following, whenever the user triggers an activity or the list of sense-and-respond rules is refreshed, the web interface connects with the administration database to retrieve the up-to-date version of the application description, and with the application database to retrieve the current set of sense-and-respond rules. When creating a new rule, the new data is written to the application database.

### 5.9.2 Interface Overview

Figure 5.28 shows the main page of the web interface. It is used by business operators to investigate the current set of sense-and-respond rules and serves as a starting point for creating, editing, and deleting sense-and-respond rules.

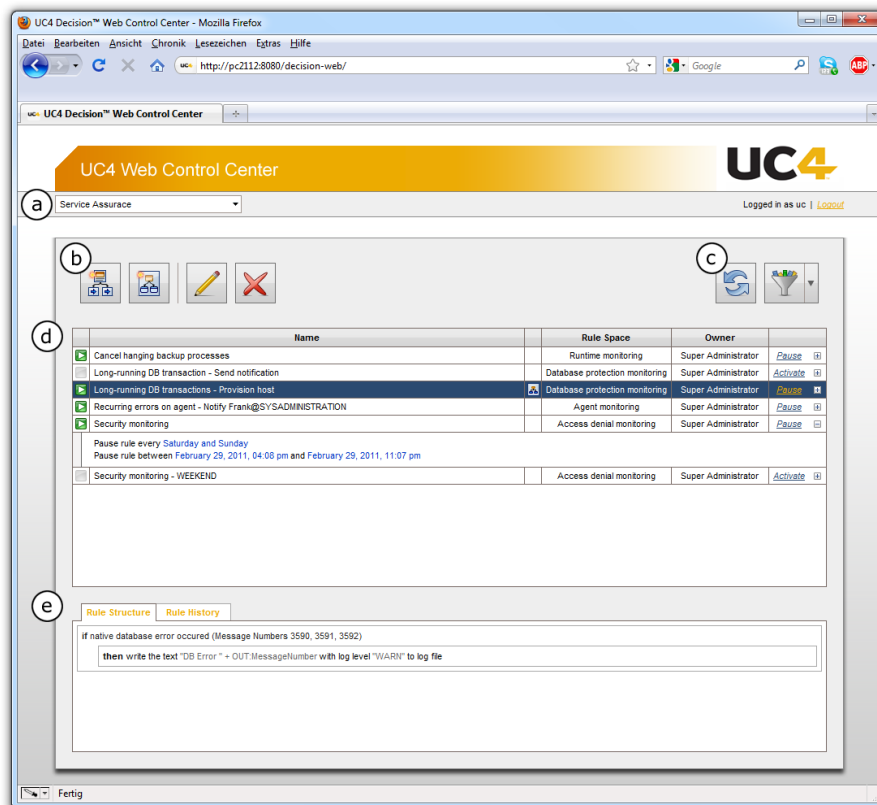


Figure 5.28. Web Client Overview

The *application selector* (Figure 5.28a.) allows business operators to switch between the various SARI application of the system through a tree-based drop-down control. If a user is authorized for one application only, the selector is entirely hidden. The *rule administration toolbar* (b.) contains action buttons for creating sense-and-respond rules from scratch or based on a template, editing rules, and deleting rules. Next to the rule administration toolbar, the *rule monitoring toolbar* (c.) allows refreshing the view of existing sense-and-respond rules as shown in the central *rule table* (d.), and filtering these rules through a drop-down control. The rule table lists all existing sense-and-respond rules of the current SARI application that (i) are visible to the logged-in user, and (ii) conform to the specified filter. The *detail panel* (e.) eventually shows the structure of a sense-and-respond rule along with its rule history; in case of an invalid rule, a detailed error message is shown.

### 5.9.3 Rule Monitoring

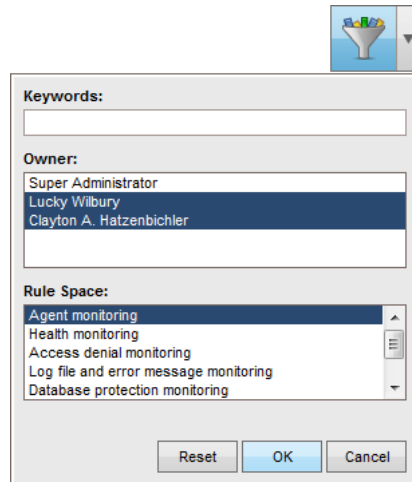
Rule monitoring – i.e., examining the overall collection of sense-and-respond rules along with their activation states and change histories – typically precedes any concrete rule-management action such as creating or removing a specific rule.

The key element for rule monitoring is the *rule table* on the web client’s main page as shown in Figure 5.28. It lists all sense-and-respond rules of the selected SARI application which the logged-in user is authorized to see. For each rule, the rule table shows the current activation state, the display name, an icon indicating whether the rule is based on a template or not, as well as the rule’s rule space and owner. A list entry can furthermore be expanded to show the collection of scheduling conditions, if available. A rule’s rule owner as well as its activation state can be changed directly in the rule table through respective drop-down controls. For any other change, the rule must explicitly be edited, which opens a wizard similar to rule creation as discussed below. If a rule is visible yet not editable to the logged-in user, it is shown grayed out. The rule table is automatically refreshed at regular time intervals; also, it may explicitly be refreshed by the user.

The filtering control as available in the rule monitoring toolbar allows restricting the set rules to be shown in the rule table. Filtering can be performed by rule space, by rule owner, or based on the display name of a rule. The dropped-down filtering control is shown in Figure 5.29.

### 5.9.4 Rule Creation from Scratch

The proposed approach to sense-and-respond rule management provides for two approaches to rule creation: Users may either create a rule *from scratch*



**Figure 5.29.** Dropped-Down Rule Filtering Control

– by selecting a business pattern and associating it with a collection business patterns according to the monitoring task that shall be implemented – or based on a predefined *template*. In the web client, both approaches are available through wizard-based workflows. In the following, we present the workflow for rule creation from scratch. Template-based rule creation is discussed in Section 5.9.5.

The presented workflow can generally be separated into four phases: *rule-space selection*, *pattern selection*, *rule assembling*, and *rule finalization*. A separate wizard page is provided for each phase.

### Rule Space Selection

Figure 5.30 shows the first page of the rule-creation wizard, where the user selects the concerned rule space from a drop-down list. Rule spaces that are not ready for rule creation – i.e., that lack *business-level building blocks* – are marked visually and cannot be selected. If a user is authorized for one rule space only, this step of the rule-creation wizard is skipped entirely.

### Pattern Selection

Figure 5.31 shows the second page of the rule-creation wizard, where the user is presented the various business patterns of the previously selected rule space in a list-based control. Showing the display name of the represented business pattern per default, a list item can be expanded to show the business pattern’s

The screenshot shows a web application interface for selecting a rule space. The top navigation bar includes the text 'UC4 Web Control Center' and the 'UC4' logo. A secondary bar indicates the user is 'Logged in as uc' with a 'Logout' link. The main window is titled 'Select Rule Space' and features a dropdown menu for 'Rule Space' currently set to 'Database protection monitoring'. A text area for 'Description' shows 'No description available.'. Navigation buttons at the bottom include '< Back', 'Next >', 'Finish', and 'Cancel'.

**Figure 5.30.** Rule Space Selection

description as well. When the user selects a business pattern, its textual representation as well as its output parameters (along with their data types and descriptions) are shown in separate boxes.

### Rule Assembling

Figure 5.32 shows the third page of the rule-creation wizard, where web-client users assemble and instantiate a full-fledged sense-and-respond rule based upon the previously-selected business pattern.

The wizard page can generally be separated into a left-hand side *rule structure panel* (Figure 5.32a.) and a right-hand side *element selector* (b.). The former shows the current structure of a sense-and-respond rule as a nesting of natural-language clauses as provided by business-level building blocks, conditions, and exceptions. It is initialized with the chosen business pattern, which – unlike any other block – cannot be removed from the nesting. The element selector provides tabs for business actions, conditions, and exceptions. The action selector lists all prepared business actions (here titled “suggested actions”), and – provided that the logged-in user is authorized to define non-prepared bindings – all non-prepared business actions (here titled “other actions”). Conditions and exceptions are generated automatically based on the output parameters of the chosen business pattern in a type-aware manner.

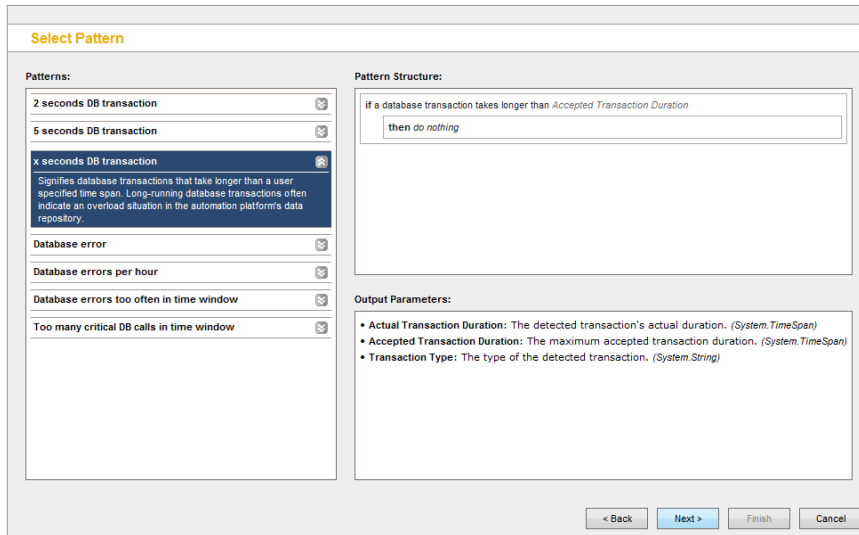


Figure 5.31. Pattern Selection

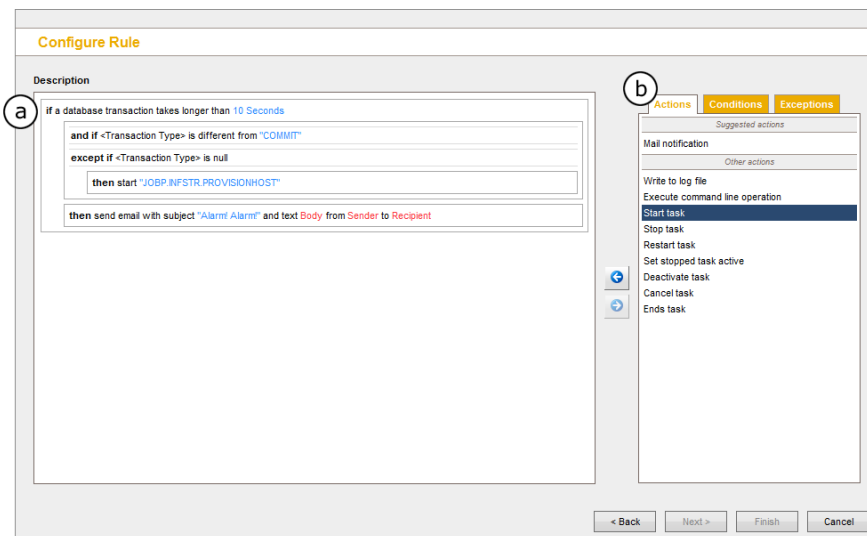
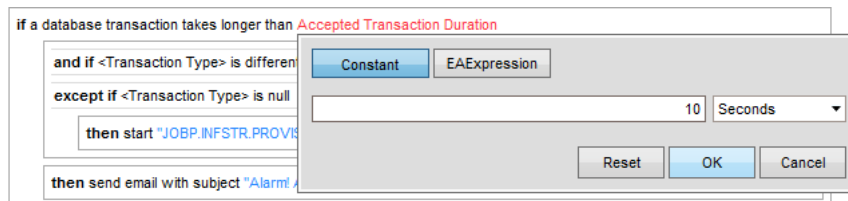


Figure 5.32. Rule Assembling

Web-client users may now add elements from the element selector to the rule structure. This can either be done via the “add” button, which adds an element depending on the present selections, or per “drag and drop”. In accordance with the sense-and-respond rule meta-model, each sub-nesting below the top-level represents a *binding*, which in turn is defined by exactly one business action and an arbitrary number of conditions/exceptions. Building blocks can be removed from the rule structure depending on the current selection via the “remove” button or by pressing “delete”.

In parallel, web-client users use the wizard page to specify concrete input-parameter values for all parts of the sense-and-respond rule. Placeholders for all previously unset input parameters are rendered as hyperlinks, showing either the display name of the input parameter or its current value in an easy-to-read formatting. Clicking such a hyperlink opens an *input parameter dialog* as depicted in Figure 5.33, through which an input parameter may be specified or reset to its default value. The web client generally provides four modes for specifying input-parameter values:

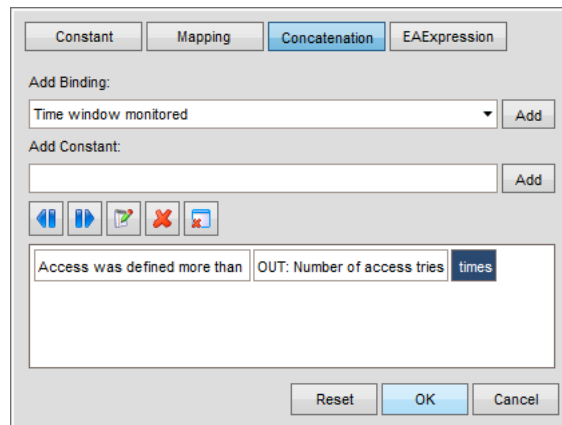


**Figure 5.33.** Input Parameter Dialog

- **Constant:** The user is provided a type-specific editor which allows specifying a constant value in the easiest possible manner. The type-specific editor for a date-typed input parameter value is shown in Figure 5.33. The constant mode is the only mode available when a validator is specified.
- **Output-parameter mapping:** The user is provided a list of all output parameters of the business pattern that are in a sub-type relationship to the selected input parameter. If chosen, such an output parameter is then directly *mapped* to the input parameter, meaning that for a triggering event situation, the calculated output-parameter value is used to dynamically adapt the associated reaction logic. Output-parameter mappings are available for the input parameters of business actions and conditions/exceptions only.
- **EA Expression:** Technically versed web-client users may specify EA Expressions directly, which provides a maximum of expressiveness and allows calculating input-parameter values from an arbitrary number of output pa-

rameters. The used EA Expression editor provides basic type-checking to reduce the risk of defining illegal expressions.

- **Concatenation:** EA Expressions provide great expressiveness, however, they may often be difficult to use for business operators. Available for string-typed input parameters only, the concatenation mode enables users to concatenate strings in a way that abstracts from the exact syntax of EA Expressions. It allows adding pieces of constant strings as well as output parameters to a list of substrings. Such substrings can then be removed and reordered freely. The concatenation control is shown in Figure 5.34.



**Figure 5.34.** Defining Input-Parameter Values in Concatenation Mode

The user may eventually proceed to the final step of the wizard when at least one business action is associated with the business pattern, and all input parameters of the rule are set.

### Rule Finalization

Figure 5.35 shows the fourth and final page of the rule-creation wizard. Having defined a full-fledged sense-and-respond rule in the previous steps of the workflow, the user is now asked to provide a display name and to specify the rule's initial activation state. In case of scheduled execution mode, an additional control allows specifying different kinds of scheduling conditions; in a separate, pop-up-like control, users may choose from the days of a week on which days a rule shall be paused, or specify a non-recurring pause interval by its exact start and end time. After finishing the rule-creation wizard, the so-defined sense-and-respond rule is eventually written to the application database, and in further consequence, enacted at the various worker nodes of the given SARI installation.

The screenshot shows a web client interface titled "Name Rule". It contains the following elements:

- Name:** A text input field containing "Handle long-running DB transactions".
- Select Activation Option:** A dropdown menu with "Scheduled" selected.
- Scheduling Options:** A list of conditions:
  - Pause rule every Saturday and Sunday (with edit and delete icons)
  - Pause rule between February 29, 2011, 04:08 pm and February 29, 2011, 11:07 pm (with edit and delete icons)
- Add new condition:** A green plus icon and a link.
- Navigation:** Buttons for "< Back", "Next >", "Finish", and "Cancel".

**Figure 5.35.** Rule Finalization

### 5.9.5 Rule Creation from Template

As with rule creation from scratch, the presented workflow can generally be separated into four phases: *rule-space selection*, *template selection*, *template instantiation*, and *rule finalization*. Again, a separate wizard page is provided for each phase. Rule-space selection and rule finalization are fully equivalent to rule creation from scratch, and thus are not handled in the course of this discussion.

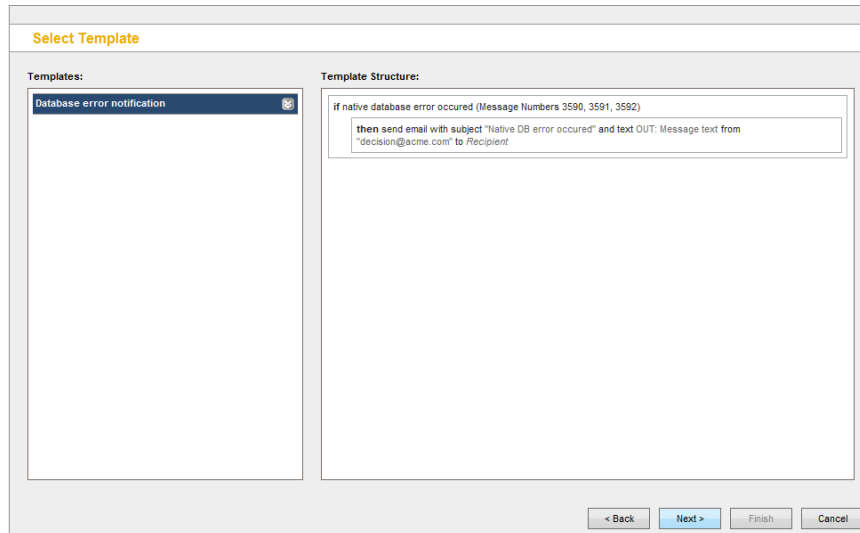
#### Template Selection

Figure 5.36 shows the second page of the rule-creation wizard. Having selected a rule space in the previous step, the web-client user is now presented the various templates of that rule space in a list-based control. When the user selects a template, its structure is shown in the right-hand side panel. The user confirms his selection by proceeding to the next wizard page.

#### Template Instantiation

Figure 5.37 shows the third page of the rule-creation wizard, where the user is asked to provide input-parameter values for all previously unset input parameters of the chosen template. The wizard page is generally equivalent to





**Figure 5.36.** Template Selection

the *rule assembling* step in rule creation from scratch; however, it is initialized with the template’s structure and does not allow adding or removing building blocks. Also, preset input-parameter values cannot be changed.

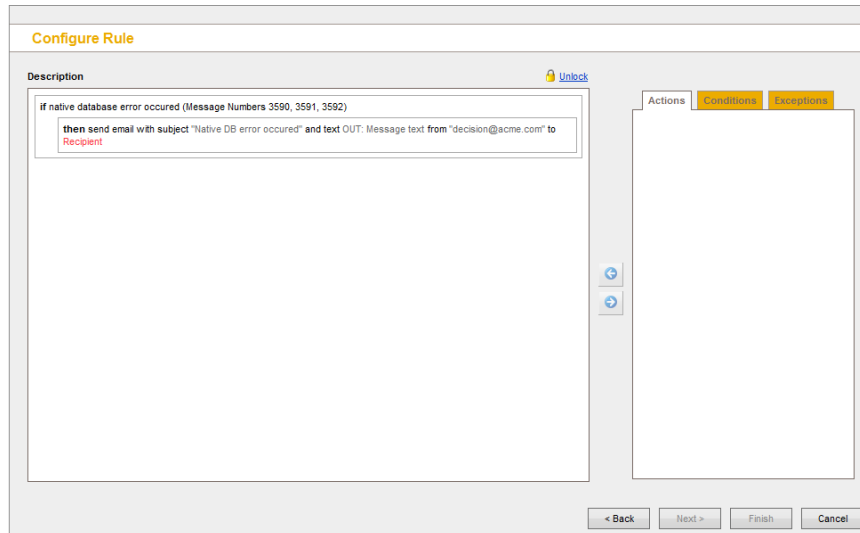
Available for authorized users only, an “unlock” button allows breaking a template. After using this button, the rule creation process changes to the from-scratch workflow and the rule structure can be adapted freely.

### 5.9.6 Handling Inconsistent Rules

In proposed implementation, updates to the application description of SARI architecture – including all pattern definitions, actions definitions, and rule spaces of an application – have no immediate effect on the set of sense-and-respond rules as stored in the respective application database. Changes in the static structure of a SARI application may therefore lead to inconsistent sense-and-respond rules, which are not further executed by the SARI back-end until resolved via the web client.

For the current version of the proposed rule management system, we provide for resolvment strategies as listed in Table 5.6.

In accordance with the presented listing, the web client generally separates between *repairable* and *non-repairable* rules; while repairable rules can be opened in the editing wizard and in this way repaired by the user, non-repairable rules can only be deleted. Depending on the so-defined degree of inconsistency, rules



**Figure 5.37.** Template Instantiation

| Inconsistency                       | Resolution Strategy  |
|-------------------------------------|--|
| Missing business actions            | The missing business actions must be removed from the concerned rule and, if necessary, replaced by new reaction logic.                                      |
| Missing business pattern            | The pattern part may be considered as the foundation of an event-pattern rule. In the proposed implementation, the concerned rule must therefore be deleted. |
| Missing rule space                  | As rules cannot be moved between rule spaces, the concerned rule must be deleted.  |
| Missing input-parameter expressions | Input-parameter expressions must be defined for all missing input parameters.  |
| Illegal input-parameter expressions | Input-parameter expressions must be redefined for all concerned input parameters.  |
| Deviations from template            | The concerned rule must be reconstructed to conform to its template.   |

**Table 5.6.** Sources of Sense-and-Respond Rule Inconsistency

are marked in the rule table with a “warning” symbol or an “error” symbol. In any case, a detailed description of the inconsistency is shown in the detail panel. After repairing and saving a rule, the update is detected by the SARI back-end and the now consistent rule is deployed as usual.

In the editing wizard, repairing inconsistent rules is supported by marking those elements that must be removed, changed, or added to the rule at hand. Figure 5.38 shows the rule-assembling wizard page for a rule deviating from its template.

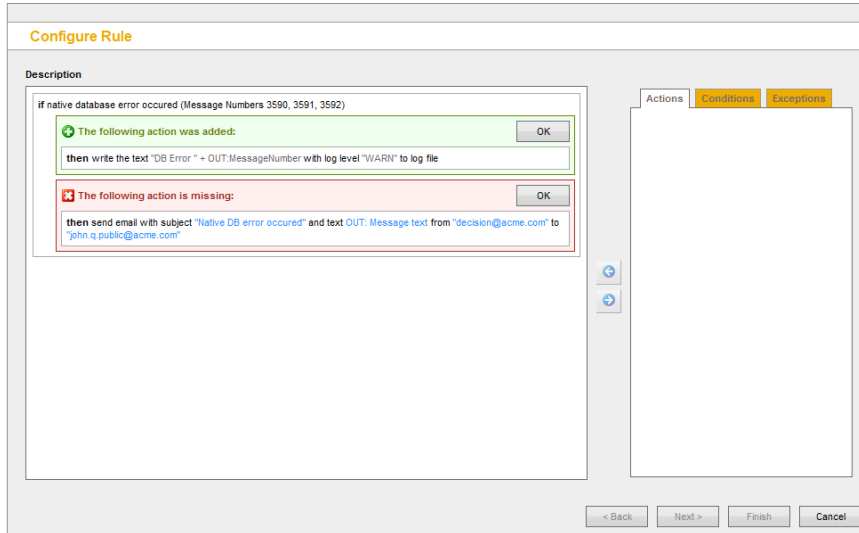


Figure 5.38. Repairing Inconsistent Rules

## Entity-Based State Management

**Abstract** Complex Event Processing using event-pattern rules has proved suitable for detecting noteworthy business situations of a defined length and structure. By contrast, challenges arise when the state of a complex, durable entity – e.g., a counter, a server, or a task queue – shall be derived from continuous streams of low-level updates. This presents a novel approach to state management for Complex Event Processing applications. We propose *business entity providers*, which encapsulate arbitrary state-calculation logic and manage state in the form of typed, application-wide data structures. Using a plug-in-based component model, business entity providers can be integrated into a Sense-and-Respond Infrastructure (SARI) application based on the specific requirements of a business scenario. We present extensions to SARI’s original correlation and decision-graph model that enable accessing business entities well-integrated with event-pattern detection. Our approach is demonstrated in a real-world scenario from the workload automation domain.<sup>1</sup>

### 6.1 Introduction

The presented approach to rule management builds upon the basic idea of (i) letting *infrastructural rules* establish an event-based image of the underlying source system, and (ii) applying *sense-and-respond rules* on this image to detect and handle noteworthy business situations. In many application scenarios that are centered around *business entities*, it may though be more natural – if not technically imperative – to detect such situations based on the aggregated state of these entities, rather than on continuous streams of state changes and actions related to them. In this chapter, we present a novel approach to state management for Complex Event Processing applications. It is seamlessly integrated with the basic event-processing facilities of Sense-and-Respond Infrastructure as discussed in Chapter 4 of this thesis and naturally complements the novel approach to rule management as discussed in Chapter 5.

---

<sup>1</sup> This chapter is based on the work of Obweiger et al. [91].

### 6.1.1 State Management in Complex Event Processing

Complex Event Processing based on event-pattern rules works particularly well for detecting noteworthy business situations of a defined length and structure, where the focus is on relationships between the involved events. By contrast, rule-based CEP faces significant challenges when the overall state of a complex, durable entity – e.g., a counter, a server, or a task queue – shall be derived from incremental, low-level updates of this state, and each update is represented by a (possibly complex) event. The state of such entities may, however, be of paramount importance for the monitoring of a system. Consider an example from the workload automation domain: Provided events of type “Task enqueued” and “Task started”, a system administrator might wish to be notified whenever the average sojourn time of a task queue exceeds a specified threshold.

We identified the following challenges for state-of-the-art, rule-based CEP:

- **Durable entity state.** Many CEP engines use sliding time windows to detect event patterns of a defined length and to limit the number of events which must be kept in memory. While this strategy is suitable for event-pattern detection, it generally contradicts with the idea of durable business entities. By contrast, entity data require a separate, non-volatile data-management layer that can be updated based on incoming events and takes into account the specific characteristics of the managed entities. Entity data, their development over time and possible relationships to event data may eventually provide valuable insights to the behavior of a system and should be available for ex-post analysis.
- **Complex state-calculation logic.** Calculating the overall state of an entity from low-level updates may be of considerable complexity; consider, for instance, the above calculation of average sojourn time from a series of “add” and “remove” operations. Implementing such logic directly within event-pattern rules easily bloats an application and makes it difficult to read and maintain. Calculation of entity state should instead be encapsulated and decoupled from end-user-defined business logic.
- **Active entity monitoring.** In many CEP frameworks, durable data can only be accessed at the occurrence of an event, in the “condition” or “action” part of a rule. Such event-driven access is perfectly useful when the state of an entity shall be tested at particular points in time, e.g., whenever an alarm occurs a source system. It is, however, impractical when entities shall be monitored *continuously* and noteworthy states shall be detected independent from the exact point in time in which they occur. To avoid a tight coupling between the updating and monitoring of business entities in such scenarios, a framework should enable rules to actively react to entity-level state changes, generally independent from the events and rules that originally caused these state changes.

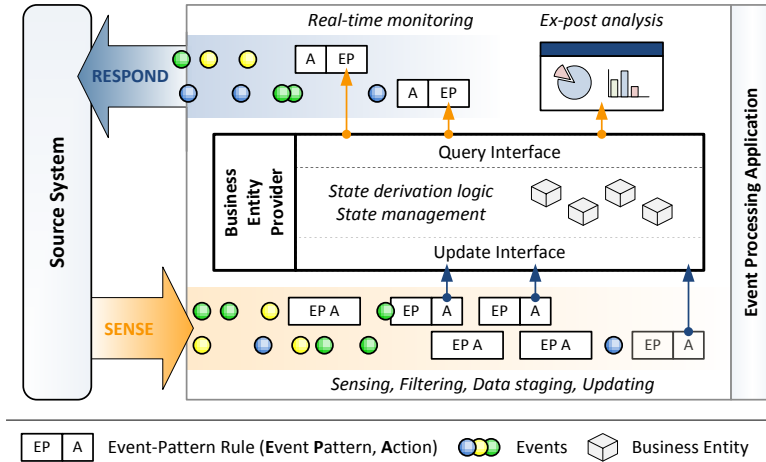
- **Context-aware data access.** Concepts such as *contexts* [4] and *correlation sets* (Section 3.4) have been developed to partition the overall set of events based on user-defined relationships between events. For the sake of consistency across event-pattern detection and state management, access to entity data should be integrated with contexts and related concepts, if such are available.
- **Ease of use.** As a general concern, the definition of application-specific state-management logic, the monitoring of business entities, as well as their integration with event-pattern detection should be oriented towards end users and require as little technical expertise as possible.

### 6.1.2 Business Entity Providers: An Architectural Overview

In this chapter, we present a novel approach to state management for Sense-and-Respond Infrastructure. We introduce *business entity providers*, which encapsulate custom state-calculation logic and manage virtual entities as system-wide, typed data structures. The presented architecture allows business entity providers to be implemented as plug-in-like components that can be incorporated depending on the specific requirements of a given business scenario. Within this architecture, plugged-in business entity providers expose easy-to-use interfaces for updating and querying entities to end-user-defined event-processing logic. Extensions to SARI's original correlation and decision-graph model enable accessing entities fully integrated with event-pattern detection.

Figure 6.1 shows the presented approach from a high-level perspective. On the “updating” side of a business entity provider, event-pattern rules are applied on the incoming stream of business occurrences for filtering, transforming, and aggregating events, as well as to detect events that signify updates to the state of an entity. In the action part of such a rule, the concerned business-entity instance is identified from the detected event pattern and the respective update operation is invoked on the business entity provider.

On the “monitoring” side, event-pattern rules may then use the query interface of a business entity provider to monitor business entity states for exceptional values, e.g., to test the load of a task queue against a specified threshold. The proposed architecture supports two access modes: On the one hand, event-pattern rules may run a query on demand, e.g., at the occurrence of an event. On the other hand, event-pattern rules may evaluate a condition continuously and react to exceptional states independent from the events that originally caused that update. Depending on the used data-management approach, a business entity provider may also provide historic entity data to tailored data mining and visualization tools for the ex-post analysis of a system.



**Figure 6.1.** A High-Level View on Business Entity Providers

Facilitating the integration of custom state calculation and management logic, the presented solution enables companies to exploit the benefits of CEP also in entity-centric business scenarios. These are difficult, if not impossible, to approach with purely event-based strategies. By exposing their functionality through easy-to-use interfaces, business entity providers simplify end-user-defined event-processing logic, which may now focus on event-pattern detection rather than on low-level calculations. The proposed architecture furthermore supports a clear separation of concerns between state updating and state monitoring: Updating rules can be defined with the general goal of keeping business entities up-to-date, independently from possible monitoring logic. Conversely, monitoring rules can focus on high-level decision making and may be added and removed without having to touch the low-level infrastructure of an application. The presented architecture outperforms problem-specific, ad-hoc solutions – e.g., a handmade event service concerned with the management of a particular kind of data – by providing full integration with the framework’s rule model independent from the concrete business-entity provider implementations in use.

### 6.1.3 Business Entities in SARI Rule Management

The presented approach to state management seamlessly integrates with the proposed differentiation of rule-based event-processing logic into *infrastructural rules* and *sense-and-respond rules* as discussed in Chapter 5. Following from that, it seamlessly integrates with the proposed approach to user-oriented rule management:

**Updating rules.** Pure updating rules – i.e., rules that are applied with the goal of keeping business entities up to date and in sync with possible real-world correspondances – prepare data for other parts of an application, but do not by themselves respond to the business environment. By definition, pure updating rules are therefore part of the *event-processing infrastructure* of a SARI application. As is it typical for infrastructural rules, updating rules require a detailed and comprehensive understanding of the event-processing application at hand and its integration with underlying source systems. Also, updating rules will typically remain relatively stable over time.

**Business entities.** Business entities that are kept up to date through a collection of updating rules and made available for arbitrary monitoring logic may be considered as part of the *event-based image* of an underlying business environment. Business entities therewith complement the preprocessed events of such an image, which describe the business environment on the level of discrete actions and state changes. Figure 6.2 sketches such business entities in a high-level perspective on SARI applications.

**Monitoring rules.** Pure monitoring rules – i.e., rules that use business entities to detecting exceptional business situations and trigger respective actions in the underlying business environment – do not prepare data for any other part of the application, but directly or indirectly, respond to the business environment. By definition, pure monitoring rules are therefore part of a SARI application’s *sense-and-respond layer* for event-driven control and decision making. As it is typical for this layer, monitoring rules require a deep knowledge of the underlying business environment and are subject to frequent change, e.g., whenever new risks or business opportunities are identified.

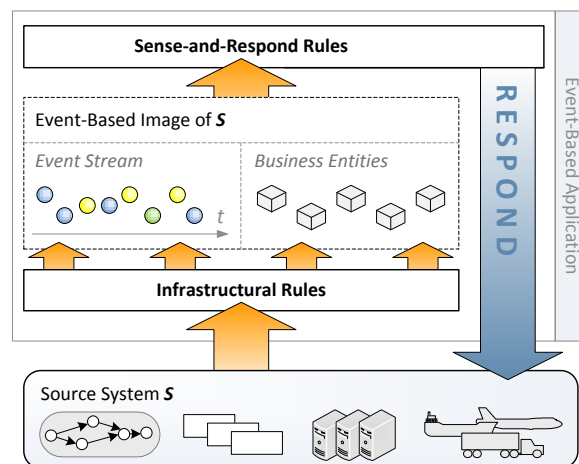


Figure 6.2. Infrastructural Rules vs. Sense-and-Respond Rules



For large-scale event-processing applications that require business logic to be (i) changed frequently, and/or (ii) managed by business users with limited technical skills, we suggest an overall architecture and application-development workflow that is structured as follows:

- **System operators** set up the event-processing infrastructure of an application to constantly update a set of business entities and keep them in sync with their real-world correspondences. System operations operate with the general goal of establishing virtual representations of all business entities of the underlying business environment that could be relevant for monitoring purposes, rather than with a specific monitoring tasks in mind.
- **Solution designers** and **rule managers** create collections of *pattern definitions* and *business patterns* based these business entities.
- **Business operators** assemble these business patterns with standard *business actions* to concrete sense-and-respond rules. It is not at all visible to business operators whether events, business entities, or both are used to model and detect the business situations they are interested in.

### Internal Business Entities

The above-described integration of business entities with the proposed rule-management framework facilitates end-user-oriented event processing also in entity-centric environments. It is essential to note, though, that the concept of business entities does not dictate this particular style of application development. In business environments that are

- (i) limited in scope,
- (ii) remain stable over time,
- (iii) are managed by users that are technically versed, or
- (iv) where from a large overall number of business entities only few need to be monitored at a point in time,

it may instead be more efficient to combine both the updating and the monitoring of business entities within one and the same decision graph. For example, when a certain reaction is required if, and only if, the size of a queue exceeds a constant threshold, logic for updating and monitoring this queue could well be expressed in a single infrastructural rule. Similar, if an online-gambling provider decides to observe user accounts on a selected basis, associated business entities could be updated and monitored within the pattern-detection part of sense-and-respond rules, where the user ID is passed as an input parameter. In the course of this chapter, we refer to business entities that are relevant within the scope of an individual decision graph as *internal business entities*.

### 6.1.4 SARI Application Model – Revisited

In Chapter 3 and 4 of this thesis, a model-driven view on SARI applications, splitting the overall complexity of an application into smaller, easier-to-understand sub-models, has been investigated. Figure 6.3 sketches the various sub-models of a SARI application, extended by a novel *business entity model*. The business entity model contains the list of plugged-in business entity providers along with the application-specific configurations of these providers. Most prominently, application-specific configurations consist of a number of *business entity types* (Section 6.3), describing the exact structure of a certain class of business entities to be managed. Within a SARI application, the business entity model is referenced by the *correlation model* as well as the *decision graph model*, both for the definition of pattern-detection logic (through *business entity conditions*, Section 6.6.2) and the definition of reaction logic (through *business entity actions*, Section 6.6.1).

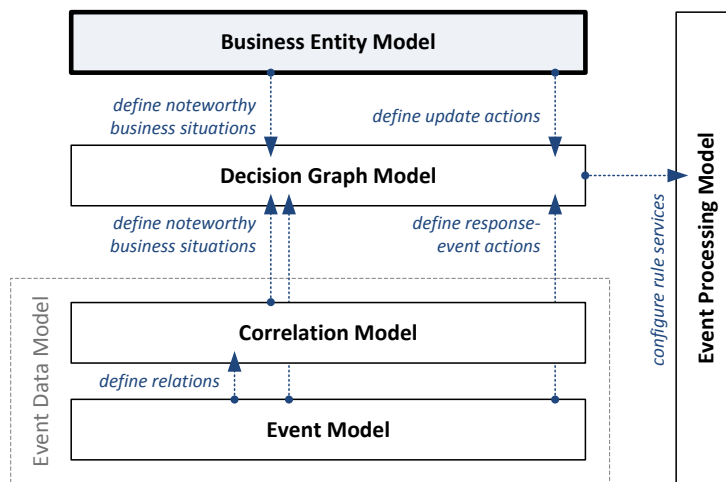


Figure 6.3. SARI Application Model, including the Business Entity Model

### 6.1.5 Outlook

The remainder of this chapter is structured as follows: In Section 6.2, we discuss related work. In Section 6.3 we present the meta model for business entity providers. Possible realizations of the described meta model are discussed in Section 6.4, where we introduce provider implementations for *scores*, *base entities*, and *sets*. Section 6.5 and Section 6.6 provide extensions to the original correlation model and decision-graph model as discussed in Chapter 3 and

Chapter 4 of this thesis. These extensions facilitate the integration of business entities with SARI’s approach to rule-based event processing. In Section 6.7, an experimental implementation of our approach is presented. We demonstrate our solution by a concrete example from the workload automation domain in Section 6.8 and conclude this chapter in Section 6.9.

## 6.2 Related Work

Etzion and Niblett [37] list *global state elements* as one of seven core building blocks for event-processing applications. In their terminology, *event processing state* refers to data that are available across event-processing agents and can be updated by an event-processing application. The authors suggest a model where event-processing state is managed by dedicated event-processing agents, which receive events from other agents and adapt the state based on incoming events’ event-attribute values. In contrast to the presented model, our approach enables updates to be triggered *directly* and *synchronously* from within event-pattern rules, which aims to simplify the overall event-processing logic of an application.

In inference-based approaches, reasoning about (complex) events is naturally integrated with reasoning about durable data, with both kinds of data being managed as *facts* in working memory. Logic-based approaches based on Event Calculus [64] and its variants (e.g., [96]) use *fluents* to model entity state. In the words of Shanahan [119], a fluent represents “anything whose value is subject to change over time”. In the following, we focus on approaches that perform event-pattern detection directly on volatile (streams of) events and do not per se set up on an underlying knowledge base.

In stream-oriented event processing, *query tables* [122] and related concepts have been introduced to make durable data joinable with event data and updatable within event-stream queries. While query tables provide means for persistent data management, the calculation of entity state must still be defined as part of a rule, using the framework’s Event Processing Language (EPL). *FlexStreams* [126] allow incorporating procedural, potentially stateful logic as part of a query. Kozlenkov et al. [60] present a context-aware approach to state management that builds upon a separation between stream processing and inference-based state management: Whenever an event changes the state of a specific context in which it occurs, the resulting state changes are forwarded to the event-processing engine, where further patterns may be detected. In both approaches, state management is well encapsulated and allows users to define event-processing logic in a decoupled manner. However, to our best knowledge, both approaches dictate a certain style of programming (*procedural* vs. *inference-based*). As a consequence, their usefulness could vary depending on the given application scenario and the data to be managed. By contrast, the

proposed state-management framework enables application developers to integrate arbitrary business-entity provider implementations. These providers may, in fact, be based on procedural or inference-based programming techniques, but could equally adhere to any other programming paradigm that is supported by the underlying implementation framework.

Several rule-based event-processing frameworks allow accessing persistent data sources – e.g., Web data such as XML or RDF [18], or databases – in the “condition” and “action” part of a rule. When using passive data storages such as files or non-active databases, entity state must be calculated directly within rules and explicitly be retrieved in order to be monitored. *Active object databases* [100] allow encapsulating durable data along with functions for updating these data, and furthermore provide means for active entity monitoring. Rules in active databases are, however, internally-oriented (i.e., limited to events that occur within the underlying database) and are typically defined with a global scope (cf. [98]). Being identifiable via (possibly composite) keys and accessible through easy-to-use interfaces, the proposed concept of business entities is, in some respects, comparable to objects in (active) object databases. In contrast to these systems, our framework provides an additional layer of abstraction that allows incorporating arbitrary state-calculation logic and data-management strategies. In addition, a type model is provided that enables end users to easily configure plugged-in business entity providers based on the specific requirements of an application scenario.

Kellner and Fiege [63] present the separation of two viewpoints in a CEP application based on Key Performance Indicators (KPIs). In the *derivation* viewpoint, KPIs are derived from lower-level events through filtering and aggregation rules. In the *interpretation* viewpoint, relevant business situations are modeled based on target values for these KPIs. Our approach generalizes the concept from KPIs to arbitrary kinds of business entities and is fully integrated with the rule model of SARI: Using the rule model, business situations can be defined based on event patterns, exceptional entity states, or both. Similar to the authors’ work, we provide for a strict decoupling between updating and monitoring so that each aspect can be handled independent from the other.

The concept of business entities (also: business artifacts, adaptive documents) as typed, identifiable and globally-accessible data has successfully been applied in the context of Business Process Management (BPM) (e.g., [28, 65, 86]). Nandi et al. [85] present the *Business Entity Definition Language* (BEDL), which is intended to evolve into a standard for modeling business entities. In BEDL, a business entity type consists of an information model (defining the attributes of an entity), a lifecycle model (defining the possible states of an entity as a finite-state-machine), access policies (defining what roles have authority to modify data and cause state transitions) and an event model (providing notifications of state and data changes to subscribers). Given a definition of one

or more business entity types, a business entity runtime then provides access to so-defined entity instances to business processes, e.g., defined in an extension to WS-BPEL [95]. Similar, business entity providers manage data based on user-defined business-entity types and provide interfaces for updating, querying, and monitoring data through event-pattern rules. In contrast to BEDL, business entities do not provide a inherent event model, but allow monitoring rules to formulate and run arbitrary queries based on a set of *query properties*.<sup>2</sup> State-machine-based lifecycles of business entities are not an integral part of the proposed framework, but could be provided by an entity-provider implementation.

### 6.3 Meta Model

The presented framework is designed as a generic state-management layer that can be equipped with plug-in-like business entity providers depending on the requirements of a given use case. Each business entity provider encapsulates state-management logic for a certain “kind” of entity; a business entity provider could, for instance, be provided for *queues* as discussed in Section 6.1. Apart from a basic structure prescribed by the framework, this state-management logic can be defined freely by the implementer of a business entity provider – e.g., using OOP and in-memory data management, SQL statements, or any other suitable approach – and optimized with respect to the managed data.

Figure 6.4 sketches the meta model for business entity providers, defining the basic structure to which a business entity provider must adhere in the proposed architecture. The meta model is roughly separated into a non-empty collection of *business entity types*, a collection of *business entities* conforming to these types, as well as collections of *update* and *query interfaces* for accessing business entities. All other elements of a business entity provider are implementation-specific and generally invisible to the end users of a system.

#### Business Entity Types

While the basic semantics of a business entity provider are given through its implementation, the proposed framework allows tailoring a provider to the specific application in which it is used through a non-empty collection of

<sup>2</sup> While an inherent event model has been investigated for business entities in SARI – here, the designer of an entity type could specify noteworthy situations and a rule could register to such a situation – it showed to be too restrictive for practical use cases. In particular, the approach would conflict with the intended decoupling between modeling/updating and monitoring business entities.

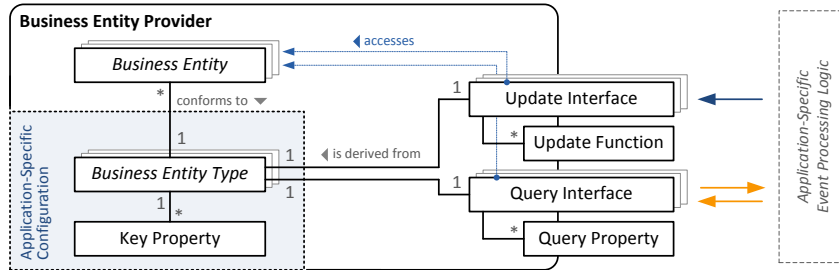


Figure 6.4. Business Entity Provider Meta-Model

*business entity types*. Each business entity type specifies the exact structure of a certain class of business entities to be managed. For example, given a business entity provider for queues, a business entity type could describe a class “Task queue” of queues, with respective characteristics. Albeit business entity types are generally provider-specific – see Section 6.4 for concrete examples – the proposed architecture requires all business entity types to define a (possibly composite) key, through a non-empty collection of *key properties*  $K$ . Each key property  $k = (i, t) \in K$  is defined by an identifier  $i$  and a primitive data type  $t$ . At run time, a tuple of key-property values can then be used to identify a specific instance of that type, e.g., when updating or querying its state.

Apart from business entity types, business entity providers may require further, provider-specific configurations for the various SARI applications in which they are used. For example, a business entity provider that sets up on an RDBMS for persistent data management may require the application developer to specify a valid database connection string, etc.

The overall collection of plugged-in business entity providers, along with their business entity types and further application-specific configurations, forms the *business entity model* of a SARI application as sketched in Section 6.1. The business entity model is referenced by the extended *correlation model* as well as the extended *decision graph model* for the definition of entity-aware pattern-detection and reaction logic.

## Business Entities

During run time, business entity providers manage state as collections of business entities. Each business entity  $e : T$ ,  $e = (V_K, \dots)$ , conforms to exactly one business entity type  $T = (K, \dots)$  and is uniquely identified by a tuple of key-property values  $V_K$  for all key properties in  $K$ . By definition, the key of a business entity is *immutable*, i.e., it must remain constant throughout its lifecycle.

### Update and Query Interfaces

Business entity providers provide access to the managed business entities through easy-to-use *update* and *query interfaces*. In the proposed architecture, these interfaces are exposed *per business entity type*. A caller must therefore specify the concerned business entity type in order to access an interface.

The update interface for a business entity type  $T = (K, \dots)$  is defined by a non-empty collection of *update functions* for creating, modifying, and destroying  $T$ -typed business entities. By definition, an update function takes as input

- a non-empty collection of *key property values*  $V_K$ , identifying the entity to which the update shall apply, and
- a collection of *function parameters*  $V_f$ , further specifying the demanded update operation.

The query interface for a business entity type  $T = (K, \dots)$  provides access to the managed entities through a non-empty collection of typed *query properties*. Provided the unique key of an entity, these properties may be used to define conditions on the current state of that entity; for example, a rule could test if the query property “Size” of a queue is greater than 100. Query properties are typically provided for all type-specific and type-independent<sup>3</sup> properties of  $T$ -typed entities. In addition, query properties may be available for aggregates and meta information such as the last update time stamp.

## 6.4 Exemplary Business Entity Providers

Based on the above meta model, the proposed framework enables application developers to integrate arbitrary business-entity provider implementations depending on the specific problems that need to be solved. In the following, we illustrate the presented model by the example of three commonly required kinds of entities: *Scores*, *base entities*, and *sets*. Business entity providers for these entities have been implemented as part of our prototype (see Section 6.7 for further details) and showed to be useful in many practical application scenarios.

In their type model, all of the following examples provide

- (i) a Boolean *history* property, indicating whether the provider shall maintain the complete history of an entity, as well as

---

<sup>3</sup> To avoid naming conflicts among type-specific and type-independent properties, a provider-implementation may mark selected terms as *reserved*, which then cannot be used in a business-entity type.

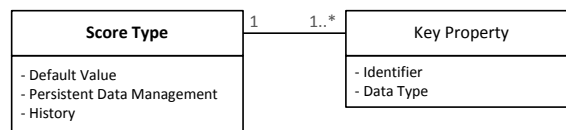
- (ii) a Boolean *persistence* property, indicating whether entity states shall be kept persistent in a database or managed in memory only.

### 6.4.1 Scores

Scores associate a tuple of key properties with a single, numeric *score value*. Through the provider’s update interface, this value can be set, incremented, and decremented. Query properties are available for the current score value and a collection of “moving” aggregates, such as the moving average and median. Albeit simple, scores form the basis for concepts such as counters and Key Performance Indicators (KPIs) and are part of almost any practical SARI installation. Monitoring rules are typically used to detect scores that are above or below a certain threshold.

#### Meta Model

Figure 6.5 shows the meta model for score types. A score type  $s = (K, d, a)$  is defined by a non-empty collection of *key properties*  $K$ , a numeric *default value*  $d \in \mathbb{N}$  and an optional *aggregation configuration*  $a$ . In accordance with the base entity-type model as presented in Section 6.3, key properties define the (possibly composite) identifier of a score instance. The default value defines the initial value of a newly-created score instance; also, an update function is provided to explicitly reset a score to its initial value. The aggregation configuration specifies either a number of updates or a time span over which moving aggregates shall be calculated from progressing score values.<sup>4</sup> If no aggregation configuration is provided, respective query properties for moving averages are not available.



**Figure 6.5.** Score Type Meta-Model

<sup>4</sup> As an example for *time-window-based aggregates*, consider a use case from ITSM where average server utilizations shall be calculated over time windows of  $n$  minutes independent of the number of update events within these time windows. As an example for *update-based aggregates*, consider a use case from online gambling where a user’s average bet amount shall be calculated from the last  $m$  bets independent of the times at which these bets took place.



## Update Interface

Table 6.1 lists the update functions for scores of a score type  $S = (K, d, a)$ . Note that neither score providers nor any of the following examples offer separate update functions for creating new business entity instances; instances are instead created implicitly, with the first call of an update operation for a non-existent instance. For example, when increasing a not-yet-existent score instance of a score type  $S = (K, d, a)$  by a value  $v$ , it is in fact instantiated with a score value of  $d + v$ .

| Function         | Parameters                 | Summary   |
|------------------|----------------------------|---|
| <i>set</i>       | <i>value</i> (Double)      | Sets the score to the specified value.  |
| <i>reset</i>     | –                          | Resets the score to the score type’s default value.   |
| <i>increment</i> | <i>addend</i> (Double)     | Increments the score by the specified value.  |
| <i>decrement</i> | <i>subtrahend</i> (Double) | Decrements the score by the specified value.  |
| <i>destroy</i>   | –                          | Removes the score instance, meaning that the score instance is not considered in subsequent queries and would be re-initialized with the score type’s default value in case of an update operation. Destroying a score does not affect a possible score history, which remains available for analysis purposes. |

**Table 6.1.** Update Functions for Scores

## Query Interface

Table 6.2 lists the query properties for scores of a score type  $S = (K, d, a)$ . Further query properties are available for all key properties in  $K$ . Moving aggregates – i.e., *MovingAverage*, *TimeWeightedMovingAverage*, *MovingMedian*, *MovingMinimum* and *MovingMaximum* – are available only if an aggregation configuration is provided.

## Example

Figure 6.6 shows an exemplary score type “Alarms per Server and Job Type” with two string-typed score properties “Server” and “Job Type”. Score values are initialized with a default value of zero, aggregated over a time span of 60 minutes, and kept persistent. Score histories shall be maintained.

### 6.4.2 Base Entities

Base entities associate a tuple of key properties with a collection of *entity properties*, which can be updated and queried via the provider’s interfaces.

| Property                         | Type      | Summary   |
|----------------------------------|-----------|---|
| <i>Value</i>                     | Double    | The score's current value.  |
| <i>LastUpdate</i>                | TimeStamp | The time stamp of the last score update.  |
| <i>MovingAverage</i>             | Double    | The moving average over the progressing score value according to the aggregation configuration <i>a</i> . |
| <i>TimeWeightedMovingAverage</i> | Double    | The time-weighted moving average over the progressing score value according to <i>a</i> .                 |
| <i>MovingMedian</i>              | Double    | The moving median over the progressing score value according to <i>a</i> .                                |
| <i>MovingMinimum</i>             | Double    | The moving minimum over the progressing score value according to <i>a</i> .                               |
| <i>MovingMaximum</i>             | Double    | The moving maximum over the progressing score value according to <i>a</i> .                               |

Table 6.2. Query Properties for Scores

| Alarms per Server and Job Type |                                     |
|--------------------------------|-------------------------------------|
| Server                         | [String]                            |
| Job Type                       | [String]                            |
| Default Value                  | 0                                   |
| Aggregation Configuration      | 60s                                 |
| Store History                  | <input checked="" type="checkbox"/> |
| Persistent Data Management     | <input checked="" type="checkbox"/> |

Figure 6.6. Exemplary Score Type

Base entities are typically used as virtual representations of complex real-world entities such as customers or products: Whenever an event indicates an update to a real-world entity, this update is “mirrored” to the respective base-entity instance. Monitoring rules are typically used to detect exceptional events based on the current state of a related entity; for instance, a rule could trigger an alert if a delay is signified for an order of a customer with the property “Rating” set to “Premium”.

### Meta Model

Figure 6.7 shows the meta model for business object types. A business object type  $B = (K, P)$  is defined by a collection of *key properties*  $K$  and a collection of *entity properties*  $P$ . An entity property  $p = (i, t, d) \in P$  is defined by an identifier  $i$ , a primitive data type  $t$ , and a default value  $d : t$ . The default value defines the initial value of an entity property in a newly-created base-entity instance.

### Update Interface

Table 6.3 lists the update functions for base entities.

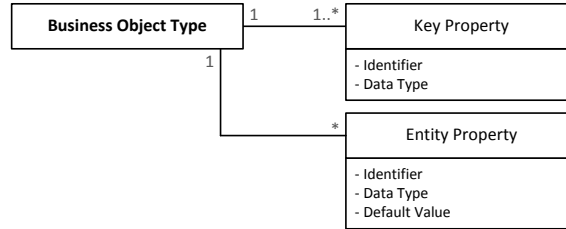


Figure 6.7. Base Entity Type Meta-Model

| Function                  | Parameters                | Summary  |
|---------------------------|---------------------------|--|
| <i>set</i> < <i>i</i> >   | <i>value</i> ( <i>t</i> ) | E.g., <i>setFirstName</i> ; sets the specified property $p = (i, t)$ .                         |
| <i>reset</i> < <i>i</i> > | -                         | E.g., <i>resetFirstName</i> ; resets the specified property $p = (i, t)$ to its default value. |
| <i>reset</i>              | -                         | Resets all properties.   |
| <i>destroy</i>            | -                         | Removes the entity.  |

Table 6.3. Update Functions for Base Entities

### Query Interface

The proposed base-entity provider exposes querying properties for all key and entity properties of a concerned base entity type, returning the (current) value of such a property for a given base-entity instance. The last update time (*Last-Update*) is provided as a type-independent query property.

### Example

Figure 6.8 shows an exemplary base-entity type “Customer”, where the represented individual’s unique social security number is used as a key property. Further characteristics are available as entity properties.

| Customer                          |                                     |
|-----------------------------------|-------------------------------------|
| <b>Social Security Number</b>     | [String]                            |
| <b>Second Name</b>                | [String] <i>null</i>                |
| <b>First Name</b>                 | [String] <i>null</i>                |
| <b>Date of Birth</b>              | [DateTime] <i>null</i>              |
| <b>Customer Category</b>          | [String] „Default“                  |
| <b>Store History</b>              | <input checked="" type="checkbox"/> |
| <b>Persistent Data Management</b> | <input checked="" type="checkbox"/> |

Figure 6.8. Exemplary Base Entity Type

### 6.4.3 Sets

Sets are an extension to above-described *base entities* that allow modeling collection data such as FIFO queues, priority queues, or stacks. Besides grouping a number of *entity properties*, sets act as a container for multi-variate *set elements*, which by themselves are defined by a *set element identifier* and a collection of *set element properties*.<sup>5</sup> Sets are particularly useful for *queue monitoring*, which is a key task in domains such as Service Level Management (SLM). For example, rules could be used to detect overflow situations and automatically request additional resources in response.

#### Meta Model

Figure 6.9 shows the meta model for set types. A set type  $S = (K, P, E)$  is defined by a non-empty collection of *key properties*  $K$ , a collection of *set properties*  $P$ , and a *set element type*  $E$ . Set properties are properties of the a instance itself and would typically include meta information such as the maximum capacity of a buffer, the guaranteed maximum waiting time in a job queue, etc.<sup>6</sup> The set element type defines the structure of elements to be contained in a set of type  $S$ . A set element type  $E = (K_E, P_E)$  is defined by a non-empty collection of set-element key properties  $K_E$  and a collection of set-element properties  $P_E$ . Set-element key properties are used to identify a set element when it is inserted and removed from a set. Set-element object properties further specify a set element and may provide additional information when analyzing the content of a set through the provider’s query interface.

#### Update Interface

Building upon the above-described base-entity provider, the proposed set provider offers all basic update functions for setting and re-setting set properties as listed in Table 6.4. Set-specific update functions are listed in Table 6.4.

---

<sup>5</sup> We refer to the entity properties of a set as *set properties* in the remainder of this thesis.

<sup>6</sup> It is essential to note that set properties are purely descriptive and have no impact on the actual behavior of a set provider. For example, having a property “Capacity” set to 1000 would not per se prevent a rule from inserting  $n > 1000$  elements to that set. A monitoring rule could, however, test the actual size of the set against the current value of the “Capacity” property and signify an overflow situation if necessary.

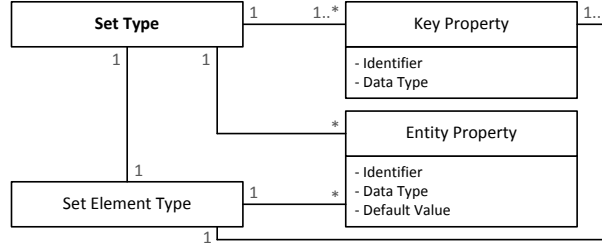


Figure 6.9. Queue Type Meta-Model

| Function      | Parameters  | Summary  |
|---------------|---|--|
| <i>insert</i> | <i>keyProperties</i> (Object[])<br><i>properties</i> (Object[]) | Inserts a new item with the specified key and object properties. No changes if the so-identified queue item is already queued. |
| <i>remove</i> | <i>keyProperties</i> (Object[])                                 | Removes the specified item from the queue. No changes if the so-identified item is not in the queue.                           |

Table 6.4. Update Functions for Sets

### Query Interface

As with base entities, the proposed set provider exposes query properties for all key and set properties of a concerned set type  $S = (K, P, E)$ . Set-specific query properties are listed in Table 6.5.

| Property                  | Type             | Description   |
|---------------------------|------------------|---|
| <i>Elements</i>           | List<SetElement> | A typed list of data records describing the current content of a set. List items expose properties for all set-element (key) properties in $E$ as well as the exact time stamp at which an item was added to the set. |
| <i>MeanSojournTime</i>    | TimeSpan         | The mean sojourn time among the current set of elements; zero in case of an empty set.  |
| <i>MinimumSojournTime</i> | TimeSpan         | The minimum sojourn time among the current set of elements; zero in case of an empty set.   |
| <i>MaximumSojournTime</i> | TimeSpan         | The maximum sojourn time among the current set of elements; zero in case of an empty set.   |

Table 6.5. Querying Properties for Sets

### Example

Figure 6.10 shows an exemplary set type “Task Queue”, where a string-typed queue ID is used as a key property and an integer-typed configuration “Capacity” is available as a set property. The set’s content – representing tasks as executed by an underlying automation platform – is defined by a unique task ID and set-element properties for a task’s type, owner, and priority.

| Task Queue                 |                                     |
|----------------------------|-------------------------------------|
| Queue ID                   | [String]                            |
| Capacity                   | Integer 300                         |
| Store History              | <input checked="" type="checkbox"/> |
| Persistent Data Management | <input checked="" type="checkbox"/> |
| Task ID                    | [Integer]                           |
| Task Type                  | [String] <i>null</i>                |
| Owner ID                   | [Integer] <i>null</i>               |
| Priority                   | [Integer] <i>null</i>               |
| ...                        | ...                                 |

Figure 6.10. Exemplary Set Type

## 6.5 Correlation Model Extensions

In a model-driven view on SARI applications, the correlation model (Section 3.4) defines how events of different event types relate to each other, e.g., with respect to a common business process from which they arise. Applied on a continuous stream of business occurrences, an application's correlation model results in a partitioning of event data into groups of related events – so-called *correlation sessions* – which can be handled separately during the event processing.

In practical application scenarios, semantic relationships may, however, exist not only between events, but also between events and entities, and between different kinds of entities. As a first extension to SARI's base application model, we present a generalization of the original correlation model as discussed in Section 3.4. The generalized model allows defining relations not only between events, but between any type of data exposing a non-empty collection of typed and immutable properties. The so-defined, abstract class of *correlatable elements* includes events as well as business entities: While events can be correlated based on their event attributes, business entities can be correlated based on their key properties. Within an event-pattern rule, a so-defined relationship then identifies the business entities that are generally concerned with the occurrence of an event or the status change of another business entity.

### 6.5.1 Meta Model

Figure 6.11 shows the meta model for generalized correlation sets. A generalized correlation set

$$s_g = \{b_1, b_2, \dots, b_n\}, n \geq 1$$

is defined by a non-empty collection of *generalized correlation bands*. A generalized correlation band  $b \in s_g$  specifies a correlation approach for one or

more *correlatable entity types*, where each correlatable entity type  $C$  is defined by a non-empty collection of *immutable properties*. An immutable property  $p = (i, d)$  is defined by an identifier  $i$  and a data type  $t$ ; in a correlatable entity  $c : C$ , the value of an immutable property is required to remain constant during the lifetime of that entity. The abstract concept of a correlatable entity type is implemented by event types and business types. In the former case, immutable properties are given through the set of event attributes. In the latter case, immutable properties are given through the set of key properties.

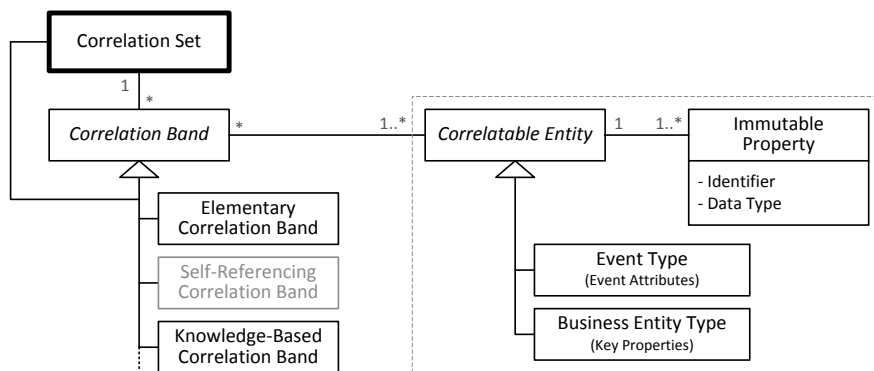


Figure 6.11. Generalized Correlation Set Meta-Model

From the original set of correlation band implementations as presented in Section 3.4.2, *elementary correlation bands* and *knowledge-based correlation bands* can be used with generalized correlatable elements without restrictions. Given a library  $\mathbb{T} = \{T\}$  of correlatable data types, a generalized variant of the elementary correlation band is defined by  $e \subseteq \{(T, a) \mid T \in \mathbb{T}, a \in T\}$ , i.e., a non-empty set of correlatable data types together with an immutable property per type. The *self-referencing correlation band* depends on a unique identifier and a temporal order of elements to correlate; as both aspects are not necessarily available with the generalized correlatable-element model, this correlation-band implementation is restricted to event types only.

### 6.5.2 Example

Figure 6.12 shows an exemplary extended correlation set on the event types of an order process and a base entity of type “Customer”. As shown here, the granularity of correlation sessions is not necessarily equal to the granularity of business entities: While correlation sessions exist per process ID, entities exist per customer ID; thus, a single entity may be associated with several correlation sessions. Conversely, in other scenarios, a single correlation session could equally be associated with several entities.

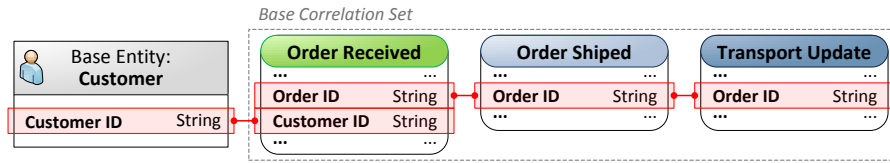


Figure 6.12. Exemplary Extended Correlation Set

## 6.6 Decision Graph Model Extensions

Describing event-processing logic of the form “if pattern, then action”, a SARI application’s *decision graph model* forms the basis for low-level, infrastructural processing steps as well as high-level, *sense-and-respond*-based decision making. In the proposed application architecture for entity-aware event processing, decision graphs consequently find a use on both the “updating side” and the “monitoring side” of business entity providers: On the updating side, decision graphs are applied in the form of *infrastructural rules* to detect updates to business entities in incoming event streams and trigger respective update operations in response. On the monitoring side, decision graphs are used within *sense-and-respond rules* to detect exceptional entity states either individually or as part of complex event patterns.

In the following, we extend the original decision-graph model as discussed in Chapter 4 towards its seamless integration with business entities and business entity providers. We introduce two novel rule components – *business entity actions* (Section 6.6.1) and *business entity conditions* (Section 6.6.2) – which enable application developers to update and monitor business entities directly from decision graphs. In Section 6.6.3, we discuss the handling of *internal business entities* based on special decision-graph variables. All extensions are based on the abstract meta model for business entity providers and therefore are usable with any provider implementation that conforms to the presented specifications.

### 6.6.1 Business Entity Actions

On the updating side of a business entity provider, event-pattern rules are applied to detect the (possibly complex) events that indicate updates to business entities, and to trigger the respective update operations in response. Depending on the given business scenario, the detection of business-entity updates may be of notable complexity and require several processing steps, including *filtering*, *event transformation*, *event aggregation*, and *situation detection*. While the base decision-graph model provides all necessary means for filtering, transforming, and aggregating events, it does not currently enable the management of business entities based on detected event situations.

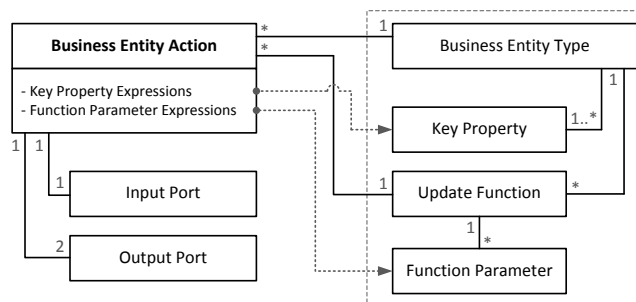


As a first extension to the original rule-component library, *business entity actions* enable application developers to directly and synchronously invoke a user-defined update operation on a user-defined “target set” of business entities. Depending on the given use case, this target set may be derived dynamically from the underlying correlation set, based on a user-defined target business-entity type  $T$ : When the rule component is activated for a correlation session  $s$ , the basic target set is constituted by all  $T$ -typed business entities that are correlated to  $s$ . Alternatively, if no correlation is specified for  $T$  in the given decision graph, the operation is aimed at *all*  $T$ -typed business entities. In either case, further restrictions to the base target set can be specified based on the entities’ key properties, through so-called *key property expressions*.

In the proposed application architecture, business entity actions are typically used in *infrastructural rules* to establish application-wide, virtual representations of underlying real-world entities and states. Business entity actions may also be used in *pattern definitions*; business entities must then be managed *internally* to keep resulting sense-and-respond rules free from side effects. The handling of internal business entities within the presented decision graph model is discussed in detail in Section 6.6.3.

## Meta Model

Figure 6.13 shows the meta model for business entity actions. A business entity action  $a = (p, T, X_K, f, X_f)$  is defined by a business entity provider  $p$ , a business entity type  $T = (K, \dots)$  for  $p$ , an optional collection of *key property expressions*  $X_K$ , an update function  $f$  for  $T$ , and a collection of *function parameter expressions*  $X_f$  for all function parameters of  $f$ .



**Figure 6.13.** Business-Entity Action Meta-Model

**Business entity provider and business entity type.** Every business entity action needs to specify the general *target* of the represented update operation, i.e., identify the exact set of business-entity instances that shall be changed in

case of an update. On a general level, this target is defined as a certain “kind” of entity, through the component’s business entity provider  $p$  and a business entity type  $T$  for  $p$ . By default, a business-entity action is then applied to all  $T$ -typed entities that are correlated to the *active correlation session*, i.e., the correlation session for which  $a$  was activated in the containing decision graph  $d \ni a$ . If no correlation session is available or no correlation relationship is specified for  $T$ , the update is applied to *all*  $T$ -typed entities.

**Key property expressions.** A correlation-based approach to business-entity management is in line with the core event-processing facilities of SARI and often leads to decision graphs that are very simple and easy to understand. However, it does not always allow specifying target sets at a sufficient level of granularity. In the proposed meta model, *key property expressions* therefore enable application developers to further specify the base target set based on selected key properties in  $K$ . Whenever a business-entity action is activated, the component’s key-property expressions are evaluated on the underlying event situation, resulting in a collection of concrete key-property values  $V_K$ . The described update operation is then applied to an element of the base target set if and only if its unique key conforms  $V_K$ .

**Update function and function-parameter expressions.** Given a certain target set of business entities, business entity actions specify the exact kind of update operation to be invoked on these entities by means of an update function  $f$  for  $T$  and a collection of *function parameter expressions*  $X_f$  for all function parameters of  $f$ . When the rule component is activated, concrete values are calculated from  $X_f$  and passed as function parameters to the business entity provider.

### Example

Figure 6.14 shows an exemplary business entity action for incrementing scores of a type “Alarms per server”. We assume that the component is used in a stateless decision graph (Section 4.5), meaning that the concerned score instance must be specified using a key-property expression on the triggering alarm event. The only function parameter expression is defined as a constant, but could as well be used to calculate a value dynamically.

## 6.6.2 Business Entity Conditions

On the monitoring side of a business entity provider, event-pattern rules are applied for the real-time detection of noteworthy entity states. From practical use cases, two access modes have been identified for rule-based entity monitoring:

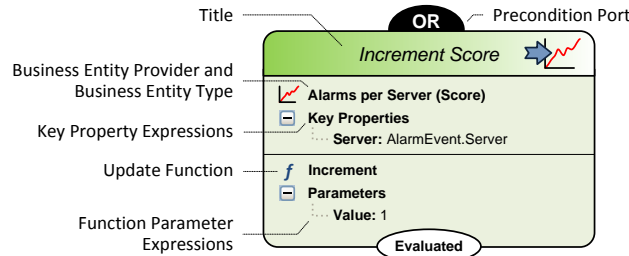


Figure 6.14. Business Entity Action Component

- On demand access:** In on-demand access mode, the state of an entity shall be checked at a certain point in time – e.g., on the occurrence of an event – while subsequent developments of that entity are not taken into account. As an example, consider a use case where an action shall be triggered whenever an alarm occurs on an overloaded server: Here, one would monitor an event stream for alarm events and query a respective score whenever such an alarm is detected.
- Continuous access:** In continuous access mode, the state of an entity shall be checked continuously, enabling rules to react on noteworthy states independent from the exact point in time in which they occur. As an example, consider a use case where an action shall be triggered whenever an overload situation occurs on a server: Here, one would continuously monitor respective scores and fire whenever a threshold is exceeded.

As a second extension to the base rule-component library, *business entity conditions* allow monitoring business entities according to these access modes, based on a user-defined, Boolean *condition expression* on one or more business entities. As with business-entity actions, the targeted set of business entities may be derived dynamically from the active correlation session and can further be restricted based on key-property values. During pattern detection, the chosen expression is then evaluated separately for each element of the target set. Depending on the results of these evaluations and a user-defined binary connective (indicating whether *all*, *at least one*, or *exactly one* business entity must fulfill the condition), an output port for “true” or “false” is activated. If a condition is evaluated on exactly one business entity, so-called *variable expressions* allow retrieving the actual values of a business entity and making them accessible to downstream rule logic through decision-graph variables.

In the proposed application architecture, business entity conditions are typically used in *pattern definitions* for the monitoring of application-wide business entities. In selected scenarios, business entity conditions may also find a use in *infrastructural rules*, where higher-granular business entities may be updated based on lower-granular business entities; for example, scores of a type “Alarms per Day and Server” could be aggregated to scores of a type “Alarms per Week

and Server”, etc. In either case, business entity conditions may be used for the monitoring of *internal business entities* as discussed in Section 6.6.3.

### Meta Model

Figure 6.15 shows the meta model for business-entity conditions. A business-entity condition  $c = (p, T, X_K, x_b, c, m)$  is defined by a business entity provider  $p$ , a business entity type  $T = (K, \dots)$  for  $p$ , an optional collection of key-property expressions  $X_K$ , a Boolean *condition expression*  $x_b$ , a *binary connective*  $c \in \{\text{AND}, \text{OR}, \text{XOR}\}$ , an *execution mode*  $m \in \{\text{on demand}, \text{continuous}\}$ , and an optional collection of *variable expressions*  $V$ . A single activator port and output ports for “true” and “false” are provided for the component’s integration with the other elements of a decision graph.

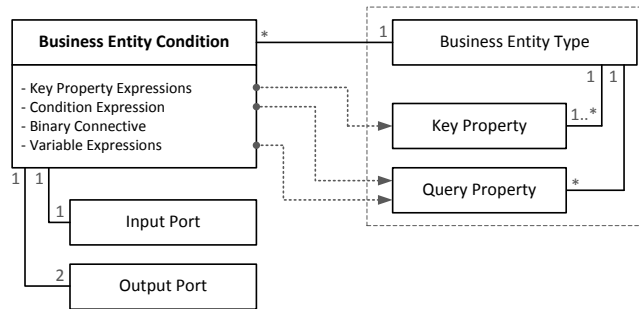


Figure 6.15. Business-Entity Condition Meta-Model

**Provider, type, and key-property expressions.** As with business entity actions, the target set of a business entity condition is defined through a business-entity provider  $p$ , a business-entity type  $T = (K, \dots)$  for  $p$ , and an optional collection of key-property expressions  $X_K$ . At run time, a query is evaluated on all business entities that are correlated to the active correlation session<sup>7</sup> and conform to the calculated key-property values, if such are available.

**Condition expression and binary connective.** The actual query logic of a business entity condition is specified as a type-safe condition expression  $x_b$  on the underlying event situation and all *query properties* for  $T$  on  $p$ . During pattern detection, this condition expression is evaluated independently for each element of the target set. The business entity condition’s *binary connective*  $c$  specifies whether all (AND), at least one (OR), or exactly one (XOR) element must conform to  $x_b$  in order for the component’s “true” port to be activated.

<sup>7</sup> As before, a query is aimed at *all* entities of the chosen business entity type  $T$  if no correlation session is available or no correlation is specified for  $T$ .

**Execution mode.** The *execution mode*  $m \in \{\text{on demand}, \text{continuous}\}$  specifies under which conditions the above-described condition expression is evaluated and the respective output port is activated. In case of *on demand* execution, the evaluation process is executed *on activate*, i.e., each time the preconditions of the condition are fulfilled. In case of *continuous* execution, the evaluation process is triggered whenever a targeted,  $T$ -typed business entity is updated at the business entity provider. As a consequence, the component allows reacting to state changes actively and fully decoupled from low-level updating logic.

**Variable expressions.** Conditional logic based on Boolean condition expressions showed to be an appropriate and intuitive means for modeling noteworthy entity states. Still, situations may arise where the *actual* values of a monitored business entity are required: Consider an example from IT service management (ITSM), where notifications shall be sent whenever the number of alarms of a server exceeds a certain threshold; here, a system administrator might want to retrieve the exact number of alarms (rather than the threshold that has been exceeded) as part of a notification.

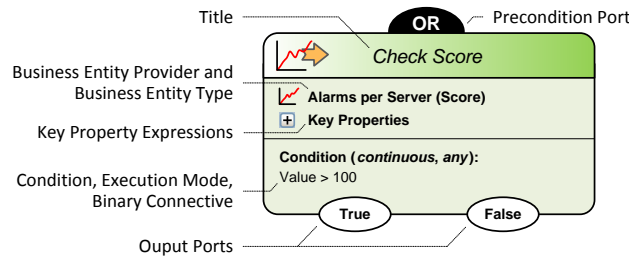
In the proposed meta model, access to the actual values of a business entity is provided through a collection of so-called *variable expressions*. A variable expression  $v = (i, t, x) \in V$  is defined by a variable identifier  $i$ , a data type  $t$ , and an expression  $x$  on the query properties for  $T$  returning a value of type  $t$ . During run time, variable expressions are evaluated together with the component's condition expression according to the chosen execution mode. The results of these evaluations are then assigned to respectively-named *decision-graph variables*, from which they can be retrieved in other rule components. To ensure a 1-to-1 relationship between retrieved values and monitored business-entity instances, variable expressions can currently be defined only if the condition is guaranteed to be evaluated on exactly one business entity. This is the case if all key properties of  $T$  are either used in the underlying correlation set or specified through key-property expressions.

### Example

Figure 6.16 shows an exemplary business entity condition on base entities of type "Customer".

### 6.6.3 Handling Internal Business Entities

In the proposed framework, business entities are managed as application-wide data structures regardless of how they are eventually used in the given SARI application. In order to prevent parallel, mutually-interfering updates between different uses of *internal business entities*, it must therefore be ensured by



**Figure 6.16.** Business Entity Condition Component

the application developer that each use operates on an independent subset of business-entity instances. Most naturally, such an association can be established by expanding the address space of the concerned business entity by an additional key property. For each use of the given business-entity type, a distinct *scope identifier* must then be assigned to this key property.

As a third and final extension to the base decision-graph model, we therefore introduce a special decision-graph variable

`$RuleIdentifier`

It provides access to the unique, string-typed identifier of the current decision-graph instance. If internal business entities are required, this value must be provided as a key-property expression for the respective key property. The rule identifier is implicitly available with all decision graphs and cannot be modified, e.g., as part of a *variable expression*.

## 6.7 Implementation

In the course of our research, the presented approach to state management has been implemented as an experimental extension to the basic SARI architecture as discussed in Section 3.6 and Section 5.7 of this thesis. Figure 6.17 shows the extended SARI architecture from a high-level perspective. Unless otherwise stated, all elements, mechanisms, and communication channels of the original architecture are preserved. In the following, we discuss the different layers of the extended architecture in Section 6.7.1 to Section 6.7.3 and illustrate the management of business entities at run time in Section 6.7.4. Reference provider-implementations for *scores*, *base entities*, and *sets* are discussed in Section 6.7.5.

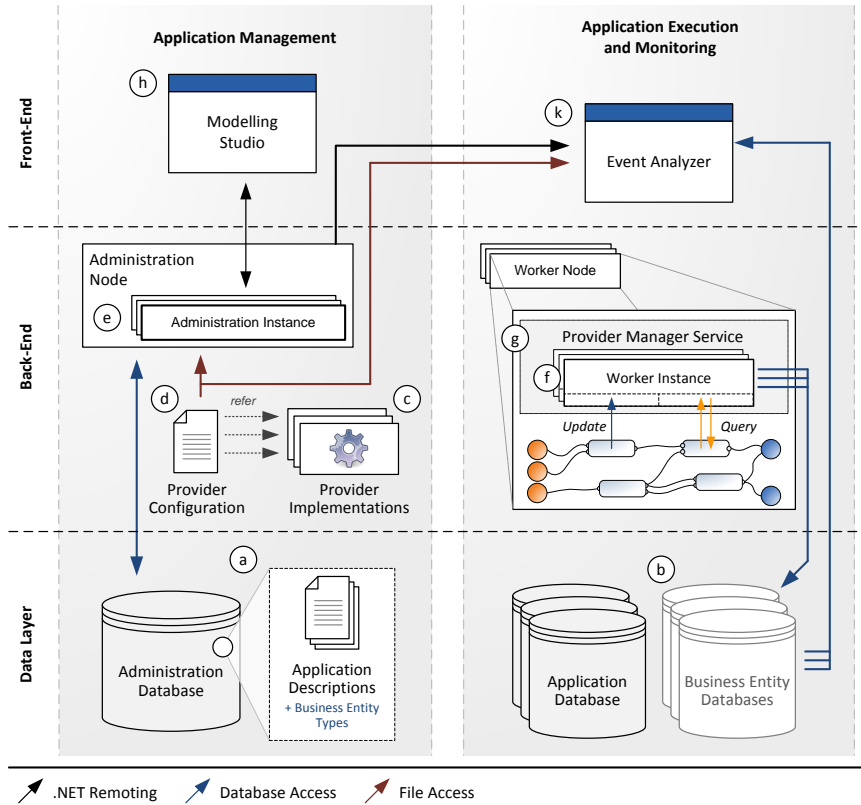


Figure 6.17. Implementation Architecture

### 6.7.1 Data Layer

The data layer of the extended SARI architecture contains all elements of the original architecture, including the central *administration database* and a collection of *application databases* for all SARI applications of the given system. On the application-management side of the architecture, extended *application descriptions* (Figure 6.17a.) are used to persist the novel *business entity model* of a SARI application in an XML-based syntax. During the creation, editing, and deployment of a SARI application, business-entity-related data are retrieved from, and delivered to, the particular business-entity provider implementations to which they belong. On the application-execution side of the architecture, an optional set of *business entity databases* (b.) may be used by the different business-entity providers for persistent data management.<sup>8</sup>

<sup>8</sup> For the sake of consistency with the overall SARI architecture and to minimize the infrastructural requirements of a business-entity provider, the proposed architecture provides provider implementations with access to the application database

Again, the exact structure of these data sources is provider-specific, and no restrictions are imposed by SARI. The used business-entity databases will typically be referenced in the application-specific configurations of the plugged-in business-entity providers, i.e., in the extended application description.

### 6.7.2 Back-End Layer

On the back-end layer of the extended architecture, the actual business-entity provider implementations are provided as .NET assemblies (c.) and referenced in an XML-based configuration file (d.).

On the administration-management side, the *administration node* parses the configuration file on start up and instantiates the so-called *administration instances* (e.) of all plugged-in business-entity providers.<sup>9</sup> The administration instance of a business-entity provider delivers all functionality that is required during the design time of a SARI application: Wizard-based configuration and type-creation dialogs are provided for the definition of a SARI application's business entity model using the *Modeling Studio*. Given the resulting business-entity types, description objects for the different update and query interfaces are exposed. The administration instance eventually provides facilities for the XML-based serialization of business-entity types in order to be persisted as part of an extended application description.

On the application-execution side, *worker nodes* are provided the list of business-entity provider assemblies when first registering at the administration node. Each worker node then instantiates a so-called *worker instance* (f.) per business-entity provider and SARI application in which it is used. Worker instances provide all functionality that is required during the run time of a SARI application: Most basically, this includes updating business entities based on update functions as well as the evaluation of queries on the list of query properties. In both cases, worker instances expect updates and queries to be passed together with a set of key-property values. The update or query is then evaluated on all business-entity instances that conform to the passed key-property values. Within a worker node, the list of worker instances is eventually passed to a central *provider management service* (g.). The provider-management service is accessible from all event services of an application and performs common tasks such as inter-service and inter-worker communication, the interpretation of correlation sets (see Section 6.7.4 for further details), as well as high-level exception handling.

---

of a SARI application. In order to use the application database, an entity provider must support all DBMSs that are supported by the SARI version at hand.

<sup>9</sup> Within the assembly, the different logical parts of a business-entity provider are identified based on predefined interfaces that must be implemented.



### 6.7.3 Front-End Layer

On the application-management side of the extended architecture, the well-known *Modeling Studio* (h.) is used by system operators, solution designers, as well as rule managers for the development and deployment of entity-driven SARI applications. From the underlying administration node, the Modeling Studio retrieves all entity-related data that are required during the design time of a SARI application: For the definition of a SARI application's business entity model, the Modeling Studio retrieves provider-specific configuration and type-definition wizards to be dynamically plugged into the basic user interface. For the integration of business entities with the other parts of a SARI application, the Modeling Studio retrieves update and query-interface descriptions for all business-entity types of the business entity model. These descriptions are then used in the respective editors for the type-safe definition of correlation sets and decision graphs. For example, in SARI's graphical decision-graph editor as sketched in Section 5.8, the creation of business-entity actions and conditions is supported by a step-by-step workflow where users successively chose the concerned business entity provider, business entity type, and eventually specify an update operation or condition expression.

On the application-analysis side, the *Event Analyzer* (i.) facilitates the ex-post analysis of a SARI application through a collection of visualization and data mining tools for historic event data. Similar to their integration with the Modeling Studio, future provider implementations could easily deliver tailored visualization tools for persistent entity data to be dynamically plugged into the base Event Analyzer interface. From the extended application description, the Event Analyzer could retrieve the business entity model of an investigated SARI application, e.g., to establish connections to the used business-entity databases with no further configuration steps required. While visualization methods for scores have been implemented as a permanent extension to the Event Analyzer, plug-in-based integration mechanisms are subject to future work.

### 6.7.4 Business-Entity Management at Run Time

Within the different worker nodes of a SARI installation, worker instances of all plugged-in business entity providers are managed in a central *provider management service*. Whenever a *business entity action* or an *on-demand business entity condition* is activated in a decision graph, the executing rule service synchronously calls the provider-management service and passes all data necessary for performing the update or query. In any case, these data include the active correlation session, if such is available: Based on the correlation session and the concerned business entity type, the provider-management service derives a set of key-property values describing the concerned set of business entities. These

key-property values are then passed to the actual business entity provider for performing the update or query.<sup>10</sup>

Figure 6.18 sketches the approach to *continuous* entity monitoring. On start up, all rules containing a continuous business-entity condition are registered as listeners at the provider-management service (Figure 6.18a.). At run time, whenever an update to a business entity  $e : T$  is performed at a business entity provider (b.), a special notification event is published across all worker instances of that business-entity provider to all registered rule services that are concerned with the affected business-entity type  $T$  (c.). Such a notification event contains the unique identifiers of the concerned business-entity provider and business-entity type as well as all key properties of the concerned business-entity instance. Based on these data, notified rule services then use SARI’s basic correlation mechanism to retrieve the decision-graph states for all related correlation sessions (d.). For each activated session, the incoming notification event is tested against the condition’s key-property expressions. If the condition is concerned with the update, the provider management is called as with on-demand evaluations (e.).

Note that notification events – just like any “common” event – are processed asynchronously by default; thus, a (potentially noteworthy) state at time stamp  $t_1$  may be re-set before the actual evaluation is triggered at time stamp  $t_2$ . While such issues showed to be of little relevance in many practical use cases, we implemented an alternative, synchronous execution mode for single-worker environments to the expense of overall event-processing performance. Further improvements to this mode, as well as its adaptation for multi-worker environments, are subject to future work.

### 6.7.5 Reference Business-Entity Providers

The proposed framework is designed as a generic state-management layer that can be equipped with arbitrary business-entity provider implementations depending on the particular problems that need to be solved. Apart from the basic structure prescribed by the framework and the presented SARI architecture, these implementations may adhere to any programming paradigm and data-management strategy that is suitable for the particular kind of data.

In the course of our research, reference provider-implementations have been developed for *scores*, *base entities*, and *sets* as discussed in Section 6.4. Via their entity-type model, these implementations can be configured to (i) use an in-memory database, or (ii) use the application database for managing entity

<sup>10</sup> On the update interface of a business entity provider, function calls are received generally independent from triggering events and their exact time of creation. Thus, if a strict ordering of events is required, this must be ensured explicitly through a preceding *resequencer* [55] service.

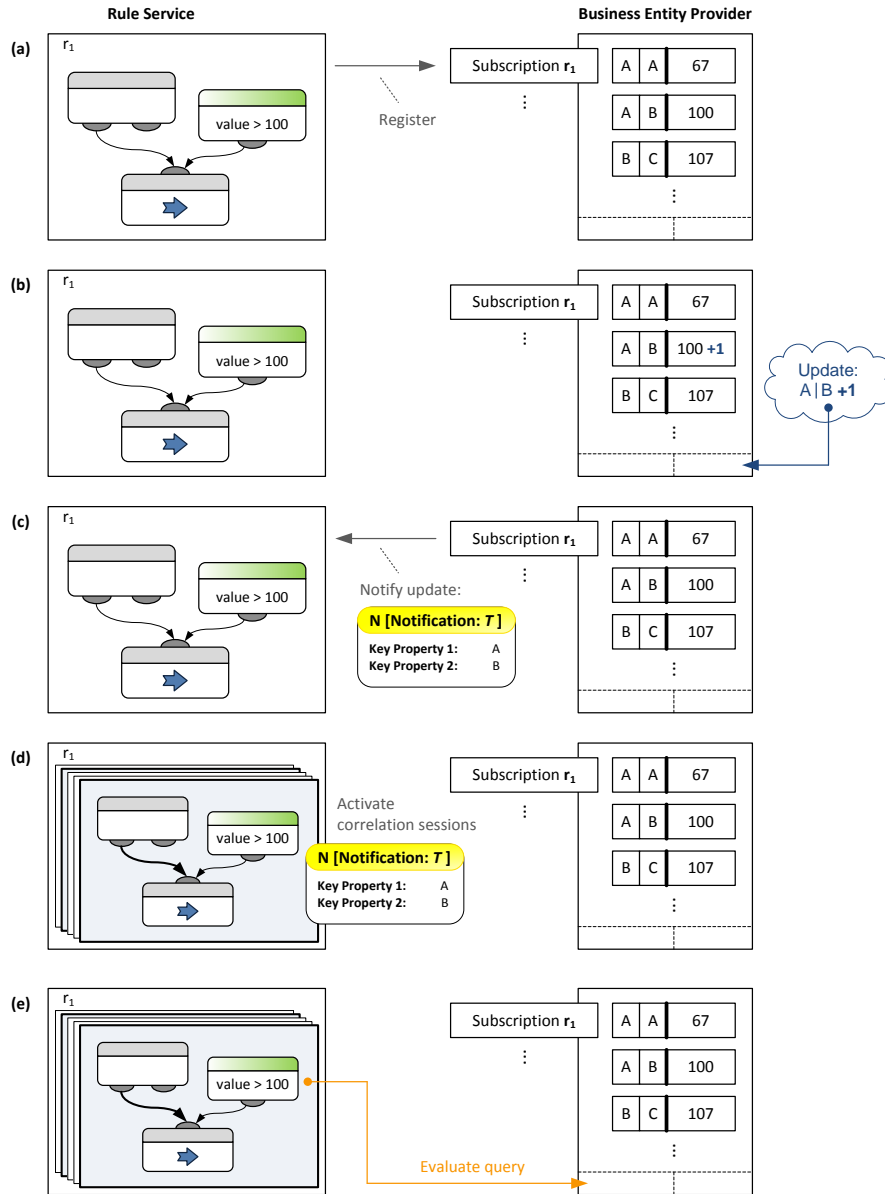


Figure 6.18. Continuous Entity Monitoring

states. In either case, all update-function calls and queries are translated into SQL statement, and, wherever possible, directly forwarded to the underlying DBMS. The used schemata provide for each business entity type one or more *value tables* containing the current state of a business entity. For respectively-configured entity types, one or more *history tables* contain the history of an entity over time. The described separation allows keeping value tables relatively small: As the majority of queries concern the most recent value of a business entity, this leads to better overall querying performance in many use cases.

For a detailed discussion on the employed data-management strategies for *scores*, the interested reader may refer to Roth et al. [104].

## 6.8 Example

In the course of this section, the presented approach to entity-based state management is demonstrated in a real-world example from the IT process automation domain. In the presented scenario, SARI is applied on top of the *UC4 Automation Platform* [130] to dynamically balance task-execution workloads on the basis of regular performance snapshots, task execution events, and the current state of the platform's *task queues*.

Figure 6.19 sketches the scenario from a high-level perspective.

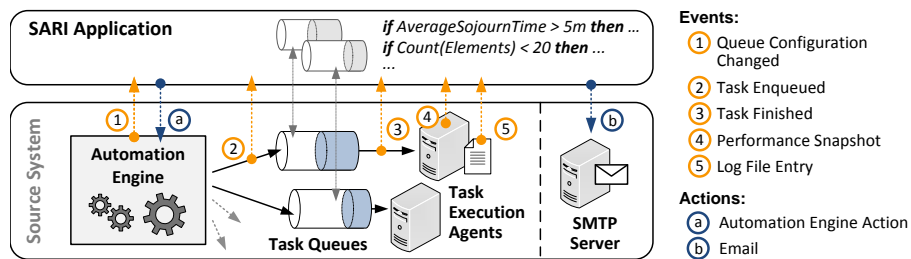


Figure 6.19. Example Overview

In the source system, the automation platform distributes tasks on a network of (virtual or physical) task execution agents. To control the load on such agents, the platform uses extended task queues as an intermediate layer between its engine and the executing agents: Besides providing priority-aware queuing functionality, such a queue allows configuring a certain number of execution slots. All tasks that lie within these slots are then executed in parallel on the associated agent. During run time, the number of slots can be adapted

dynamically, e.g., based on a schedule or on demand by a user. SARI continuously senses the given IT landscape at several integration points: Events of type “Task enqueued”, “Task finished” and “Queue configuration changed” are retrieved directly from the platform. “Performance snapshot” events are published at regular intervals on all agent hosts. “Log file” events are generated whenever message are written to the log. Conversely, SARI may trigger actions directly on the platform and send email notifications.

The described scenario illustrates many of the challenges rule-based CEP is faced with in entity-centric environments: Serving as a key indicator to the health of a system, the overall state of task queues needs to be calculated from incremental, low-level updates and made accessible to rule-based decision making. Counteractions will typically be required whenever a task queue reaches an exceptional state, e.g., whenever its load exceeds a certain threshold. To support this behavior decoupled from low-level updating logic, means for active entity monitoring must be provided. Eventually, the application’s business logic is likely to change over time and should be understandable and editable to users with limited technical skills.

In the example SARI application, a set type “Task Queue” is used to track the overall state of underlying, real-world task queues and make it accessible to rule-based monitoring logic. It is defined by a single key property “Host ID” (identifying a queue by the host to which it belongs), a set of entity properties including an integer-typed property “Execution Slots”, as well as a collection of set-element properties including a task’s ID, type, and priority.

Figure 6.20 shows an exemplary infrastructural rule for updating the resulting business entities: In response to incoming “Task enqueued” events, the “insert” operation is called for the correlated task queue. Further updating rules are applied to remove tasks and to update the number of slots.

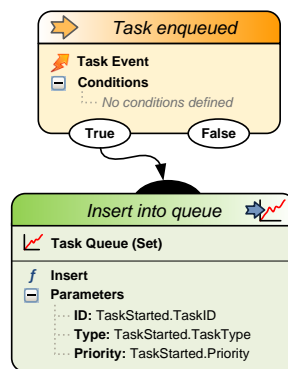
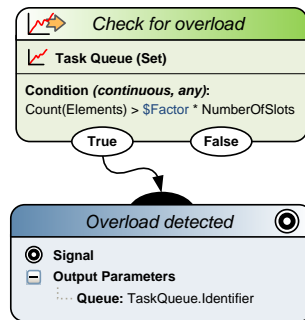


Figure 6.20. Exemplary Updating Rule

Figure 6.21 shows a simple pattern definition for detecting overload situations: Executed in continuous access mode, a condition triggers the pattern’s signal whenever a queue’s load exceeds the number of execution slots by a user-defined factor. The concerned queue’s identifier is published as an output parameter. Within the pattern definition, this identifier is retrieved from the incoming signal (`TaskQueue.Identifier`), which is accessible throughout the processing steps that follow from its occurrence.



**Figure 6.21.** Exemplary Pattern Definition for Continuous Entity Monitoring

Figure 6.22 demonstrates on-demand access to task queues: Here, the number of high-priority file transfers is retrieved and tested against a user-defined threshold whenever an event of type “Recurring file-write error” occurs. In response, the automation engine could be instructed to throttle the number of file transfers or to provision an additional agent. Recurring file-write errors are detected through a preceding infrastructural rule from accumulations of “Log file” events.

## 6.9 Summary

In this chapter, we presented a novel approach to state management for Complex Event Processing. It is based on the idea of business entities, which we understand as identifiable, typed data structures that are accessible across an application. The proposed framework allows integrating arbitrary business entity providers as plug-in-like components depending on the specific requirements of a business scenario. The model is fully integrated with SARI’s ECA-based rule model, which enables users to define event patterns fully integrated with updating and monitoring business entities.

Existing challenges in state-of-the-art, event-pattern-rule-based CEP as discussed in Section 6.1 are addressed as follows:

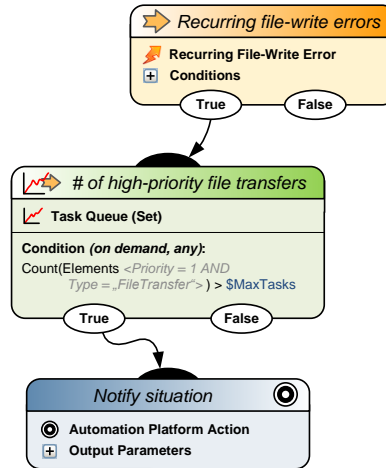


Figure 6.22. Exemplary Pattern Definition for On-Demand Entity Monitoring

- **Durable entity state, complex state-calculation.** Within SARI applications, business entity providers fully encapsulate the calculation of complex entity states from low-level updates, as well as the management of resulting data over time. Access to business entities is provided through easy-to-use update and query interfaces.
- **Active entity monitoring.** In the proposed framework, business entities can be monitored based on two access modes – *on-demand* vs. *continuous* access. In the latter case, a rule evaluates its conditions whenever an update to a business entity is signified, independent from the events that originally caused that update.
- **Context-aware data access.** The concept of business entities is fully integrated with SARI’s correlation model, which enables users to define semantic relationships between events and the business entities to which they belong. In an event-pattern rule, updates and queries are implicitly directed towards those business entities of a user-defined type that are associated with the triggering event or business entity.
- **Ease of use.** The presented approach explicitly focuses on usability from a business-user perspective. For the simple definition of state-management logic, a type model for business entities is presented that enables end users to configure prepared state-management logic according to the specific requirements of their application. Access to so-defined business entities is well integrated with SARI’s decision-graph model and can be defined using predefined, easy-to-use rule components.

## Hierarchical Pattern Modeling

**Abstract** The novel rule-management framework of Sense-and-Respond Infrastructure (SARI) inherently builds upon the reuse of pattern-detection logic on the level of *sense-and-respond rules*. By contrast, SARI’s original event-processing facilities do not support the reuse of pattern-detection logic on the level of event patterns: Each decision graph must instead be defined individually and from scratch, even though considerable commonalities may exist between event patterns. This chapter presents an approach to hierarchical pattern modeling for SARI. It allows integrating sub-level pattern-detection logic into super-level decision graphs through novel rule components – so-called *sub-pattern components* – which serve as a reference to *pattern definitions* and can be integrated with the other elements of SARI’s rule-component library. We present tailored evaluation strategies that enable high-performance event processing as well as arbitrary nestings of pattern definitions and demonstrate our approach in an example from the fraud-detection domain.<sup>1</sup>

### 7.1 Introduction

The presented approach to user-oriented rule management inherently builds upon reuse of pattern-detection logic on the level of *sense-and-respond rules*, which are assembled from prepared, configurable “building blocks”. For example, given a pattern definition “Putter-on fraud”<sup>2</sup> with a string-typed input parameter “League”, business users could use this pattern in a variety of concrete event-pattern rules such as “*If putter-on fraud in ‘Premier League’, then block user account*”, “*If putter-on fraud in ‘Primera Division’, then notify fraud department*”, etc.

---

<sup>1</sup> This chapter is based on the work of Obwegger et al. [87].

<sup>2</sup> In sports betting, the term “putter-on” refers to a person that places bets as a strawperson, on behalf of an official or sportsman that is directly involved in the concerned game.



Reuse is, by contrast, not supported on the level of individual event patterns: In the proposed rule-management framework, each decision graph (i.e., each *pattern definition*, see Section 5.5.3, or *rule definition*, see Section 5.4.3) must instead be designed individually and from scratch, even if some of the described event patterns may have larger parts in common.<sup>3</sup> In practical use cases, commonalities between event patterns showed to be the rule rather than the exception, though: Consider a (sub-) event pattern “Short-term betting transaction” as discussed in Section 5.4.3, where a user pays into a (near-) empty account, places and wins a single high-risk bet, and immediately pays off in full. While not automatically indicating fraud on its own, this pattern may contribute to a whole range of fraud strategies depending on its specific shape (e.g., the specific amount of money involved) and the context in which it occurs.

In the course of our research, we identified the following weaknesses as directly or indirectly attributable to missing reusability on the level of decision graphs:

- **High-complex event-processing logic:** Forcing application developers to define complex event situations “from scratch” and in a single, comprehensive model, the existing approach quickly results in event-processing logic of high complexity, incorporating dozens of events and relationships between them. Such event patterns not only place high demands on application developers, but also make an application difficult to read and prone to errors of semantic as well as syntactic nature. The restricted usability of event patterns eventually complicates the setup and maintenance of an application and leads to higher costs for adopters of an event-processing framework.
- **Redundancy:** Commonalities between two or more event patterns inevitably lead to redundancies within an application’s event processing logic. Such redundancies may in turn lead to inconsistencies between event patterns, increased administration efforts, and reduced maintainability.
- **Limited expressiveness:** In many cases – although, not necessarily – certain parts of a complex event pattern may be considered as self-contained procedures and processes that occur as part of a superordinate, more complex parent process. Depending on the chosen event-processing language, defining such child processes within a single, comprehensive event pattern may impede modeling aspects that are restricted to certain sub-processes only. For instance, in SARI’s decision graph model, the “reset evaluation state” option as is available with all action components can only be applied to a complete decision graph rather than certain parts of it.

From the above weaknesses, we derived the following requirements towards reusability on the level of event patterns:

---

<sup>3</sup> In the following, when speaking of reusability, we refer to reusability on the level of event patterns and decision graphs.

- **Reuse by reference:** To avoid redundancies, an approach to reuse of pattern-detection logic must realize *reuse by reference* as, for instance, known from object-oriented programming. A superordinate event pattern thereby contains pointers to one or more sub-level event patterns rather than the actual pattern-detection logic by itself. Referred event patterns remain as self-contained entities and may be used in an arbitrary number of super-level event patterns. Alternative approaches to reuse are discussed as part of related work in Section 7.2.
- **Expressiveness:** An approach to reuse shall enable users to combine subordinate event patterns in an arbitrary manner and associate these event patterns with arbitrary pre- and postconditions. Generally speaking, given a collection of event patterns, the approach shall enable users to express the largest-possible number of superordinate event patterns from combinations of these patterns.
- **Configurability:** Maximum reusability can be obtained only if pattern-detection logic can be defined in a form that is inherently generic and can be adapted with respect to the different contexts in which it is to be used. Pattern-detection logic must therefore be decoupled from any kind of (purpose-specific) reaction logic and furthermore be configurable.
- **Ease of use:** An approach to reuse shall be well-integrated with the existing pattern-modeling facilities of an event-processing framework and support a continuous flow of work across both the creation of new event patterns and the integration of existing event patterns. To ease the cooperation between multiple application developers and facilitate the definition of event-pattern libraries, the approach shall furthermore provide a concept of *information hiding* for event patterns. In particular, the approach shall enable users to incorporate event patterns without having to know and understand their exact structure, based on documented interfaces and high-level descriptions of their semantics.
- **Performance:** The approach shall deliver a good overall event-processing performance. In particular, incorporating an event pattern as part of another event pattern shall not have any significant impact on a framework's event-processing performance in comparison to a possible stand-alone, single-model equivalent.

In this chapter, we present a novel approach to hierarchical pattern modeling for Sense-and-Respond Infrastructure. Together with the proposed rule-management framework, it enables reuse of pattern-detection event-processing logic not only on the level of sense-and-respond rules, but also on the level of *event patterns*, which may now be composed from collections of sub-level event patterns in a hierarchical decision graph.

The main idea of our approach is that of letting users refer and incorporate encapsulated pattern-detection logic into a higher-level decision graph in the form of configurable, easy-to-use rule components, so-called *sub-pattern components*. These rule components can be integrated into a decision graph just like any other rule component in non-hierarchical decision graphs: Via its *input port*, a sub-pattern component can be equipped with arbitrary preconditions in full accordance with the modeling workflow for non-hierarchical decision graphs. Via its *output ports*, the represented pattern-detection logic can by itself be used as a precondition for downstream rule components. In the proposed model, output ports allow separating different manifestations of a detected event situation; for instance, one could distinguish between “normal”, “serious” and “extreme” manifestations of an alert and activate downstream parts of a decision graph accordingly. *Input parameters* allow configuring sub-level decision graphs depending on the specific contexts in which they are used. Input-parameter values are specified at the referring rule component and can be calculated dynamically from preceding events where necessary. *Output parameters* provide insights to selected characteristics of matched sub-level event situations in a controlled and abstracted manner and can be mapped to “local” variables in the super-level decision graph. The proposed evaluation strategies are tailored to the characteristics of hierarchical pattern definitions and facilitate arbitrary nestings of pattern-detection logic.

Taken together, the proposed approach fulfills the above key requirements as follows: Reuse by reference is achieved through the idea of sub-pattern components, which serve as pointers to sub-level pattern-detection logic. Expressiveness is achieved through the arbitrary integration of sub-pattern components into super-level decision graphs. Configurability is achieved through the use of input parameters. Ease of use and information hiding is achieved through input parameters and output parameters. High event-processing performance is achieved through tailored, integrated evaluation strategies.

Figure 7.1 illustrates the detection of hierarchical event patterns in a simple example. Consider a rule from the fraud-detection domain, where a notification shall be triggered if

- (i) a user repeatedly carries out high-stake short-term transactions, i.e., if the user pays into his account, places a single high-risk bet, and immediately pays off in full, and
- (ii) the user’s total balance is above a certain threshold.

In the underlying decision graph, this event situation could be assembled as a combination of respectively-configured sub-patterns. If these individual patterns are detected under the specified temporal and/or contextual constraints in the incoming event stream, the overall event pattern is matched and the associated action is executed.<sup>4</sup>

<sup>4</sup> A more detailed discussion of the given example is presented in Section 7.5.

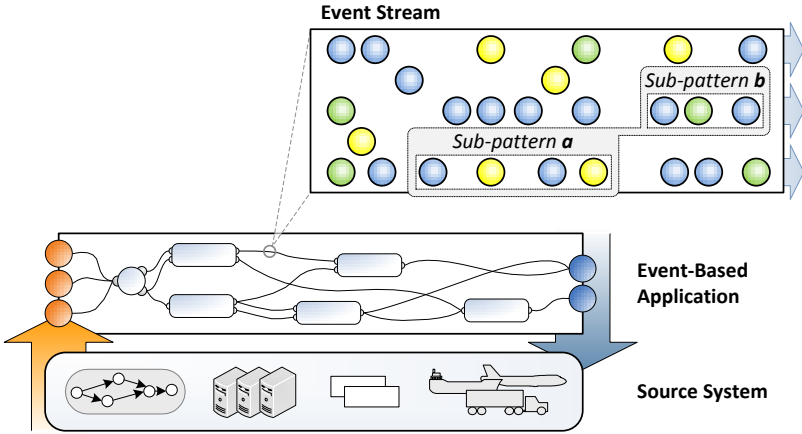


Figure 7.1. Hierarchical Event-Pattern Detection

### SARI Application Model – Revisited

Figure 7.2 sketches the model-driven view on SARI applications, extended by hierarchical decision graphs. In contrast to the original model, decision graphs may now refer to other decision graphs for their integration as part of higher-level, hierarchical event-processing logic.

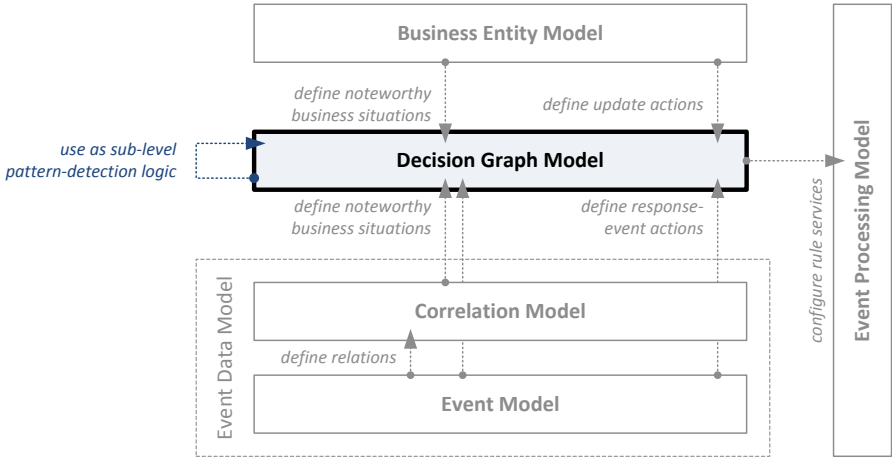


Figure 7.2. SARI Application Model with Sub-Pattern Relationships

## Outlook

The remainder of this chapter is structured as follows: In Section 7.2, we discuss related work. In Section 7.3, extensions to the base decision-graph model are presented. Section 7.4 discusses the evaluation of hierarchical decision graphs during the execution of an event-processing application. A concrete example, illustrating the use of hierarchical pattern-detection logic in the context of online gambling and fraud detection is presented in Section 7.5.

## 7.2 Related Work

Hierarchical structures have always played a crucial role for describing complex situations and procedures. Allen's interval-based temporal logic [6] laid the formal foundations for hierarchical representations. It specifies thirteen possible relationships between two intervals: *before*, *equal*, *meets* (where an interval  $t_1, t_2$  directly follows an interval  $t_3, t_4$ ,  $t_2 = t_3$ ), *overlaps*, *during*, *starts* (where two intervals have the same start time) and its counterpart *finished*.<sup>5</sup> The proposed approach to hierarchical pattern modeling allows assembling sub-level pattern definitions in a decision graph, where temporal relations between the represented intervals can be specified as series and parallel combinations of two or more sub-pattern components. With respect to Allen's temporal logic, these combinations allow distinguishing between serial relationships (i.e., *before* and *meets*) and parallel relationships. Further distinctions are not per se expressible. If such are required, these must be explicitly ensured in a (post-) condition component based on output parameter values.

*Event-abstraction hierarchies* as originally discussed by Luckham [71] describe a step-by-step aggregation of events from low-level core events to high-level business situations, where events of the first layer contribute to event patterns/complex events on the second layer, and so forth. Event-abstraction hierarchies thereby establish different views to a system, which may be optimized towards the demands of different groups of stakeholders. Event-abstraction hierarchies are – more or less explicitly – supported by any modern event-processing framework, including SARI. Here, the separation between infrastructural and sense-and-respond rules may, in fact, be considered a further extension of event-abstraction hierarchies, where the top-most layer is conceptually separated from lower ones and made accessible to business users. Hierarchical pattern modeling as proposed in this chapter shall therefore serve as a complement rather than a replacement for event-abstraction hierarchies: While event-abstraction hierarchies follow an *indirect* approach to reuse based

<sup>5</sup> A framework for mining *meets*, *overlaps* and *during* relationships between recurring patterns in long strings of tokens has been proposed by Mooney and Roddick [80].

on detected events (and therefore reduce overall pattern-detection efforts), hierarchical event patterns directly incorporate the actual pattern-detection logic (and therefore reduce the overall number of event instantiations). On a more technical level, direct reuse provides greater control over sub-level event patterns.

In RAPIDE [71], direct reuse of pattern-detection logic is enabled through the concept of *pattern macros*. A pattern macro associates an event pattern with an easy-to-understand macro name and a list of pattern parameters. At run time, the macro call is replaced by the represented event pattern in a so-called *expansion* process. Being performed *on demand* – i.e., only if the represented event pattern is actually required in the pattern-detection process – pattern-macro expansion facilitates recursive nestings of event patterns. Hierarchical pattern modeling adopts from pattern macros the lazy expansion strategy and its support for recursive pattern-detection logic. However, it follows a very different approach in the handling of input and output parameters. ruleCore and its ruleCore markup language (rCML) [117] features the stepwise composition of composite events from sets of basic events and/or other composite events in an XML-based syntax, where they can be combined using operators such as *or*, *and*, and *sequence*. At run time, complex patterns are evaluated in an integrated, tree-based algorithm. In contrast to the proposed approach, event patterns in ruleCore can not be adapted to different contexts (e.g., through input parameters or similar concepts), which we believe is a key criterion for end-user-oriented application development.

In Event Stream Processing (ESP), concepts such as *sub-queries* and *joins* provide the basic means for combining lower-level event patterns. Mangkorntong and Rabhi [76] present an approach to event-pattern composition where event patterns are described along with sets of typed *event-pattern parameters* in a generic, system-independent model. Associated with such a model, vendor-specific representations for different event-processing systems may exist. In the proposed architecture, so-defined event patterns may then be combined to higher-level event patterns by “binding” respective parameters; for instance, the parameter “Output stream” of a first sub-pattern may be associated with an input parameter “Input stream” of a second sub-pattern, etc. The authors’ work offers a sophisticated parameter model that allows modeling complex dependencies between parameters (both on the level of input-parameter values and on the level of input-parameter data types), which is not currently supported with hierarchical pattern modeling in SARI. Yet, to our best knowledge, there is no consistent workflow across the definition of low-level, non-composite patterns and the definition of hierarchical event patterns. Also, the composition of event patterns appears to be restricted to the various parameters of the involved sub-patterns and does not allow incorporating any further pattern-detection logic specific to the super-level pattern.

Liu and Rundensteiner [70] and Liu et al. [68, 69] introduce the concept of *event-pattern hierarchies* as directed, acyclic graphs of event-pattern queries with edges representing sub-query relationships. In their work, a pattern query is defined as a sequence of event types to be detected in the specified order in an incoming stream of events. A so-defined pattern query  $q$  is in a sub-query relationship to another query  $q'$  if it (i) contains additional event types (*pattern hierarchy*), or (ii) uses an event type  $T$  as a replacement for an event type  $T'$  so that  $T' \text{ :> } T$  (*concept hierarchy*). Based on the resulting graph structure, the authors propose a technology where interrelationships between queries are exploited for optimized shared processing and maximum reuse of intermediate results. While the authors' approach aims to detect hierarchies "a posteriori" (in queries that have originally been defined in an independent manner, by different users of a system) with the goal of performance optimization, our approach provides for hierarchies to be modeled *explicitly* with the goal of reusability and ease of use. Performance optimizations based on hierarchical structures are not currently addressed in the proposed solution. Still, further improvements of the present evaluation strategies are subject to future work.

### 7.3 Decision Graph Model Extensions

In the course of this chapter, the concept of *hierarchical pattern modeling* has been introduced as a novel approach to reuse on the level of event patterns and low-level, infrastructural application development. In the following, we present different extensions to SARI's base decision-graph model, which are necessary for modeling hierarchical pattern-detection logic in a way that is suitable for the targeted user group. We give a short recapitulation of *pattern definitions* as originally discussed in Chapter 5 of this thesis and extend SARI's existing rule-component library by novel *sub-pattern components*.

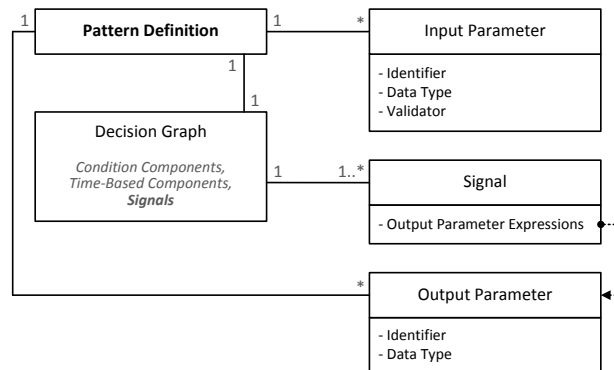
#### 7.3.1 Pattern Definitions

Hierarchical pattern modeling builds upon on the basic idea of letting users assemble complex event patterns through special rule components, which serve as pointers to the required, lower-level pattern-detection logic. To be (re-)usable as part of hierarchical decision graphs, pattern-detection logic must thereby be represented in a form that adheres to the following requirements:

- **Reuse by reference:** To be incorporated "by reference" into higher-level event-processing logic, meaningful pieces of pattern-detection logic must be uniquely addressable as part of an event-based application. This is the case if pattern-detection logic is available in the form of identifiable, self-contained artifacts.

- **Configurability:** To obtain maximum reusability, pattern-detection logic must be configurable and decoupled from any kind of (purpose-specific) reaction logic.
- **Ease of use:** To ensure usability from an end user’s perspective and facilitate cooperation between multiple application developers, pattern-detection logic shall realize a form of *information hiding*, i.e., be usable based on documented interfaces and high-level descriptions of the represented semantics.

Within SARI’s existing event-processing facilities, the described requirements are effectively fulfilled through the concept of *pattern definitions* as presented in Chapter 5 of this thesis. Figure 7.3 recapitulates the respective meta model. A pattern definition  $p = (IN, OUT, d)$  is defined by a collection of input parameters  $IN$ , a collection of output parameters  $OUT$ , and a passive decision graph  $d$ . A decision graph is said to be *passive* if it is composed from condition components, time-based components, and signals only.



**Figure 7.3.** Pattern Definition Meta-Model

**Input parameters.** Input parameters allow configuring the represented pattern-detection logic depending on the specific context in which it is used. Within the pattern definition’s decision graph, input parameters can be used as typed placeholders across all rule components.

**Output parameters.** In many cases, continuing processing steps demand access to selected characteristics of a matching event situation, e.g., to dynamically adapt associated reaction logic in a sense-and-respond rule. Output parameters enable designers of pattern definitions to specify such characteristics in the form of typed, named, and documented data fields; the exact calculation of the corresponding values is specified with the different signals of the underlying decision graph and therefore encapsulated as part of a pattern definition.



**Decision graph and signals.** The actual pattern-detection logic of a pattern definition is defined as a passive decision graph, which is assembled from condition components, time-based components, as well as a non-empty collection of signals. Signals are special rule components that signify the detection of a matching event situation to arbitrary signal listeners. Multiple signals allow distinguishing different manifestations of an event situation, e.g., between “high”, “serious” and “extreme” overload situations.

For a detailed discussion on pattern definitions, the interested reader may refer to Section 5.5.3.

### 7.3.2 Sub-Pattern Component

Pattern definitions provide effective means for describing pattern-detection logic in a form that facilitates reuse and hides underlying complexity. To support the modeling of complex pattern-detection logic from combinations of so-defined pattern definitions, we extend SARI’s existing rule-component library as presented in Chapter 4 of this thesis by the novel *sub-pattern component*. Within a decision graph  $d$ , a sub-pattern component  $sub_p \in d$  serves as a reference to a sub-level pattern definition  $p$  and thereby establishes a hierarchical *sub-pattern relationship*,  $d \xrightarrow{sub} p$ . From an end user’s perspective, the sub-pattern component then *represents* the referenced sub-level pattern definition  $p$  in the super-level decision graph, where each *signal* in  $p$  corresponds to an output port of the sub-pattern component, and output parameters are mapped to local decision-graph variables. In order to fulfill the component and activate an output port  $out_i$  (representing a signal  $signal_i \in p$ ), an event situation must comply to

- (i) all preconditions of  $sub_p \in d$  in  $d$ , and
- (ii) all preconditions of  $signal_i$  in  $p$  as configured based on user-defined input-parameter values for  $sub_p$ .

Figure 7.4 illustrates the described semantics in a simple example. In the shown decision graph, an incoming event  $e_1$  matches the initial event condition  $c_{1.1} \in d$  and causes the sub-pattern component  $sub_p$  to be activated. In further consequence,  $sub_p$  represents the pattern-detection logic of a respectively-configured instance of  $p$  within  $d$ : The following events  $e_2$  and  $e_3$  together match the resulting sub-level event pattern and cause the sub-pattern component’s output port to activate.  $e_4 \in d$  matches the final condition  $c_{1.2} \in d$  and triggers a signal.

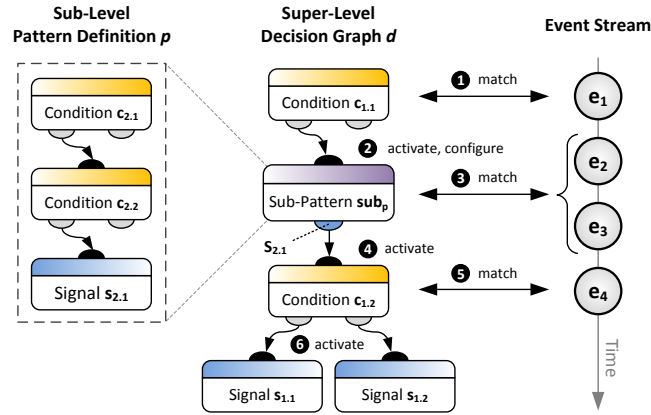


Figure 7.4. Pattern Detection with Sub-Pattern Components

### Meta Model

Figure 7.5 shows the meta model for sub-pattern components. Let  $p = (IN, OUT, d)$  be a pattern definition defined by a collection of input parameters  $IN$  and a collection of output parameters  $OUT$ . A sub-pattern component  $sub = (p, X_{IN}, M_{OUT})$  is then defined by a referred *sub-level pattern definition*  $p$ , a collection of *input-parameter expressions*  $X_{IN}$  for all input parameters in  $p$  and a collection of *output parameter mappings*  $M_{OUT}$  for all output parameters in  $p$ . A single activator port  $p_{in}$  and a collection of output ports  $P_{out}$  are provided for the component's integration with the other elements of a decision graph,

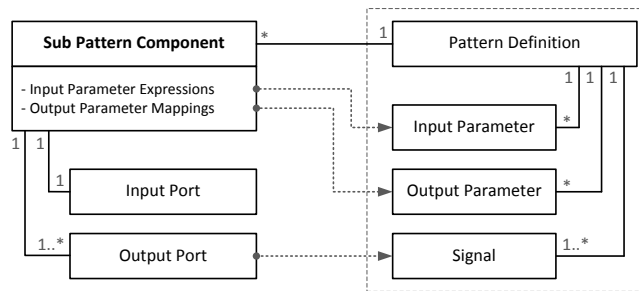


Figure 7.5. Sub-Pattern Component Meta-Model

**Sub-level pattern definition.** The sub-level pattern definition represents the actual pattern detection logic that is referred by and integrated into the super-level decision graph  $g$  through  $sub$ . The proposed model allows integrating any pattern definition that fulfills the following conditions:

- (i) *Equal correlation sets*: SARI's approach to rule-based event processing builds upon a separation between *event correlation* and *pattern detection*, where decision graphs are evaluated independently for each correlation session of the associated correlation set. To guarantee the evaluation of sub-level components in accordance with the expected semantics, a referred sub-level pattern definition must therefore be based on the same correlation set as the super-level decision graph; i.e., given a sub-level pattern definition with a decision graph  $g_{sub} = (C, P, c, \Delta t)$  and a super-level decision graph  $g_{super} = (C', P', c', \Delta t')$ ,  $c = c'$  must hold.
- (ii) *Non-infinitive recursions*: The proposed approach facilitates recursive pattern definitions, both through direct and indirect recursions. To avoid *infinite recursions*, a recursive pattern definition must specify an exit condition that not again results in a recursion. Such an exit condition is given only if preconditions are defined for a recursive sub-pattern component at some level of the hierarchy.

**Input parameter expressions.** The concept of pattern definitions has been designed with the focus on reusability across different application scenarios. In the proposed architecture, the referenced sub-level pattern definition can be configured according to the context in which it used through a collection of input parameter expressions  $X_{IN}$  for all input parameters in  $p$ . When the sub-pattern component is activated as part of the super-level decision graph, these expressions are evaluated and the resulting values are made available to the referenced pattern-detection logic.

**Output parameter mappings.** In the proposed architecture, output parameters of a referred sub-level pattern definition are accessible to super-level event-processing logic, both at design time and during run time. To resolve possible naming conflicts between the inherent variables of a decision graph (e.g., input parameter placeholders) and the output parameters of a sub-level pattern definition, the sub-pattern model provides for the definition of so-called output-parameter mappings for selected output parameters of  $p$ . An output-parameter mapping  $m = (out, i_D) \in M_{OUT}$  is defined by an output parameter  $out = (i, t)$  and a variable identifier  $i_P$ . In a decision graph  $d \ni sub$ , the output parameter  $out$  is then accessible through a variable  $v = (i_D, t)$ . If no output parameter mapping is defined for an output parameter, this output parameter is not accessible in a super-level decision graph.<sup>6</sup>

**Input and output ports.** Just like any other rule component, sub-pattern components are integrated with the other elements of a decision graph through collections of input ports and output ports. On the input side, a single activator

---

<sup>6</sup> Naming conflicts are not a major issue in real-world scenarios. In SARI's rule and event-pattern modeling facilities, default output-parameter mappings are therefore created for all output parameters of a sub-level pattern definition such that  $\forall m \in M_{OUT} : i_{out} = i_D$ . These mappings can then be changed by the user.

port  $p_{in}$  allows defining preconditions for the referenced sub-level pattern-detection logic. On the output side, output ports are available for all signals of the referenced sub-level pattern definition. During run time, an output port is then activated whenever the corresponding signal is activated in the sub-level pattern definition.

### Example

Figure 7.6 shows an exemplary sub-pattern component “High CPU utilization”, referring to a pattern definition of the same name. Static input-parameter expressions are defined for the sub-level pattern definition’s input parameters. From the sub-level pattern definition’s output parameters, the “Utilization” parameter is mapped to a local variable of the same name. Possible other output parameters are not mapped and, thus, are not accessible in the super-level decision graph. The sub-level pattern definition’s signals – “Extreme”, “Serious” and “High” – are accessible through output ports.

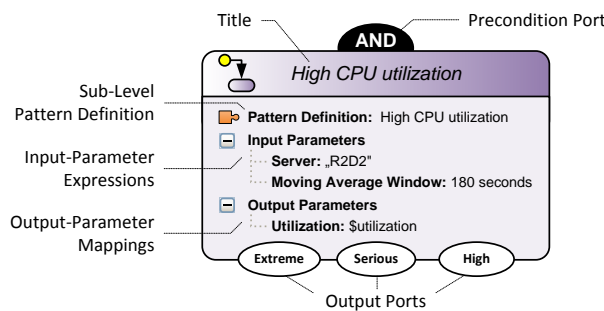


Figure 7.6. Sub-Pattern Component

## 7.4 Evaluation

While the presented approach to hierarchical pattern-modeling has been designed with the goal to seamlessly integrate with the original rule model from an end user’s perspective, it inevitably requires changes to the actual evaluation of decision graphs in SARI’s back-end. In the course of our research, we investigated two possible approaches to the evaluation of hierarchical decision graphs: *Evaluation by expansion* builds upon a stepwise flattening of hierarchical structures so that the original evaluation strategy can be applied without further adaptations. In the *hierarchical evaluation* approach, the various layers of hierarchical pattern-detection logic are treated as separate – though

interconnected – decision graphs. In the following, we present the described evaluation modes in greater detail, discuss possible “pros” and “cons” and compare the approaches’ suitability for real-world applications of CEP.

#### 7.4.1 Evaluation by Expansion

As a first evaluation strategy, *evaluation by expansion* has been designed with the basic goal of minimizing changes to the original evaluation mechanism and decision-graph state-management logic. The proposed approach builds upon the idea of adequately replacing a sub-pattern component by the referenced sub-level pattern-detection logic as soon as it is activated for one of the underlying correlation sessions. We refer to such a replacement as *expansion* in the remainder of this section. On the flattened decision graph, evaluation and state management can be performed without further adaptations.<sup>7</sup>

Algorithm 7.1 outlines the *Expand* algorithm in pseudo code: *Expand* receives a super-level decision graph  $d$ , a referred sub-level pattern-definition  $p$  as well as a referring sub-level component  $sub$ ;  $p.initials$  refers to all components in  $p$  that do not have preconditions. *Expand* is called when the preconditions for  $sub$  are fulfilled for the first time. Unless otherwise stated, all operations are applied on  $d$ .

In lines 1 and 2, we replace  $sub$  by the referenced sub-level pattern-detection logic  $p$  excluding all signals and ingoing connections thereof.

In lines 3 to 6, we establish dependencies between

- all predecessors of  $sub$  in  $d$ , and
- all initial components in  $p$ .

Furthermore, we set the precondition operators of all initial components in  $p$  to the precondition operator as defined for  $sub$ .

In lines 7 to 12, the above step is repeated accordingly for signal components/output ports: For each signal component  $signal_i \in p$ , we establish dependencies between

- all predecessors of  $signal_i$  in  $p$ , and
- all successors of the corresponding port  $out_i$  in  $d$ .

---

<sup>7</sup> It is essential to note that the described approach does not manipulate the actual decision graph as originally modeled by the user, but rather is applied within the respective *decision-graph evaluator*. Decision-graph evaluators are created from decision graphs at run time and perform the actual event processing based on incoming events and associated decision-graph states (see Section 4.5 for further details).

**Algorithm 7.1**  $\text{Expand}(d, p, sub)$ 


---

```

1: remove sub
2: add ( $p \setminus p.signals$ )

    // reconnect “in”
3: connect sub.predecessors with p.initials
4: for all initial  $\in p.initials$  do
5:   initial.operator  $\leftarrow sub.operator$ 
6: end for

    // reconnect “out”
7: for all signal  $\in p.signals$  do
8:   connect signal.predecessors with sub.outport[signal].successors
9:   for all successor  $\in sub.outport[signal].successors$  do
10:    successor.operator  $\leftarrow signal.operator$ 
11:   end for
12: end for

    // replace input and output parameters
13: replace p.inputParameters by sub.expressions
14: for all signal  $\in p.signals$  do
15:   replace signal.outputParameters by signal.expressions
16: end for

```

---

We again adapt the precondition operators of all successors in  $d$ . (Note that if a successor has further preconditions and the original operator differs from the operator of  $signal_i$ , the newly-established dependencies must be grouped, e.g., via an intermediate rule component that evaluates to “true” by definition.)

In lines 13 and 16, we replace any occurrence of an input parameter  $in_i$  in  $p$  by the corresponding expression  $x_{in_i}$  as defined in  $sub$ ; this replacement includes all occurrences in output-parameter expressions as defined in signal components. Finally, we replace any occurrence of an output parameter  $out_i$  in  $d$  by its expression  $x_{out_i}$ .

Figure 7.7 and Figure 7.8 sketch the behavior of the *Expand* algorithm in a simple example. Figure 7.7 shows the starting situation. Figure 7.8a. shows  $P$  after replacing the sub-pattern component and reconnecting pre- and postconditions. Figure 7.8b. shows the final outcome of *Expand*, with resolved input- (green) and output-parameters (red).

After finishing *Expand*, the basic evaluation for non-hierarchical pattern-detection logic can be continued without restrictions.

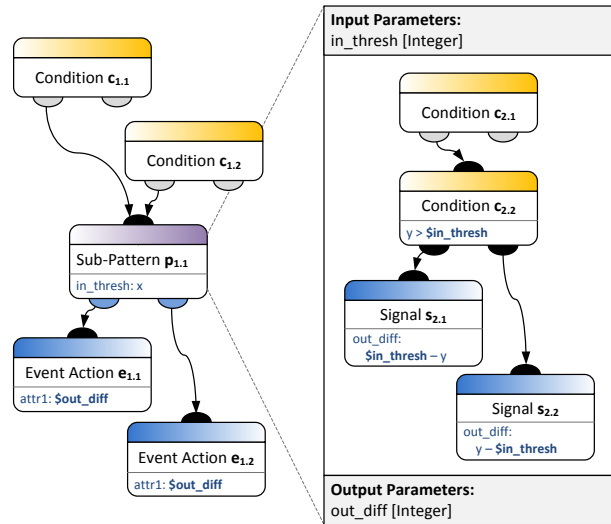


Figure 7.7. Applying *Expand* to an Exemplary Decision Graph (1 of 2)

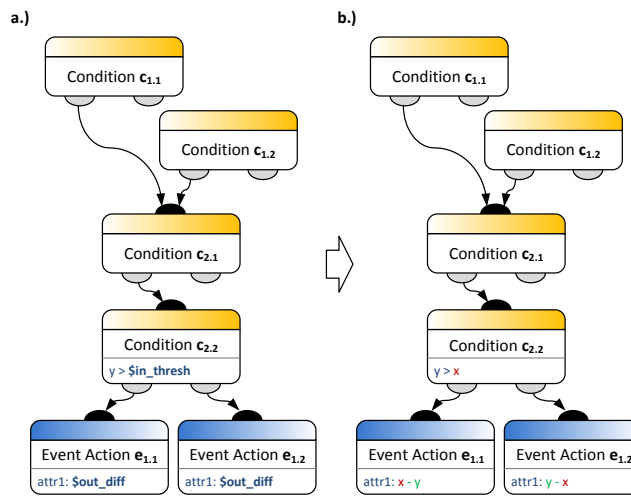


Figure 7.8. Applying *Expand* to an Exemplary Decision Graph (2 of 2)

### 7.4.2 Hierarchical Evaluation

*Evaluation by expansion* as presented in the above section allows maintaining the original evaluation and state-management approach as has been used for non-hierarchical decision graphs. However, by resolving hierarchical structures in their entirety, it impedes treating the various layers of a decision graph separately. This is needed when configurations shall apply to a certain sub-level event pattern only. As a consequence, an alternative evaluation approach has been investigated that is more aligned with the specific structure of hierarchical decision graphs.

Using the so-called *hierarchical evaluation* approach, a new, generally self-contained decision-graph evaluator is instantiated with the first activation of a sub-pattern component in a decision graph. This evaluator is based on the referenced sub-level decision graph and made accessible for subsequent processing steps. The top-level decision-graph evaluator then registers as a listener to the sub-level decision graph's signals. During run time, the top-level decision-graph evaluator is synchronously notified whenever a signal is activated in the sub-level decision-graph evaluator. The hierarchical structure of decision-graph evaluators is also reflected in state management, where the data of a sub-level decision graph are managed as a part of the top-level decision-graph state for each underlying correlation session. Note that in contrast to top-level decision-graph evaluators where possible input-parameter values are equal for all correlation sessions, input-parameter values for sub-level decision graphs are calculated dynamically and, thus, may be distinct for each correlation session. Input-parameter values for sub-level decision graphs are therefore managed as part of the sub-level decision-graph state.

The behavior of a decision-graph evaluator is now defined as follows:

1. Given an incoming event or signal, the state of a decision graph is retrieved along with the states of all sub-level decision graphs, if available.
2. The incoming event is processed based on the represented decision graph "bottom up", i.e., each rule component is evaluated prior to all its preconditions in the decision graph.
3. If a sub-level component is *active*, the incoming event or signal is forwarded to the corresponding sub-level decision-graph evaluator, where it is processed in the described fashion and may be forwarded to sub-sub-level evaluators, and so forth.
4. If a signal is activated in the sub-level decision-graph evaluator, the respective output port is synchronously activated in the super-level decision-graph evaluator. Here, the activation may cause further processing steps, e.g., may trigger an action.



5. If a sub-level component is deactivated as a result of the given event or signal, the state of the sub-level decision graph is reset for the given correlation session.

### Extended Sub-Pattern Configurations

By managing the various layers of hierarchical pattern-detection logic as separate decision graphs, the presented approach provides a degree of flexibility in the activation, deactivation, and configuration of sub-level pattern-detection logic that is not available with *evaluation by expansion* as discussed above. When the *hierarchical evaluation* strategy is applied, sub-pattern components may therefore provide the following, extended configurations to application developers:

- **Reset on activate.** The Boolean “reset of activate” property specifies whether the state of a sub-level decision graph shall be reset with each activation of the sub-pattern component in the super-level decision graph. Reset on activate would be required if the preconditions of a sub-pattern component must be fulfilled exactly once before the represented sub-level pattern definition.
- **Recalculate input parameters.** The Boolean “recalculate input parameters” property specifies whether the input parameters of a sub-level pattern definition shall be recalculated whenever the sub-pattern component is activated. The described property is of practical relevance only if the “reset on activate” property is set to false. (After a sub-level decision graph has been reset, input-parameter values are re-calculated anyway.)

#### 7.4.3 Discussion and Comparison

In the course of our research, we identified the following key requirements for evaluation strategies for hierarchical decision graphs. An approach must

- (i) conform to the described semantics of sub-pattern components,
- (ii) support arbitrary nestings of decision graphs, including direct and indirect recursions,
- (iii) allow evaluation steps across all (activated) layers of a decision graph; for instance, an incoming event shall generally be able to trigger *event conditions* both on the main layer and all sub-level pattern definitions of a decision graph.

Both *evaluation by expansion* and *hierarchical evaluation* fulfill these requirements. For both approaches, recursions are supported through “on demand” expansions and “on demand” instantiations of sub-level evaluators, respectively. Evaluation steps on all (activated) layers are implicitly given with the default evaluation approach for non-hierarchical decision graphs as used with *evaluation by expansion*. With *hierarchical evaluation*, this is ensured by explicitly forwarding events and signals to lower-level evaluators.

While both approaches conform to the basic semantics of sub-pattern components, their eventual results may differ in certain evaluation scenarios. This is due to the different handling of input parameters and output parameters. With *evaluation by expansion*, all parameter expressions are resolved and integrated into the different expressions of an expanded, thus non-hierarchical decision graph. As a consequence, input and output-parameter expressions are implicitly re-evaluated with each evaluation of an expression in which the respective parameter participates. By contrast, using the *hierarchical evaluation* approach, calculated input and output-parameter values are passed down and up the hierarchy and used in the receiving layer until new values are passed. Experience from real-world use cases showed that the latter approach better matches the expected semantics of a sub-pattern component. Moreover, while not discussed in the above section, input-parameter handling “by expansion” could relatively easily be implemented as a further option for hierarchical evaluation.

More decisive, *evaluation by expansion* is generally opposed to any operation that shall be restricted to exactly one layer of a hierarchical decision graph. Such an operation is, for instance, given through the “reset evaluation state” option as may be specified for action components (including *signals*). When using *evaluation by expansion*, this configuration must either be ignored (which necessarily distorts the semantics of a pattern definition), or information about which parts of a flattened decision graph belong to a certain layer of the original decision graph must be maintained. The latter approach again complicates the base evaluation approach, whose simplicity was considered a major advantage in comparison to the hierarchical evaluation approach.

Taken together, while a hierarchical evaluation approach requires considerable adaptations of the existing evaluation strategy, it offers greater flexibility and is certainly better suited for future extensions of the decision graph model. For the eventual implementation of hierarchical decision graphs we therefore opted for the *hierarchical evaluation* approach instead of the easier-to-implement, yet less flexible *evaluation by expansion*.

## 7.5 Example

In the following, we present a hierarchical decision graph for monitoring user behavior in an online-betting platform. We assume that an event-based application shall notify the betting provider’s fraud department if

- (i) a user repeatedly carries out high-stake short-term transactions, i.e., if the user pays into his account, places a single high-risk bet and immediately pays off in full, and
- (ii) the user’s total balance is above a certain threshold.

User behavior as described in the former condition is highly suspicious of fraud: The user is clearly focused on selected – possibly fixed – sports events. Also, the user seems to be afraid of being discovered, therefore trying to keep those periods where money is held by the betting provider as short as possible. The latter condition, by contrast, restricts alarms to those cases that are relevant in terms of monetary profit.

Table 7.1 shows a pattern definition for detecting above-described short-term transactions. Series-connected event conditions check if a user pays into a (near-) empty account, wins a bet and pays off in full within a user-defined time span; for both pay-in and cash-out, an impreciseness of 5% is considered. The bet’s odds as well as the paid-in amount can be tested against user-defined thresholds. For the sake of reusability, the pattern definition does not consider repetitiveness, but instead notifies any occurrence of the described event situation.

Table 7.2 shows a pattern definition for monitoring a user’s overall balance. Whenever a “Bet Won” event occurs, a score “Balance” for the user’s account is increased by the given revenue; reversely, the measure is decreased in case of “Bet Lost” events. When a measure exceeds a user-defined threshold, a signal is published.

Figure 7.9 shows the complete decision graph. On the left-hand side, the decision graph features a “High profit” sub-pattern component with a threshold of 5000 dollars. Users therefore qualify as potential fraudsters only if their overall gains are above that threshold. On the right-hand side, a score “Suspicious Transactions” counts the occurrence of “short-term transaction” event patterns (with a minimum pay-in amount of 100\$) per user account. A downstream condition tests the measure value against a threshold of three. Finally, if both the left and the right-hand side are fulfilled, an email is generated.

| Input Parameters | ID     | Type       | Validator  |
|------------------|--------|------------|------------|
|                  | Amount | Integer    | $x \geq 0$ |
| Odds             | Float  | $x \geq 0$ |            |

| Decision Graph   |
|--|
| <p>The decision graph consists of four nodes connected sequentially:</p> <ul style="list-style-type: none"> <li><b>Cash-in to empty account:</b> A 'Cash In' node with conditions: <math>CashInAmount &gt; \\$amount</math> and <math>TotalAmount / CashInAmount &lt; 1.05</math>. It has 'True' and 'False' paths.</li> <li><b>Bet won:</b> A 'Bet Won' node with condition: <math>Odds &gt; \\$odds</math>. It has 'True' and 'False' paths.</li> <li><b>Clear out account:</b> A 'Cash Out' node with 'Conditions (AND)': <math>TotalAmount / CashOutAmount &lt; 0.05</math> and <math>CashIn.Timestamp - Timestamp &lt; 24h</math>. It has 'True' and 'False' paths.</li> <li><b>Signal: Short-term transaction:</b> A 'Hit' node with 'Output Parameters': <math>Profit: CashOut.Amount - CashIn.Amount</math> and <math>Account: CashIn.Account</math>.</li> </ul> |

| Output Parameters | ID      | Type    |
|-------------------|---------|---------|
|                   | Profit  | Integer |
| Account           | Integer |         |

Table 7.1. Pattern Definition “Short-Term Transaction”

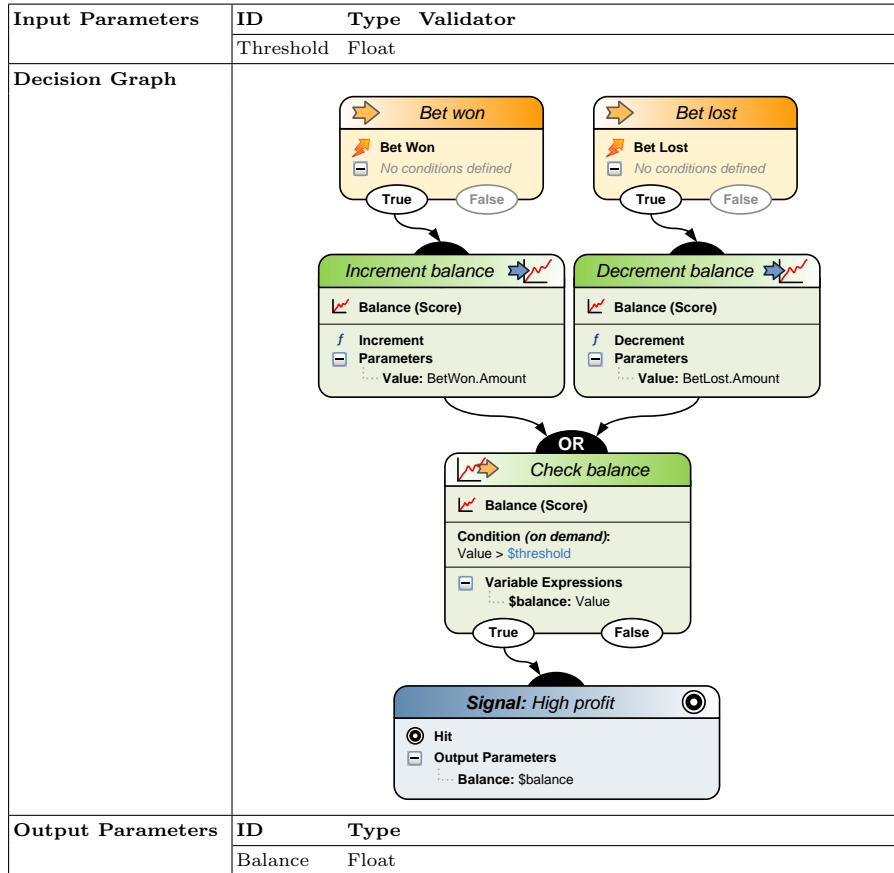


Table 7.2. Pattern Definition "Balance Above Threshold"

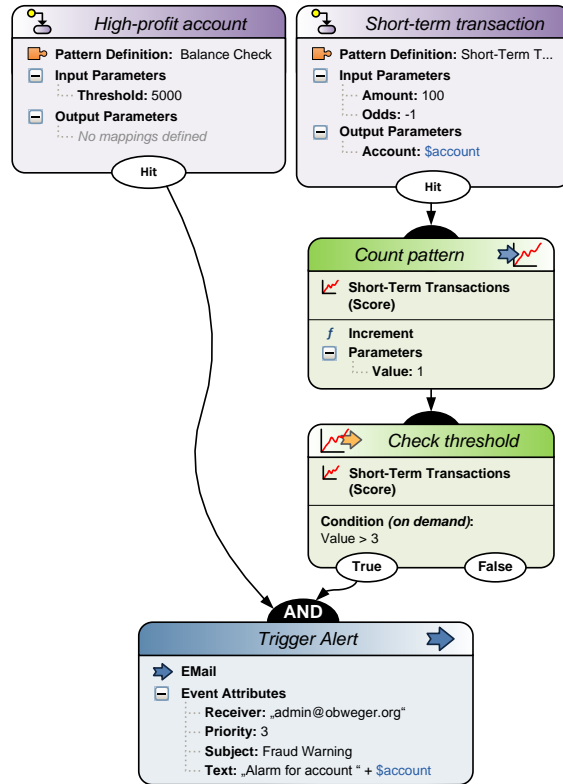


Figure 7.9. Rule Definition “Send Email on Suspicious User Behavior”



## Example

**Abstract** In the course of this thesis, a user-oriented rule management system for Complex Event Processing applications, along with extensions for hierarchical pattern modeling and entity-based state management, has been proposed. This chapter demonstrates the proposed framework using a real-world SARI application for *event-based service assurance*, which we understand as the proactive monitoring of business environments with the goal of detecting fault patterns and ensuring reliability and performance in a system landscape. The presented application is discussed following the model-driven view on SARI applications, with a particular focus on the application's infrastructural rules and sense-and-respond rule-management artifacts.<sup>1</sup>

### 8.1 Introduction

One of the major challenges observed in corporations with distributed, inhomogeneous IT landscapes is to manage the proper execution of IT processes and ensure the performance of potentially thousands of servers across disconnected data centers. Early in the history of enterprise computing, *IT Process Automation* has been developed to overcome these difficulties, and today it forms the nerve center of almost any larger company's IT infrastructure. Still, traditional IT process automation is driven by scheduled batch execution and lacks capabilities for flexible, event-driven execution. In recent times, Complex Event Processing has therefore been recognized as an important complementary technology to automation platforms (cf. [42]). Using CEP, adopters of IT process automation are capacitated to launch tasks in response to complex events, emitted from multiple, potentially disconnected sources and processed in near real time. While unified solutions are the exception rather than the rule, more and more vendors strive to integrate their core frameworks with proprietary or third-party event-processing systems.

---

<sup>1</sup> This chapter is based on the work of Obweiger et al. [88].



In this chapter, the proposed approach to user-oriented rule management is demonstrated by the example of a real-world SARI application for *event-based service assurance*, where SARI is applied as an extension to the *UC4 Automation Engine* [130]. We understand service assurance as the proactive monitoring of business environments with the goal of detecting fault patterns and ensuring reliability and performance in a system landscape, based on event data from both the technical infrastructure (including individual hosts, databases, and network connections) and the application layer of a corporation (which, besides the central automation platform, typically includes other enterprise systems such as Enterprise Resource Planning [ERP] or Customer Relationship Management [CRM] systems). As such, service assurance includes – yet, is not restricted to – disciplines such as Business Service Management (BSM) and Application Performance Management (APM). Service assurance in the described sense has, for instance, been discussed in the context of cloud computing [110].

The presented SARI application has been developed by members of the UC4 Senactive development team based on customer requests and input gathered from the company’s consulting, sales, and pre-sales personnel. It is distributed with *UC4 Decision* as an extension to the core automation platform and has been successfully set up at customers from different business domains.

## Outlook

The remainder of this chapter is structured as follows: Section 8.2 provides an overview of the described application scenario. In Section 8.3, Section 8.4 and Section 8.5 we discuss the event model, business entity model, and correlation model of the presented SARI application for event-based service assurance. In Section 8.6, the event processing model is presented. The application’s infrastructural rules, as well as the diverse parts of the application’s sense-and-respond rule management configuration, are presented in Section 8.7 and Section 8.8.

## 8.2 System Overview

Figure 8.1 sketches the described environment from a high-level perspective. The core element of a customer’s IT landscape is the UC4 Automation Engine, which provides all necessary facilities for defining and scheduling tasks, modeling dependencies between tasks, distributing tasks on a network of agents, and calculating forecasts. To support IT services and cloud computing use cases, the automation engine can be logically separated into so-called *clients*, where each client operates and is managed generally independent from all others. As

an exception, the ever-present *default client #0* (zero) enables super administrators to monitor executions in all non-default clients, but does not allow task executions by itself.

Connected to the automation platform, a collection of *agents* is responsible for actually carrying out tasks as provided by the automation engine. Agents are available for all current operating systems, as well as for a wide range of enterprise applications: While former are responsible for carrying out basic system operations such as file transfers and command-line actions, latter allow triggering application-specific operations such as, for example, starting and stopping an SAP job. The physical or virtual machine running an agent is referred to as *host*.

Third-party applications may eventually integrate with the automation engine using a message-bus- and web-service-based interface. Via the message bus, events such as the start and end of a task, as well as system messages such as the planned or exceptional shutdown of an agent, are published. Via web services, operations such as starting, pausing, and canceling a task are exposed.

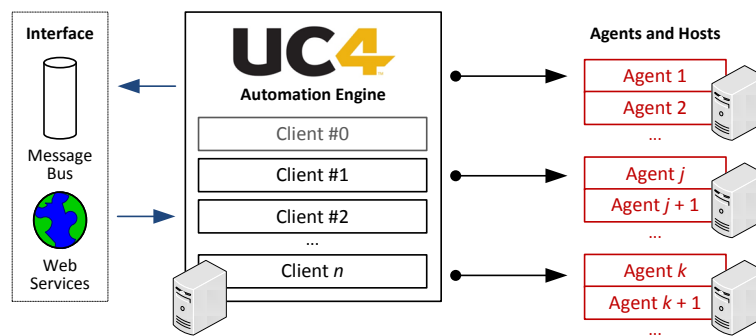


Figure 8.1. System Overview

## 8.3 Event Model

The application's event model contains an overall number of nine event types, from which three event types are input events (i.e., retrieved from the underlying source system), two are virtual events (i.e., created during event processing) and four are response events (i.e., sent to response event-adapters and translated into real-world actions). Table 8.1 summarizes the application's event types and their role in event processing.

| <b>Input Events</b>   |   |
|---|---|
| <b>Task Events</b>  | Task events represent the end of a task in the underlying automation engine and form the major input for detecting situations such as delayed task executions, manual cancellations of tasks, or recurring task failures. Via their event attributes, task events provide access to the represented task's type and instance ID, its activation, start and end time, its estimated runtime (which can be defined statically or to be calculated from past executions in the automation engine), its end status (e.g., finished normally or with an error), the responsible user account, and the executing agent, among others. |
| <b>Message Events</b>   | Message events represent noteworthy occurrences as are signified by the automation engine; for instance, a message could indicate the shutdown of a task-execution agent. Message events expose the represented message's message number and string, the concerned agent, user account, and task type (if available), among others.   |
| <b>Log-File Events</b>  | Log-file events are semantically related to message events, however, represent data as is written to the application's log files and contain detailed system-trace information such as execution times of system-internal activities or database transactions. Log-file events expose the represented log-file entry's number and string, the concerned task type (if available) and the path of the log-file from which the entry was pulled, among others.  |
| <b>Virtual Events</b>   |   |
| <b>System Checkpoint Events</b>   | System checkpoint events are generated within the application, and basically form an aggregation of a configurable number of incoming task events. These events contain, for instance, the number of failed tasks among the last $x$ tasks, the average overtime, etc. Reducing the overall number of task data, system checkpoint events can well be used for detecting trends.  |
| <b>Database Log-File Events</b>   | Database log-file events are special log-file events that signify database-related entries. In addition to the basic log-file event data, they provide event attributes for the type of database transaction and its exact duration.  |
| <b>Response Events</b>  |   |
| A collection of response events are provided for responding back to the underlying source system. These response events include <b>Email Events</b> , <b>Log-File Entry Events</b> (which cause a respective event adapter to write an entry to log file), <b>Command-Line Action Events</b> , and <b>Web-Service Call Events</b> . |   |

Table 8.1. Event Types

## 8.4 Business Entity Model

The proposed SARI application makes use of the reference business-entity provider implementation for *scores* as discussed in Section 6.4.1 and defines an overall number of 31 score types, from which seven are used as application-wide, persistent data structures and 24 are used internally and in memory only. In the present version of the application, scores of the former types are *not* supervised during run time using monitoring rules, but used exclusively for the ex-post analysis of the system.

## 8.5 Correlation Model

The presented application does not require event correlation and so comes with an empty correlation model.

## 8.6 Event Processing Model

Due to its focus on rule spaces, the presented application can be realized using a single event-processing map while remaining simple in structure and easy to understand. Figure 8.2 shows the application’s event processing model; for the sake of readability, input ports, output ports, and event channels are colored according to the event types they are concerned with.

On the input side of the event processing map (Figure 8.2a.), a collection of *sense event adapters* (Section 3.5.2) is applied to receive task events, message events, and log-file events from the underlying source system. Task events and message events are retrieved directly from the automation engine, which publishes the data as typed, comma-separated value tuples to a message bus. The respective adapters are based on configurable message transformers, where application developers can define a separator character and map the individual values to event types and event attributes. Log-file events are retrieved through a so-called *log-file adapter*, which basically monitors a number of text files by reading them and examining them for new lines at regular time intervals. Each line is then transformed into an event instance based on user-defined mappings similar to that of message transformers.

From the collection of sense event adapters, input events are forwarded to the “Filter” service (b.), where illegal and irrelevant events are eliminated; for example, task events with a run time of zero signify tasks that have actually been discarded by the underlying automation engine due to missing preconditions, and thus need not to be considered for subsequent event processing. The filtering service is implemented as a rule service, hosting infrastructural filtering rules for task events and message events. The applied rules are discussed in greater detail in Section 8.7, “Infrastructural Rules”.

All remaining input events are delivered to the “Message Enrichment” service (c.), where based on event-attribute values, additional data are retrieved from external sources and attached to events. For all types of input events, diverse message numbers and status codes are used to retrieve corresponding string representations from an external data source; albeit not immediately required for event processing, these data facilitate full-text search and easy-to-interpret

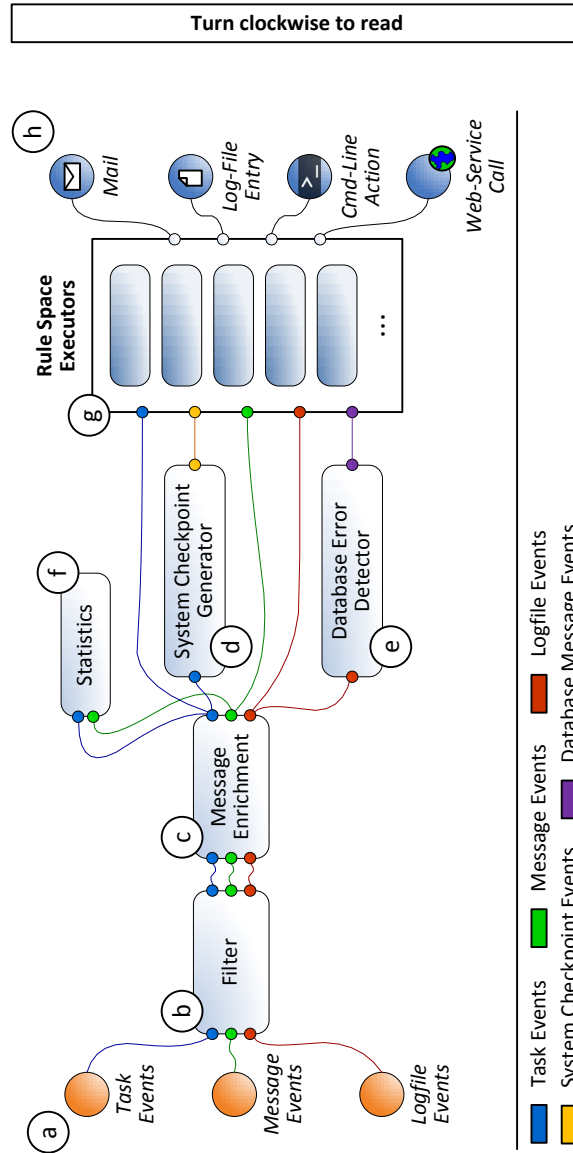


Figure 8.2. Event Processing Model

reports in the ex-post analysis of historic event data.<sup>2</sup> In case of log-file events, the type of the concerned task – if such is available – is extracted from the log-file entry; while basic parsing mechanisms are provided by the above-mentioned log-file adapter, more sophisticated strategies are required here due to legacy message formats.

From the message-enrichment service, preprocessed input events are then forwarded to the “System Checkpoint Generator” (d.), “Database Error Detector” (e.), and “Statistics” (f.) service according to their event types. In the system-checkpoint generator, system checkpoints events are generated for each  $x$  incoming task events using a custom .NET service implementation. In the database-error detector service, an infrastructural rule is applied to detect database-related log-file events by their message number and transform them into database log-file events. In the statistics service, infrastructural rules are applied to calculate application-wide aggregations (e.g., the number of tasks per host, task type, and end status) using scores. The application’s business-entity model, as well as the different infrastructural rules applied in this part of the application, are discussed in greater detail in Section 8.4 and Section 8.7, respectively.

Task, message, and (database) log-file events are eventually delivered to a parallel grouping of rule services, where each rule service is associated with exactly one rule space as discussed in Section 8.8. Although not depicted in Figure 8.2, each rule service is provided with exactly those events that are potentially relevant for the provided pattern-detection logic.

On the output side of the event processing map, a collection of *response event adapters* allows sending emails, writing to log files, executing arbitrary command-line actions, as well as web-service calls. Again, rule services are connected to all event adapters that are required for the provided set of action definitions.

## 8.7 Infrastructural Rules

The presented application includes an overall number of six infrastructural rules, of which two are run on the initial “Filter” service (Figure 8.2b.) to remove illegal and irrelevant events, one is run on the “Database-Error Detector” service (8.2e.) to detect database-related message events and three are run on the “Statistics” service (8.2f.) to update application-wide scores.

Table 8.2 describes these rules in greater detail. The decision graph of rule definition “Filter duplicate agent messages” is shown in Figure 8.3.

<sup>2</sup> While not discussed in the course of this chapter, the presented application persists noteworthy event instances and provides predefined search and visualization templates to be used in the *Event Analyzer* [124, 123].

| <b>Filter Service</b>                  |  |
|--|--|
| <b>Filter discarded tasks</b>          | <p>Filters task with a runtime of zero; such events have been discarded in the underlying automation engine due to missing preconditions and must not be considered for subsequent event processing. In the rule's decision graph, a condition component evaluates an expression</p> $\text{ToMillis}(\text{EndTime} - \text{StartTime}) = 0$ <p>on incoming task events. Connected to the "false" port of the condition, a response-event action re-publishes the triggering task event. The condition's <i>true</i> port is left unconnected.</p>  |
| <b>Filter duplicate agent messages</b> | <p>Filters duplicate <i>agent messages</i>. Agent messages signify agent-related occurrences such as planned or exceptional shutdowns and are generated once per <i>client</i> by the underlying automation engine. To avoid duplicate reactions, messages for all clients except of the ever-present <i>default client #0</i> are eliminated using a condition and a re-publishing response-event action similar to "Filter discarded tasks".</p>   |
| <b>Database Error Detector Service</b> |  |
| <b>Database error detection</b>        | <p>Detects database-related log-file events based on their message number and transforms them into "Database Log File" events, using a condition component and a response-event action component. In the response-event action component, all attributes of the triggering log-file event are mapped to attributes of the database log-file event to be generated, and additional attributes are derived from the triggering event using the string-manipulation functionalities of <i>EA Expressions</i>.</p>   |
| <b>Statistics Service</b>              |  |
| <b>Agent statistics</b>                | <p>Updates the number of activated, abnormally ended, and normally ended agents per host and weekday based on incoming message events. In the rule's decision graph, scores of type "Activated agents", "Abnormally ended agents" and "Normally ended agents" are incremented depending on the incoming event's message number, using a <i>case</i> component and three business entity actions. Key property values are extracted from the triggering message event.</p>  |
| <b>Message statistics</b>              | <p>Updates the number of errors per user, host, task name, and client, as well as the number of access denials per task name, based on incoming message events. In the rule's decision graph, scores of type "Error count" and "Access denied statistics" are updated if the message type is set to "Error" and if the message number indicates a denied access attempt, respectively. In both cases, a condition component is applied together with a business entity action to model the described functionality. Key property values are extracted from the triggering message event.</p> |
| <b>Task statistics</b>                 | <p>Updates the overall number of tasks and the number of delayed tasks per host, task name, client, and user, based on incoming task events. In the decision graph, scores of type "Tasks per host" and "Delay count" are updated with any occurrence of a task event and if a task's runtime exceeds its estimated runtime, respectively. In both cases, a condition component is applied together with a business entity action. Key property values are extracted from the triggering task event.</p>   |

Table 8.2. Infrastructural Rules

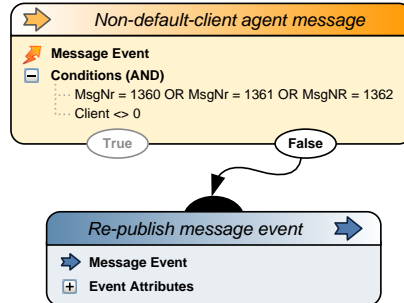


Figure 8.3. Rule Definition “Filter duplicate agent messages”

## 8.8 Sense-and-Respond Rules

The presented application for event-based service assurance includes an overall number of 36 pattern definitions and 13 action definitions, which are organized in six rule spaces and serve as a basis for an average of about three business patterns per pattern definition and about four business actions per action definition. In the following, we give an overview of the application’s different rule spaces in Section 8.8.1 and particularly focus on a selected rule space “Runtime monitoring” – basically concerned with the detection of tasks that outrun their scheduled execution time – when it comes to concrete pattern detection and reaction logic in Section 8.8.2 and Section 8.8.3.

### 8.8.1 Rule Spaces Overview

In the proposed approach to sense-and-respond rule management, rule spaces serve as the basic work space for business operators and expose pattern detection and reaction logic according to the different operational tasks in an application. The presented SARI application for service assurance offers six rule spaces, where each is concerned with a different aspect of the underlying automation engine and mapped to exactly one rule service in a parallel grouping of rule-space executors as shown in Figure 8.2: *Runtime monitoring*, *access denial monitoring*, *log file and error message monitoring*, *agent monitoring*, *health monitoring*, and *database protection monitoring*.

Table 8.3 provides an introduction to said rule spaces and their specific role in the presented application.



|  |   |
|--|---|
| <b>Runtime Monitoring</b>                    | An important indicator for service assurance are task runtimes. Runtime monitoring rules allow monitoring task runtimes for exceptional delays. Details on the rule space's pattern definitions and business patterns, action definitions, and business actions are discussed in greater detail in Section 8.8.2 and Section 8.8.3 of this chapter.   |
| <b>Access Denial Monitoring</b>              | Messages about access denials on objects of the automation engine are important hints for security violations. Access-denial monitoring rules allow detecting conspicuities in access denial messages, such as a certain user trying to access an object unsuccessfully multiple times within a short period, or with recurring retries after a break.  |
| <b>Log File and Error Message Monitoring</b> | In case of a failure, manifold information is available in the server log files, indicating what led to the problem. Log file and error message monitoring rules allow analyzing these log files in real-time, to detect early indicators for system troubles. Frequent errors are recognized as well as an increase in the total number of error log entries. Known messages and indicators might also trigger an immediate alert.   |
| <b>Agent Monitoring</b>                      | Agents need to be constantly up and running in order to execute the scheduled processes. Agent monitoring rules allow monitoring agents for availability and performance. Shutdown and failure patterns are identified as well as trouble agents with frequent shutdowns or insufficient response times. In addition, the rule space allows monitoring if certain tasks fail in recurring manner on an individual host.   |
| <b>Health Monitoring</b>                     | Indicators for system health include execution delays (indicating unbalanced load), high rates of task failures, or cumulations of error messages, among others. While individual triggers might not yet alert, trends in the number of failures or delays might show changes in the application stability. Health monitoring rules target exactly these trends, alerting upon increasing numbers of also minor delays, slight increases in error occurrences and task failures, or a combination of all three. |
| <b>Database Protection Monitoring</b>        | The automation engine keeps track on every action in an underlying database. Also, messaging is relayed over the database to keep it reliable and persistent. Thus, the database performance is vital for the performance of the automation engine. Database protection monitoring rules allow monitoring in real-time for indicators on potential database issues. These include log entries on critical, long-running database calls, error messages, timeouts, or even deadlocks.                            |

Table 8.3. Rule Spaces

### 8.8.2 Runtime Monitoring: Pattern Definitions, Business Patterns

In the presented SARI application, the rule space for *runtime monitoring* provides diverse pattern definitions based on the occurrence of delayed tasks, where the actual execution time outruns the scheduled execution time by more than a specified percentage. Specialized pattern-detection logic allows detecting delays on certain hosts, at certain days of a week, or at certain times. With large companies typically focusing on recurring delays rather than individual outliers, other pattern definitions allow specifying a certain number of runtime exceedances to be reached; depending on the pattern definition, the concerned tasks must occur in total or within a specified time frame.

Table 8.4 and Table 8.5 summarize the pattern definitions of the runtime monitoring rule space. Pattern definition “Task runtime exceeded per type” is shown in full detail in Table 8.6.

| <b>Task runtime exceeded</b>                         |   |
|--|---|
| <b>Description</b>                                   | Triggers whenever an individual task event occurs with a relative runtime delay greater than a user-specified threshold. Relative runtime delays are calculated from estimated and actual runtimes of signified task executions in the underlying automation engine.  |
| <b>Decision Graph</b>                                | A condition component evaluates an EA Expression<br><br>$\text{ToMillis}(\text{EndTime} - \text{StartTime}) / \text{ToDouble}(\text{ERT}) > \$\text{AcceptedRatio}$ on incoming task events. Directly connected to the “true” port of the condition, a signal component signifies the detection of delayed task executions.   |
| <b>Input Parameters</b>                              | Maximum accepted relative runtime delay   |
| <b>Output Parameters</b>                             | Task type, agent, execution ID, start time, actual runtime, estimated runtime   |
| <b>Business Patterns</b>                             | Available for different levels of delay, where each level is associated with a different threshold. For maximum flexibility, an additional business pattern enables authorized users to freely specify the demanded threshold.  |
| <b>Average delay rate over threshold</b>             |   |
| <b>Description</b>                                   | Triggers whenever an individual system checkpoint event occurs with an average relative runtime delay ratio greater than a user-specified threshold.  |
| <b>Input Parameters</b>                              | Maximum accepted average relative runtime delay   |
| <b>Output Parameters</b>                             | Average relative runtime delay  |
| <b>Business Patterns</b>                             | As with “Task runtime exceeded”, business patterns for different levels of delays, as well as a business pattern that can be freely specified by the user, are provided.  |
| <b>Task runtime exceeded per type</b>                |   |
| <b>Description</b>                                   | Triggers whenever a user-specified number of tasks of a certain task type occur with a relative runtime delay greater than a user-defined threshold. No time window is specified for these occurrences.   |
| <b>Decision Graph</b>                                | A condition component evaluates to true whenever incoming task events fulfill the specified runtime condition. In response, a rule internal score is incremented for the task’s type and tested against the specified quantity threshold. If the threshold is exceeded, the signal is activated and the score is reset.   |
| <b>Input Parameters</b>                              | Number of tasks, maximum accepted relative runtime delay  |
| <b>Output Parameters</b>                             | Task type   |
| <b>Business Patterns</b>                             | See above.  |
| <b>Task runtime exceeded per type in time window</b> |   |
| <b>Description</b>                                   | Detects delayed tasks per task type similar to “Task runtime exceeded per type”, but extends the original pattern-detection logic by a (repeating) time window. (Note that this time window can be set to infinite to imitate the behavior of “Task runtime exceeded per type”; however, a separate pattern definition is provided to optimize event-processing performance.) |
| <b>Decision Graph</b>                                | Extends the original decision graph by an additional business action condition between the initial condition component and the score update. Here, the score’s initialization time stamp is tested against null (no prior reset) and the triggering event’s time stamp. If a new time window is reached, the score value is reset before being incremented.                   |
| <b>Input Parameters</b>                              | Number of tasks, maximum accepted relative runtime delay, time window   |
| <b>Output Parameters</b>                             | Task type, beginning of current time window   |
| <b>Business Patterns</b>                             | See above; different predefined time windows  |

Table 8.4. Pattern Definitions

|   |
|---|
| <b>Task runtime exceeded per agent</b>  |
| <i>Similar to "Task runtime exceeded per type", but aggregates delayed tasks based on their agent.</i>                |
| <b>Task runtime exceeded per agent in time window</b>   |
| <i>Similar to "Task runtime exceeded per type in time window", but aggregates delayed tasks based on their agent.</i> |

**Table 8.5.** Pattern Definitions (continued)

|                          |                 |             |                  |
|--------------------------|-----------------|-------------|------------------|
| <b>Input Parameters</b>  | <b>ID</b>       | <b>Type</b> | <b>Validator</b> |
|                          | Accepted Delay  | Float       | $x \geq 0$       |
|                          | Number of Tasks | Integer     | $x \geq 1$       |
| <b>Decision Graph</b>    |                 |             |                  |
| <b>Output Parameters</b> | <b>ID</b>       | <b>Type</b> |                  |
|                          | Task Type       | String      |                  |

**Table 8.6.** Pattern Definition "Task runtime exceeded per type"

### 8.8.3 Runtime Monitoring: Action Definitions, Business Actions

On its action-definition side, the runtime monitoring rule space provides diverse email-based actions for notifying responsible departments within a company. As overload situations may often be resolved automatically through additional resources or rescheduling of tasks, the rule space furthermore provides a variety of so-called *system actions*: Based on a web-service call, they directly feed back into the automation engine, where they may request additional resources, pause or cancel running tasks, or delay the execution of scheduled ones.

Table 8.7 summarizes the action definitions of the runtime monitoring rule space. Table 5.2 and Table 5.4 show the action definition “Start automation engine object” along with an exemplary business action.

## 8.9 Discussion

The presented application for event-based service assurance successfully demonstrates the applicability of our approach in the context of a real-world use case. Infrastructural rules are applied for the filtering and transformation of raw input events, as well as the calculation of application-wide event-data aggregations using the concept of business entities. Together with the presented set of event adapters and event services, these rules establish an event-based image of the underlying automation platform, including all task executions, messages, and log-file entries. End-user-defined sense-and-respond rules set up on this image and can be added and removed safely without affecting the proper functioning of the application. The application’s pattern and action definitions enable domain experts to detect and counteract exceptional situations such as bottlenecks, policy violations, or failures and are organized in a total number of six rule spaces.

Given a high-level functional specification, the described application could be implemented by members of the UC4 Senactive development team within eight man-days using the above-presented front-end tools. This time frame includes several test installations of the presented application, the evaluation of these installations against prepared test data, and subsequent refactorings. The provided front-end tools, as well as the changed application architecture (which is now centered around the concept of rule spaces), were well received by the involved team members.

As a negative point, it was remarked that the given 1-to- $n$  relationship between rule spaces and business-level building blocks (meaning that a business-level building block is part of exactly one rule space) may be too restrictive in case of general-purpose reaction logic. In the described application, business actions

such as “sending an email to the system administrator” were, in fact, needed in almost any of the application’s rule spaces. This forced the application developers to repeatedly duplicate these entities. Future research will therefore reconsider the said relationship.

| <b>Mail notification</b>   |   |
|--|---|
| <b>Description</b>   | Generates events of type “Mail action”, eventually causing downstream email response adapters to send emails with specified configurations.   |
| <b>Response Event Template</b>   | Being based on the “Mail action” event type, the response event template maps input parameters to the sender, recipient, subject, and body event attributes. SMTP-server- and user-account-related attributes (like host, port, use of SSL, user name, and password) are assigned application-wide constants that are to be defined during set up.  |
| <b>Input Parameters</b>  | Sender, Recipient, Subject, Body  |
| <b>Business Actions</b>  | Typically available for different departments of a company, with pre-defined receiver addresses. Note that email actions will typically incorporate output-parameter values in their subject and/or body part; thus, to support standard emails with a specified subject/body format, <i>prepared bindings</i> and <i>templates</i> will be needed in many scenarios.   |
| <b>Start automation engine object</b>  |   |
| <b>Description</b>   | Triggers a method invocation on the web-service interface of the underlying automation engine, causing the engine to start an instance of a user-defined task type.   |
| <b>Response Event Template</b>   | Based on an event type “Web Service Action”, where the exact path to the web service description is retrieved from an application-wide resource string, the method to be invoked is defined as a constant string (“executeObject”) and the “Arguments” attribute is concatenated based on a user-defined input parameter (providing the task to be started) and a number of application-wide resource strings (providing predefined authentication data). |
| <b>Input Parameters</b>  | Task type   |
| <b>Business Actions</b>  | Available for different tasks that help improving the overall task-execution performance of the system, e.g., by setting up and incorporating an additional host.   |
| <b>Pause automation engine object</b>  |   |
| <i>Similar to “Start automation engine object”, but pauses a running task based on a user-defined task ID.</i>   |   |
| <b>Resume automation engine object</b>   |   |
| <i>Similar to “Start automation engine object”, but resumes a paused task based on a user-defined task ID.</i>   |   |
| <b>Cancel automation engine object</b>   |   |
| <i>Similar to “Start automation engine object”, but cancels a running or paused task based on a user-defined task ID.</i>  |   |
| <b>End automation engine object</b>  |   |
| <i>Similar to “Start automation engine object”, but ends a running or paused task based on a user-defined task ID. Unlike cancellation, ending a task results in an orderly shutdown where possible post-execution instructions are performed.</i> |   |

**Table 8.7.** Action Definitions

## Case Study

**Abstract** This chapter illustrates the structure and results of a customer project in the manufacturing domain, where the proposed rule-management framework was successfully applied in the context of event-based service assurance. In the investigated project, the standard application for event-based service assurance was extended towards enhanced hardware-level monitoring by a vendor-side consultant and a technically experienced, customer-side infrastructure analyst. The readily-prepared sense-and-respond rule-management system was made available to technically less experienced users. We analyze the observed distribution of tasks and responsibilities and discuss possible implications to future research.

### 9.1 Introduction

Since implemented within Sense-and-Respond Infrastructure (SARI) [114], the proposed rule-management framework and its extensions have been successfully applied in use cases from different business domains, including finance, e-commerce, and logistics. In this chapter, we illustrate the structure and results of a customer project at a leading manufacturer of agricultural machinery. In the investigated project, SARI is used in the context of *event-based service assurance* as discussed in great detail in the previous chapter. Our case study is based on a series of interviews with the responsible pre-sales engineer at UC4 Software GmbH and an internal experience report [9] provided by the customer.

## Outlook

The remainder of this chapter is structured as follows: Section 9.2 and Section 9.3 illustrate the project's environment and objective in greater detail. In Section 9.4, the structure of the project is presented. An overview of the developed SARI application is provided in Section 9.5. Section 9.6 concludes our case study and discusses possible implications to future research.

## 9.2 Project Environment

The customer operates two physical computer centers, housing six IBM zSeries mainframes, over 850 servers ranging from simple two-processor to high-end 32-processor machines, and administrating over 1000 Terabytes of data with differing availability and redundancy requirements. Several thousand servers are located at business units and company facilities all over the world.

At the time of writing, the customer uses the UC4 Automation Engine [130] to automate IT processes across this complex and inhomogeneous environment. Used primarily as a job scheduling engine, tasks are executed to steer enterprise resource planning (ERP), financial, and human resources (HR) applications, to control the company's mainframe computers, and to manage database backups. The UC4 Automation Engine environment is staged into a test system (running about 150,000 tasks per month), a development system (running about 1.2 million tasks per month) and a production system (running about 2.5 million tasks per month). These systems distribute tasks on a total number of 128 agents, from which 96 agents execute SAP R/3 tasks, 24 agents run SAP BI tasks, four agents run on Unix machines and another four run on Windows machines. As the system grows in complexity and size, the customer plans to evolve UC4 Automation Engine into a full-fledged workload automation solution, then controlling more than 2000 agents.

At the customer, two persons are responsible for the administration of the company's Automation Engine installations. These are a senior infrastructure analyst with about 10 years of experience with the Automation Engine and strong technical skills in general (referred to as "infrastructure analyst" in the remainder of this section), as well as a junior assistant (in the following referred to as "assistant").

### 9.3 Problem

The infrastructure analyst regularly experienced situations in which the performance of the Automation Engine decreased without obvious reasons. These decreases primarily manifested in longer task-execution times, but also in weak response times of the system's front-end tools. In such situations, the infrastructure analyst relied on input from the different support and service groups of the enterprise (e.g., the database group, the network group, or the server group) about recent performance bottlenecks in their particular part of the infrastructure.

The aim of the observed customer project was to collect and analyze performance data in a continuous and automated manner, therewith enabling the infrastructure analyst to proactively investigate shortages in disk space, memory, CPU performance, or network response time, and take appropriate actions before actual problems arise.

### 9.4 Project Structure

The customer purchased UC4 Decision and UC4 Insight (the commercial distribution of SARI's Event Analyzer [124, 125, 123]) in July 2011 to cope with the afore-mentioned problems. UC4 Decision was installed along with the prepared standard application for *event-based service assurance* (see Chapter 8 for further details) by pre-sales personnel and introductory trainings were given to the infrastructure analyst. The standard application was then used for the ex-post analysis of the customer's IT landscape using UC4 Insight; real-time monitoring using event-pattern rules was not employed at this time.

In September 2011, a second on-site workshop took place to further tweak the existing application to the particular needs of the customer and to enable the sense-and-respond-based detection of upcoming resource bottlenecks. This second workshop was held by a consultant specialized in Complex Event Processing and can be roughly separated into three phases as discussed in the following.

#### Phase 1: Event-Processing Infrastructure

In a preceding requirements analysis, it had become clear that the required event-processing functionality goes beyond the installed standard application for event-based service assurance; not only on the level of building blocks for sense-and-respond rule management, but also on the level of event types and the integration of SARI with the underlying Automation Engine. In the first



phase of the workshop, the consultant therefore extended the existing application by new event types, event adapters, event services, and infrastructural rules. These extensions prepare the event-data input for a newly-created rule space “Performance Monitoring”, which is hosted on a single rule service and serves as the point of integration between the extended event-processing infrastructure of the application and the end-user defined event-processing logic.

### **Phase 2: Building Blocks**

Provided a readily-preprocessed image of the underlying source system, the objective of the second phase was to set up the building blocks for sense-and-respond rule management. The consultant and the infrastructure analyst defined an overall number of four pattern definitions and corresponding business patterns, from which the first pair was modeled by the consultant and the last three pairs could be modeled by the infrastructure analyst alone. Business actions and templates were modeled in similar procedures.

### **Phase 3: Rule Instances**

In the third and final phase of the workshop, the UC4 Decision Web Client was installed on Apache Tomcat [10] and rule instances were created according to the current processing requirements of the company. These rule instances remained relatively stable until today, however, could easily be changed whenever new processing requirements shall arise. The web client is accessible to the infrastructure analyst and the assistant, who was little involved in the design and modeling of the extended SARI application but received introductions to the web client and UC4 Insight.

Although generally available, the infrastructure analyst does not currently make use of the standard rule spaces for event-based service assurance. An introduction to these rule spaces, as well as a further extension of the application towards enhanced monitoring of SAP systems, will be subject to a future workshop.

## **9.5 Application Overview**

In the course of the investigated project, several extensions were made to the standard application for event-based service assurance. In the following, we discuss the extensions towards rule-based monitoring the customer’s IT landscape, which include an additional event processing map, four new pattern definitions, as well as a new rule space. Other extensions concern the ex-post analysis of the system and are outside the scope of this discussion.

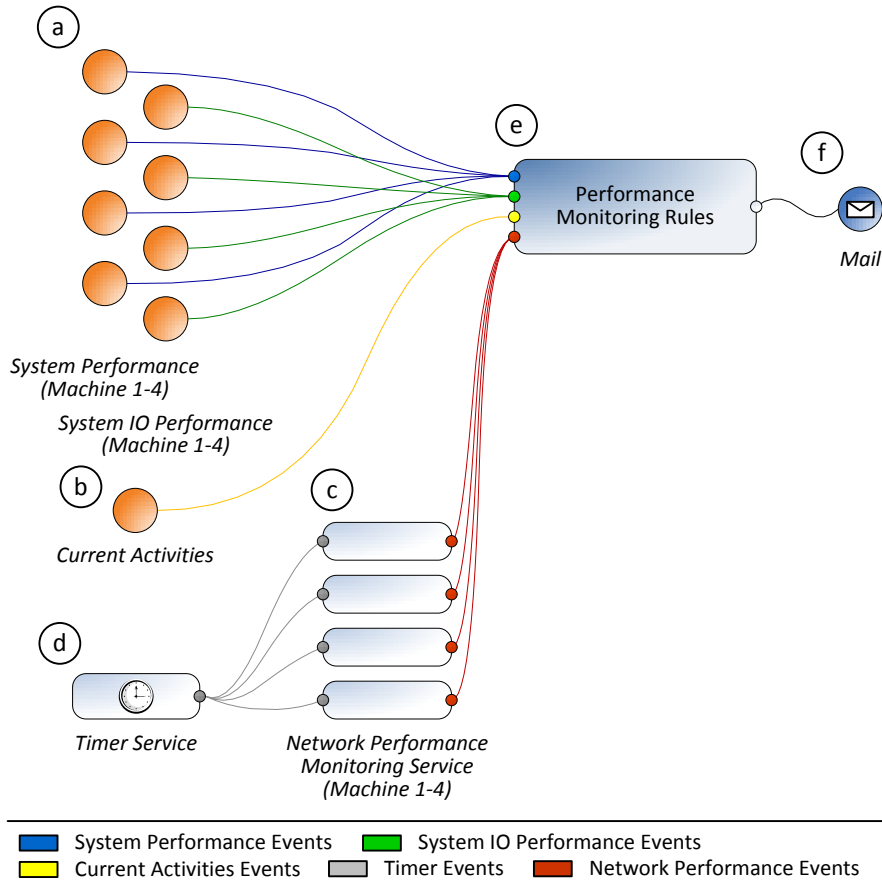


Figure 9.1. Event Processing Model

### Event Processing Model

Figure 9.1 shows the newly-created event processing map.

On the input side, eight sense event adapters are applied to receive “System Performance” and “System IO Performance” events from the four Windows machines of the customer’s IT landscape (Figure 9.1a.). System performance events contain the machine’s current CPU load, free memory, and used memory. System IO performance events contain the machine’s average physical disk queue length, average logical disk queue length, and available hard disk space. Both event types are generated at regular time intervals based on Windows’ performance counters; the respective sense event adapters are available out of the box with UC4 Decision. Another sense event adapter (b.) retrieves the current number of *activities* – i.e., the number of tasks in execution at

this particular point in time – for each client of the supervised UC4 system. These data are pulled from the automation engine’s database at regular time intervals using a standard SQL adapter and published in the form of “Current Activities” events.

In parallel, four event services generate “Network Performance” events for the monitored Windows machines (c.). Network performance events contain the response time of a monitored machine and are generated based on “ping” commands executed at the event service. The application’s network-performance monitoring services are custom .NET implementations and triggered at regular time intervals by standard “Timer” events as generated by a respectively-configured timer service (d.).<sup>1</sup>

System performance, system IO performance, and network performance events are forwarded to the “Performance Monitoring” rule service (e.), which serves as a host for the extended application’s novel “Performance Monitoring” rule space. On the output side of the event processing map, a single response event adapter for emails processes the output of the application’s sense-and-respond rules (f.).

### **Sense-and-Respond Rule Management**

The extended application includes four new pattern definitions as shown in Table 9.1. All these pattern definitions are associated with the new “Performance Monitoring” rule space, where exactly one business pattern is defined per pattern definition. From the standard application’s existing action definitions, the “Email” action definition is added to this rule space; again, exactly one business action is defined. To provide pattern-specific default configurations for this business action, templates are defined for all business patterns.

## **9.6 Discussion**

Although limited in size, the investigated use case well demonstrates the applicability and utility of the proposed rule-management framework in real-world business scenarios. The project was successfully implemented within seven days (two of them via web conferencing, five at customer-site) and yielded positive feedback from both the involved consultant and the customer-side representative, a senior, technically versed infrastructure analyst. In the words of the

---

<sup>1</sup> Network performance events would usually be generated using a respective event adapter. However, for reasons that go far beyond the scope of this thesis, it is currently easier to integrate a custom event-service implementation than to integrate a custom event-adapter implementation.

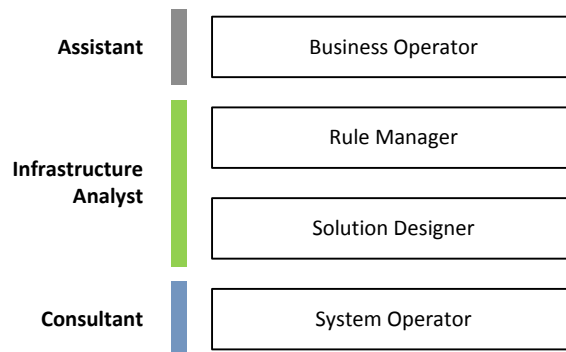
| <b>Network response time exceeded</b> |   |
|---------------------------------------|---|
| <b>Description</b>                    | Triggers whenever two successive network performance events show a network response time greater than a user-specified threshold.   |
| <b>Decision Graph</b>                 | A condition component tests the network response time of all incoming network performance events against the user-specified threshold. If the threshold is exceeded, the current value of a score “Network response time” is retrieved for the triggering event’s “Host” attribute and also tested against the specified threshold. If this condition evaluates to true, a signal component is activated. In any case (i.e., if the initial condition evaluates to false, if the business entity condition evaluates to false, or if the business entity condition evaluates to true), the “Network response time” score is updated to the most recent network response time as is available from the triggering event. |
| <b>Input Parameters</b>               | Maximum accepted network response time  |
| <b>Output Parameters</b>              | Host name, maximum accepted network response time   |
| <b>CPU load exceeded</b>              |   |
| <b>Description</b>                    | Triggers whenever two successive system performance events show a CPU load greater than a user-specified threshold.   |
| <b>Decision Graph</b>                 | Analog to “Network response time exceeded”, but based on “System performance” events and the CPU load attribute.  |
| <b>Input Parameters</b>               | Maximum accepted CPU load   |
| <b>Output Parameters</b>              | Host name, maximum accepted CPU load  |
| <b>Disk space critical</b>            |   |
| <b>Description</b>                    | Triggers whenever disk space falls below a user-specified percentage.   |
| <b>Decision Graph</b>                 | A condition component evaluates to true and activates the pattern definition’s signal whenever an incoming system IO performance event shows a free disk-space value below the user-specified percentage.   |
| <b>Input Parameters</b>               | Minimum accepted disk space   |
| <b>Output Parameters</b>              | Host name, free disk space  |
| <b>Number of activities exceeded</b>  |   |
| <b>Description</b>                    | Triggers whenever the number of activities exceeds a user-specified threshold.  |
| <b>Decision Graph</b>                 | A condition component evaluates to true and activates the pattern definition’s signal whenever an incoming current activities events shows a value greater than the user-specified threshold.   |
| <b>Input Parameters</b>               | Maximum accepted activities   |
| <b>Output Parameters</b>              | Client, Activities count, Maximum accepted activities   |

Table 9.1. Pattern Definitions

analyst, the project successfully “enable[d] [the customer] to create custom alerts to inform [...] about potential problems so they can be addressed before they impact the system or users” [9]. The customer thereby benefits from the full-fledged rule-management architecture not so much because of a particular need for rapid rule changes – in fact, the created rule instances remained relatively stable until today – but because of the increased usability for end users of the system. In an experience report, the involved infrastructure analyst emphasized that the UC4 Modeling Studio “will only need to be referenced when adding or removing entries from the [SARI application]” [9]. Also, the established architecture enables technically inexpert employees to investigate the underlying system both in real time (through sense-and-respond rules) and retrospectively (through prepared visualization templates in UC4 Insight).

### User Roles

A particular focus of our case study was on the distribution of tasks and responsibilities among the involved individuals. Figure 9.2 shows the involved individuals along with their user roles in the investigated business environment, as can be concluded from the above-described project phases.



**Figure 9.2.** User Roles and Implementations

In the investigated use case, the consultant acts as a *system operator* in designing and setting up the event-processing infrastructure of the extended SARI application. At the time of the workshop, this could not be accomplished by customer-side personnel; although significant skills in SARI application development were acquired by the infrastructure analyst, it would have been difficult, if not impossible, for him to develop custom event-service implementations or to write SQL statements that retrieve internal data from the underlying Automation Engine. A consultant-based implementation of the system operator role seems reasonable due to the greater stability of an application's event processing infrastructure. Still, if further extensions to the application are demanded, it may be more efficient to train a customer-side technician.

The infrastructure analyst implements the *solution designer* role as well as the *rule manager* role, i.e., is responsible for the design, modeling and administration of both event-level and business-level building blocks. A combination of these roles seems natural in environments where a senior domain expert has great technical skills, and/or where the number of business-level building blocks is very small. Both is given in the investigated use case.

The assistant – technically less experienced than the infrastructure analyst and little involved in the design and implementation of the investigated SARI application – is expected to implement the *business operator* role as soon as changes in the rule set become necessary. As such, the assistant is enabled

to create, deploy and administrate sense-and-respond rules based on the pre-defined building blocks and templates in a way that fully abstracts from the underlying integration and rule-evaluation logic. In the medium run, it seems reasonable to grant well-defined business-operator rights to larger user groups, e.g., to members of the customer's different support and service groups.

We conclude that the observed distribution of task and responsibilities complies in a very natural manner with the proposed system of user roles. Still, it is imperative for us to investigate the corporation between user groups in larger projects. Such investigations are subject to near future work.

### Rule Management Facilities

The proposed rule-management facilities proved suitable for creating, deploying, and administrating event-pattern rules in the investigated application scenario. Our case study also showed, however, that the given 1-to- $n$  relationship between event-level building blocks and business-level building blocks does not necessarily meet the requirements of real-world business environments. In the presented SARI application, exactly *one* business-level building block is defined per event-level building block. In such situations, a strict separation of these concepts can easily become counterintuitive to end users and be perceived as an administrative overhead. As reported by the consultant, it totally made sense for the infrastructure analyst to define natural-language representations of the created event-level building blocks. It was, however, not clear to him why this must happen in separate entities.

As a future improvement of the proposed framework, we therefore plan to introduce an optional *default business-level building block* per event-level building block. This default block could be defined and administrated as part of the event-level building block to which it belongs, and implicitly be created whenever the event-level building block is added to a rule space. Further business-level building blocks could be defined as usual.



## Conclusion

### 10.1 Summary

In this thesis, we presented a novel rule-management framework for the generic event-processing system Sense-and-Respond Infrastructure (SARI) [114]. The primary goal of this framework is to cater to the needs of IT experts as well as business users, both of which shall be supported in the creation, deployment, and administration of event-pattern rules according to their specific needs and requirements. This is achieved through two complementary, yet clearly decoupled rule-management approaches for IT experts and business users, referred to as *infrastructural rule management* and *sense-and-respond rule management*. Both approaches eventually result in *decision graphs*, which form the basis for any rule-based event processing in SARI and are directly interpretable by SARI's rule engine. However, while infrastructural rule management seeks to support the management of these decision graphs with minimal administrative overhead, sense-and-respond rule management builds upon a sophisticated template model that hides underlying complexity from business users.

The presented framework has been extended by approaches to *entity-based state management* – making SARI applicable also in entity-centric business environments – and *hierarchical pattern modeling*, which facilitates reuse of pattern-detection logic on the level of event patterns. Our research is framed by a model-driven reference description of SARI, which is intended to serve as a basis for future research and facilitate communication, interchange, and cooperations within the event-processing community.

Infrastructural rule management denotes the management of processing logic through IT experts and technically versed domain experts. It particularly focuses on

- (i) *expressiveness* – i.e., rule-authoring facilities must be expressive enough to establish a proper image of the underlying source systems



- (ii) *efficiency of use* – i.e., the creation, deployment, and administration of rules should be as immediate, clear, and transparent as possible
- (iii) *full and system-wide access* – i.e., users must be able to align the implemented processing logic with all other elements of an application

In the proposed workflow, so-called *system operators* model event-pattern rules in a single, comprehensive model, in parallel and fully integrated with the other elements of an application’s event-processing infrastructure. The resulting *rule definitions* are immediately interpretable to SARI and can be deployed by directly assigning them to one or more of the application’s rule services. System operators are provided a comprehensive, power-user-oriented IDE for event-based applications, in which rule definitions are managed along with event types, correlation sets, business entities, and event processing maps.

Sense-and-Respond rule management denotes the management of business logic through business users. Its primary focus is on *ease of use*, i.e., the process of creating, deploying, and administrating an event-pattern rule must fully abstract from the underlying complexity of an event-processing framework. Other emphases are

- (i) *personalized rule management* – i.e., the system must support a notion of rule ownership
- (ii) *rule activation and scheduling* – i.e., it shall be possible to pause and resume the execution of a rule either manually or based on a calendar
- (iii) *hot deployment* – i.e., it shall be possible to create and remove rules at any point in time, without having to restart the entire system
- (iv) *security* – i.e., it shall be possible to clearly restrict the competences of a business user

The proposed workflow for sense-and-respond rule management is rooted in a two-layered model of “building blocks” of pattern-detection and action logic. These building blocks fully abstract from underlying complexity and can be assembled to concrete event-processing logic according to the given processing needs of a company.

In the first step of this workflow, well-trained power users of a system prepare so-called *event-level building blocks* of pattern-detection logic and reaction logic according to the general requirements of a company. These building blocks encapsulate event-processing logic in a form that is interpretable to SARI; however, they are designed with focus on reusability across different application scenarios and often too generic to be directly usable for domain experts. In the second step of the workflow, senior domain experts therefore refine these building blocks into less flexible, yet easier-to-use *business-level building blocks*. The basic aim of these building blocks is to simplify the instantiation of an underlying event-level building block with respect to the specific use case in which it

is to be used. Most notably, this is achieved by setting input parameters or restricting their input domains. Eventually, business-level building blocks define a high-level, textual description of the represented event-processing logic, with placeholders for all input parameters. In the third and last step of the workflow, technically inexperienced domain experts can assemble business-level building blocks based on their textual representations to a natural language sentence of the form “if pattern, then action(s)”. The deployment of resulting event-pattern rules is performed transparently through the concept of rule spaces, which group building blocks and implicitly associate resulting event-pattern rules with one or more rule services. For the creation and administration of sense-and-respond rules, business users are provided a simplified web application, through which event-pattern rules can be assembled using a wizard-based interface.

Entity-based state management addresses the problem of monitoring a complex, durable entity – e.g., a counter, a customer, or a queue – which state is accessible only in the form of continuous, low-level update events. We extended SARI by the concept of *business entity providers*, which encapsulate arbitrary state-calculation logic and manage state in the form of typed, application-wide, identifiable data structures. These so-called *business entities* can then be updated and monitored for exceptional states using standardized interfaces. A particular focus of our work was on the seamless integration of business entities with existing rule management and evaluation facilities. In a generalized correlation model, semantic relationships can now be expressed not only between events, but also between events and business entities, and between different kinds of business entities. In an extended decision-graph model, specialized rule components allow invoking update operations and evaluating Boolean conditions on those business entities that are related to the active correlation session. Entity-based state management was furthermore designed to naturally complement the proposed differentiation into infrastructural and sense-and-respond rule management: Infrastructural rules can be applied to keep business entities up-to-date and in-sync with possible real world correspondences. Sense-and-respond rules monitor business entities as a part of the event-based image of underlying source systems and trigger actions in response to noteworthy states. It is essential to note, however, that such architecture is mandatory; in many cases, it may be sufficient to update and monitor a business entity within a single rule definition.

Hierarchical pattern modeling facilitates reuse of pattern-detection logic on the level of *pattern definitions* and *rule definitions*, which otherwise have to be modeled in separate, potentially highly complex and redundant decision graphs. Reusability is achieved through special rule components – so-called sub-pattern components – which serve as references to sub-level pattern definitions in the super-level decision graphs in which they are used. These rule components can be integrated with the other elements of SARI’s graphical event-pattern language in an accustomed workflow and can be configured to

the given application context through respective input-parameter values. Tailored evaluation strategies enable high-performance event processing as well as arbitrary nestings of pattern-detection logic. Similar to entity-based state management, hierarchical pattern modeling does well integrate with, but not immediately depend on the proposed approach to rule management. It relies, however, on the concept of pattern definitions, which are implicitly available when the presented rule-management framework is used.

Since implemented within SARI, the presented framework and its extensions have been successfully applied in use cases from business domains. Experience from these projects confirmed that a separation into infrastructural and sense-and-respond rule management is particularly useful when the high-level business logic of an application shall be administrated by end users with restricted technical skills, and/or changes frequently. Here, significant efficiency enhancements could be achieved for both the customers' IT departments – which may now focus on maintaining the low-level processing logic of an application – and the involved domain experts, which may now administrate their rules by themselves, in a straightforward and fail-safe manner. Experience also showed, however, that in comparison to less flexible solutions that are based on rule definitions only, setting up a full-fledged rule-management system requires considerable additional efforts, e.g., for identifying event-processing logic that qualifies as a “building block”, abstracting from such logic through input and output parameters, and defining and configuring rule spaces. Such efforts may not be justified in smaller installations or if all users have sufficient skills anyway. It is essential to note, though, that the proposed framework provides full support for both kinds of architectures: If a separate sense-and-respond layer is not rewarding, the complete functionality of an application can just as well be implemented based on rule definitions.

Resulting in applications that are inherently generic and adaptable to the current processing needs of an enterprise through sense-and-respond rules, existing projects eventually showed that our framework supports the definition of “standardized” CEP solutions, which can be offered to multiple customers with similar basic processing requirements and business environments. Customers can then benefit from the typical advantages of commercial off-the-shelf (COTS) software, including reduced costs, rapid deployment, well-tested and documented functionality, and standardized updates. The idea of *solution templates*, which combine a prepared sense-and-respond rule management system with so-called *visualization templates* for the ex-post analysis of a system, is not a focus of this thesis but has been elaborated in a separate publication [88].

|   | <b>Guideline</b>                  | <b>Description</b>   |
|---|-----------------------------------|--|
| 1 | <b>Design as an Artifact</b>      | Design science requires the creation of a purposeful IT artifact.  |
| 2 | <b>Problem Relevance</b>          | Design science must be relevant with respect to an important business problem.   |
| 3 | <b>Design Evaluation</b>          | Design science must evaluate the utility, quality, and efficacy of a design artifact using well-executed evaluation methods.                         |
| 4 | <b>Research Contributions</b>     | Design science must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. |
| 5 | <b>Research Rigor</b>             | Design science relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.                       |
| 6 | <b>Design as a Search Process</b> | Design science must apply a search process to reach desired ends while satisfying laws in the problem environment.                                   |
| 7 | <b>Communication of Research</b>  | Design science must be presented effectively both to technology-oriented as well as management-oriented audiences.                                   |

**Table 10.1.** Guidelines for Design-Science Research [53]

## 10.2 Evaluation Against Design Science

The presented research on user-oriented rule management is located at the intersection of computer science and the Information Systems (IS) discipline, which goal is to develop “knowledge concerning both the management of information technology and the use of information technology for managerial and organizational purposes” [143]. It is particularly oriented towards Alan Hevner’s well-known guidelines for *Design Science in IS Research* [53]; in contrast to IS’ complementary *behavioral-science paradigm* that focuses on theories about human and organizational phenomena, design science thereby seeks to design, implement, and evaluate innovative artifacts that solve organizational problem. In the remainder of this section, Hevner’s guidelines are discussed in the context of this thesis.

### Research Contributions

This thesis contributes to the field of Complex Event Processing an innovative framework for user-oriented rule management, along with its extensions towards entity-based state management and hierarchical pattern modeling. These contributions advance our understanding of how best to provide CEP functionality within practical business environments where heterogeneous user groups are concerned with the setup and maintenance of event-driven applications. Another contribution is a model-driven reference description of SARI, which is intended to serve as a basis for future research efforts and encourage communication, interchange, and cooperation within the community.

### **Problem Relevance**

In recent times, mismatches between the complexity of CEP frameworks and the technical abilities of their adopters have motivated an active discussion on how to make CEP accessible to business users. Etzion and Niblett [37] list the development from programming-centered to semi-technical development tools as one of the emerging directions in event processing. Chandy and Schulte [27] identify the ability to “enable business users to tailor systems to their needs” as a major criterion for the relevance of an event-based system, and claim that a one-size-fits-all specification of events and responses does not work. Surveys among potential and actual adopters of CEP showed that a vast majority of businesses would like to have rules defined by business specialists or business analysts [33]. We conclude that a rule-management framework that is aware of different user groups, their particular skills, responsibilities, and requirements is demanded both in the academic world and in the industry.

### **Design as an Artifact**

The research efforts presented in this thesis have led to a range of fully functional and partly commercialized improvements of the pre-existing SARI suite. Using the novel rule-management framework and its extensions, technical experts as well as business users are now provided workflows and tools that are tailored to their particular skills, competences, and requirements. These tools and workflows are applicable also in entity-centric environments which have been difficult, if not impossible, to approach with existing rule-modeling and evaluation facilities. We therefore argue that the developed framework qualifies as an “purposeful IT artifact created to address an important organizational problem” [53].

### **Design Evaluation**

The presented concepts have been evaluated for technical feasibility, applicability, and utility in practical business environments using well-recognized methods. Technical feasibility of our concepts is proofed by their successful implementation within SARI. Applicability is demonstrated using an exemplary SARI application for event-based service assurance, where SARI is applied as an extension to the UC4 Automation Engine [130]. Utility in practical business environments has been investigated in a case study, conducted over a period of seven days at a leading manufacturer of agricultural machinery.

## Research Rigor

According to Hevner et al. [53], research rigor

“is derived from the effective use of the knowledge base – theoretical foundations and research methodologies. Success is predicated on the researcher’s skilled selection of appropriate techniques to develop or construct [...] [an] artifact and the selection of appropriate means to [...] evaluate the artifact.”

The presented work draws from existing research in the areas of event-based computing, Business Rule Management (BRM), and active object-oriented databases, all of which build upon a solid and well-formalized theoretical background (e.g., [21, 51, 83, 100, 101]) and have proved effective by their widespread use in practical enterprise computing. Template-based rule creation, which underlies our approach to sense-and-respond rule management, has been studied in the context of BRM and is commonly recognized as a suitable method for business-user-oriented rule creation (e.g., [47]). The applicability and generalizability of template-based rule creation strategies is utilized in several of the market’s leading BRM systems (e.g., [38, 57]). All throughout this thesis, particular attention is paid to solid meta-model design as well as type safety (e.g., [79]) to assure logical validity across heterogeneous application scenarios. Our work is thoroughly evaluated for technical feasibility, applicability, and utility using the afore-mentioned means.

## Design as a Search Process

The presented approach to user-oriented rule management is an iterative advancement of SARI’s pre-existing rule management framework as originally presented by Schiefer et al. [113]. This approach equally builds upon the concept of decision graphs, however, is exclusively oriented towards power users of a system and imposes a strong coupling of event patterns, actions, and rule services for all event-pattern rules of an application. Customer feedback showed that these restrictions do often not correspond with the organizational framework conditions of a company. With further improvements in mind, the present framework is designed with a particular focus on extendability and implemented using commonly used and well-maintained technologies such as .NET, Java, and Google Web Toolkit (GWT) [46].

## Communication of Research

Our research efforts have resulted in an overall collection of six papers and articles, all of which were presented at international conferences or accepted for publication by international journals. These are:

- H. Obweger, J. Schiefer, M. Suntinger, and P. Kepplinger, “Model-driven rule composition for event-based systems,” *Int. J. Business Process Integration and Management*, vol. 5, no. 4, 2011.
- H. Obweger, J. Schiefer, M. Suntinger, P. Kepplinger, and S. Rozsnyai, “User-oriented rule management for event-based applications,” in *DEBS '11: Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2011, pp. 39–48.
- A. Kavelar, H. Obweger, J. Schiefer, and M. Suntinger, “Web-based decision making for Complex Event Processing systems,” in *Services '10: Proceedings of the IEEE 6th World Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 453–458.
- H. Obweger, J. Schiefer, M. Suntinger, and R. Thullner, “Entity-based state management for Complex Event Processing applications,” in *Rule-Based Reasoning, Programming, and Applications*, ser. Lecture Notes in Computer Science, N. Bassiliades, G. Governatori, and A. Paschke, Eds. Springer Berlin / Heidelberg, 2011, vol. 6826, pp. 154–169.
- H. Obweger, J. Schiefer, P. Kepplinger, and M. Suntinger, “Discovering hierarchical patterns in event-based systems,” in *Proceedings of the 2010 IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 329–336.
- H. Obweger, J. Schiefer, M. Suntinger, F. Breier, and R. Thullner, “Complex Event Processing *off the Shelf* – Rapid development of event-driven applications with solution templates,” in *MED '11: Proceedings of the 19th IEEE Mediterranean Conference on Control and Automation*, 2011.

Technical audiences are provided with a model-driven overview of SARI, complete and formalized descriptions of the proposed rule-management framework and its extensions, as well as architectural details on the presented reference implementations. Researchers are therewith enabled to further extend the proposed concepts. Also, researchers are encouraged to port the proposed concepts to other CEP engines. Managerial audiences are provided with detailed descriptions of the different user roles that are assumed to participate in the creation and maintenance of event-driven applications. Potential adopters can then decide whether or not an implementation of the proposed system is valuable in their particular organizational settings. A case study is presented to illustrate the performance of the proposed architecture in real-world business environments.

### 10.3 Open Issues and Future Work

In this thesis, a novel approach to rule management has been presented and thoroughly evaluated for technical feasibility, applicability, and utility. Despite the mainly positive results of these evaluations, several potential improvement points could be identified in the proposed rule-management facilities; these have been discussed in Section 8.9 and Section 9.6 and will be subject to further research and development over the next months. Apart from these issues, the following ideas and open issues will drive our future research:

Luckham [72] identified “ensuring logical consistency and absence of redundancies” as one of four key tasks for event-pattern rule management systems. This task is not currently addressed with the proposed approach to user-oriented rule management, neither on the level of individual decision graphs nor on the level of entire rule services or rule spaces. We believe that rule-space-level consistency checking could provide a great benefit to sense-and-respond rule management, where business users operate in parallel and a potentially mutually exclusive manner. Here, it is currently possible to trigger counteracting actions (such as *upgrading* and *blocking* a user) in response to one and the same event pattern, or to trigger an atomic action (such as incrementing the rating of a customer) in several, identical copies of a sense-and-respond rule. The definition of a semantic model, enabling application developers to mark action definitions as *counteracting*, *atomic*, or similar, is subject to future work.

Another open issue is full *versioning support* as is commonly available in business-rule management (BRM) solutions. Versioning enables users to investigate the history of a given rule instance and to revert unsuccessful changes, e.g., if a changed threshold in an event pattern results in an unacceptably high false-positive rate. A significant challenge here is in the strong dependency between a rule instance and the *application description* of a given SARI application; in particular, a rule instance can be rolled back to a certain version only if this version is consistent with respect to the given set of rule spaces, pattern definitions, and action definitions.

Future research will furthermore focus on the statistical analysis of rule instances and rule spaces during the run time of a SARI application. Currently, it is not easily possible for users of the proposed rule-management system to answer questions like: How often does a particular event-pattern rule trigger per hour or day? How did my recent changes affect the behavior of this rule? Are there periods of increased activity, or is there a long-running trend? We believe that well-edited runtime data – e.g., delivered in the form of daily reports or interactive dashboards – could help IT experts as well as business users in creating lean and effective rule sets and reacting to emerging developments in a proactive manner. Apart from tailored user interfaces, this extension requires changes in the back-end and data management layer of SARI.



In entity-based state management, future research will focus on an efficient implementation and thorough performance evaluations of business entity providers and underlying rule evaluation strategies. In the latter case, special emphasis is on an alternative, synchronous rule evaluation mode, where noteworthy states are guaranteed to be detected. Another research topic is the entity-driven analysis of historical event data; while a line-chart based visualization for scores has been implemented and incorporated into the Event Analyzer [123, 124], tailored visualization methods for base entities and sets are still to be designed.

In hierarchical pattern modeling, it is conceivable to exploit the hierarchical structure of decision graphs towards their optimized evaluation on incoming event streams, through extensive reuse of intermediate event-processing results. While comparable approaches have been discussed in the literature (e.g., [68, 69, 70]), the proposed approach is unique in its extensive use of input parameters and other configuration options, which we believe makes it extremely difficult to achieve valuable results.

### **Outlook: CEP as a Service?**

The presented approach to user-oriented rule management may be considered as a first step away from the large-scale, lengthy-to-install, and difficult-to-use IDEs that were long dominating CEP, towards tools that are tailored to the particular requirements of their users, available “any time, anywhere”, and usable without any preliminary steps. In the long run, we believe that this development is highly promising and should be pursued further – not only with respect to front-end tools, but also regarding the actual event-processing and data-management facilities of a CEP engine. Why investing time and money into setting up a CEP engine, why worrying about reliability, why messing around with software updates and bug fixes, if all this could be done somewhere else by someone else, say, in cloud computing environments? At a very final stage of this process, customers could then consume CEP “as a service”: Subscribing to CEP functionality for exactly as long as it is needed, modeling the desired event-processing logic through tailored browser applications, and integrating their business with the CEP provider using standard protocols such as SOAP, JSON, or SMTP. CEP as a service will not be available tomorrow, and it may not be available next year. But it could become an important future direction for CEP, and it could become a great chance for innovative startups to (re-)position themselves in this fast-growing market. Until then, plenty of research problems are waiting to be worked on: How best to price such service? How to protect your system and your customers from threats, e.g., through spoofed event submissions? What about availability and service-level agreements? And, last but not least: How to process the events of hundreds or thousands of customers in a performant and scalable manner?

---

## List of Figures

|      |  |    |
|------|--|----|
| 1.1  | Scope of this Thesis .....                               | 28 |
| 2.1  | Preliminary Example (cf. [105]) .....                    | 41 |
| 3.1  | Sense-and-Respond Loop .....                             | 45 |
| 3.2  | SARI Application Overview .....                          | 45 |
| 3.3  | SARI Application Model .....                             | 48 |
| 3.4  | Event Type Meta-Model .....                              | 52 |
| 3.5  | Exemplary Event Type.....                                | 53 |
| 3.6  | Correlation Set Meta-Model .....                         | 54 |
| 3.7  | Exemplary Correlation Set .....                          | 55 |
| 3.8  | Correlating Events at Run Time .....                     | 56 |
| 3.9  | Event Processing Meta-Model.....                         | 57 |
| 3.10 | Map Element Hierarchy .....                              | 58 |
| 3.11 | Exemplary Event-Processing Map .....                     | 61 |
| 3.12 | Implementation Architecture.....                         | 62 |
| 4.1  | Decision Graphs in the SARI Application Model .....      | 70 |
| 4.2  | Decision Graph Meta-Model .....                          | 73 |
| 4.3  | Event Condition Component .....                          | 76 |
| 4.4  | Event Case Component .....                               | 77 |
| 4.5  | Timer Component .....                                    | 78 |
| 4.6  | Scheduler Component .....                                | 78 |
| 4.7  | Response-Event Action Component .....                    | 80 |
| 4.8  | Exemplary Decision Graph .....                           | 80 |
| 4.9  | Handling Decision-Graph State at Run Time .....          | 82 |
| 4.10 | Exemplary Stateless Decision Graph .....                 | 83 |
| 5.1  | Rule Management in Power-User-Oriented CEP Systems ..... | 86 |
| 5.2  | Rule Management in CEP/BRE Architectures .....           | 87 |
| 5.3  | Rule Management in SARI .....                            | 88 |

|      |   |     |
|------|---|-----|
| 5.4  | Infrastructural Rules vs. Sense-and-Respond Rules.....        | 89  |
| 5.5  | Overview of Infrastructural Rule Management .....             | 99  |
| 5.6  | Rule Definition Meta-Model .....                              | 101 |
| 5.7  | Exemplary Rule Definition .....                               | 102 |
| 5.8  | Overview of Sense-and-Respond Rule Management .....           | 105 |
| 5.9  | Rule Spaces in a SARI Application .....                       | 109 |
| 5.10 | Pattern Definition Meta-Model.....                            | 111 |
| 5.11 | Signal Component .....  | 112 |
| 5.12 | Action Definition Meta-Model .....                            | 114 |
| 5.13 | Business Pattern Meta-Model.....                              | 116 |
| 5.14 | Business Action Meta-Model.....                               | 119 |
| 5.15 | Sense-and-Respond Rule Meta-Model .....                       | 120 |
| 5.16 | Exemplary Sense-and-Respond Rule .....                        | 122 |
| 5.17 | Rule Space Meta-Model .....                                   | 123 |
| 5.18 | Rule Space Meta-Model: Prepared Bindings and Templates ....   | 123 |
| 5.19 | Implementation Architecture.....                              | 127 |
| 5.20 | Sense-and-Respond Rule Schema .....                           | 129 |
| 5.21 | Pattern Definition Editor .....                               | 133 |
| 5.22 | Action Definition Editor.....                                 | 133 |
| 5.23 | Pattern Page of the Rule Space Editor .....                   | 135 |
| 5.24 | Action Page of the Rule Space Editor .....                    | 135 |
| 5.25 | Templates Page of the Rule Space Editor .....                 | 136 |
| 5.26 | Template Creation Wizard .....                                | 136 |
| 5.27 | Web Client Integration.....                                   | 137 |
| 5.28 | Web Client Overview .....                                     | 138 |
| 5.29 | Dropped-Down Rule Filtering Control .....                     | 140 |
| 5.30 | Rule Space Selection .....                                    | 141 |
| 5.31 | Pattern Selection .....                                       | 142 |
| 5.32 | Rule Assembling .....   | 142 |
| 5.33 | Input Parameter Dialog .....                                  | 143 |
| 5.34 | Defining Input-Parameter Values in Concatenation Mode.....    | 144 |
| 5.35 | Rule Finalization .....                                       | 145 |
| 5.36 | Template Selection .....                                      | 146 |
| 5.37 | Template Instantiation .....                                  | 147 |
| 5.38 | Repairing Inconsistent Rules .....                            | 148 |
| 6.1  | A High-Level View on Business Entity Providers .....          | 152 |
| 6.2  | Infrastructural Rules vs. Sense-and-Respond Rules.....        | 153 |
| 6.3  | SARI Application Model, including the Business Entity Model . | 155 |
| 6.4  | Business Entity Provider Meta-Model .....                     | 159 |
| 6.5  | Score Type Meta-Model.....                                    | 161 |
| 6.6  | Exemplary Score Type .....                                    | 163 |
| 6.7  | Base Entity Type Meta-Model .....                             | 164 |
| 6.8  | Exemplary Base Entity Type .....                              | 164 |
| 6.9  | Queue Type Meta-Model .....                                   | 166 |

|      |  |     |
|------|--|-----|
| 6.10 | Exemplary Set Type .....   | 167 |
| 6.11 | Generalized Correlation Set Meta-Model .....                         | 168 |
| 6.12 | Exemplary Extended Correlation Set .....                             | 169 |
| 6.13 | Business-Entity Action Meta-Model .....                              | 170 |
| 6.14 | Business Entity Action Component .....                               | 172 |
| 6.15 | Business-Entity Condition Meta-Model .....                           | 173 |
| 6.16 | Business Entity Condition Component .....                            | 175 |
| 6.17 | Implementation Architecture .....                                    | 176 |
| 6.18 | Continuous Entity Monitoring .....                                   | 180 |
| 6.19 | Example Overview .....   | 181 |
| 6.20 | Exemplary Updating Rule .....  | 182 |
| 6.21 | Exemplary Pattern Definition for Continuous Entity Monitoring .....  | 183 |
| 6.22 | Exemplary Pattern Definition for On-Demand Entity Monitoring .....   | 184 |
|      |  |     |
| 7.1  | Hierarchical Event-Pattern Detection .....                           | 189 |
| 7.2  | SARI Application Model with Sub-Pattern Relationships .....          | 189 |
| 7.3  | Pattern Definition Meta-Model .....                                  | 193 |
| 7.4  | Pattern Detection with Sub-Pattern Components .....                  | 195 |
| 7.5  | Sub-Pattern Component Meta-Model .....                               | 195 |
| 7.6  | Sub-Pattern Component .....  | 197 |
| 7.7  | Applying <i>Expand</i> to an Exemplary Decision Graph (1 of 2) ..... | 200 |
| 7.8  | Applying <i>Expand</i> to an Exemplary Decision Graph (2 of 2) ..... | 200 |
| 7.9  | Rule Definition “Send Email on Suspicious User Behavior” .....       | 207 |
|      |  |     |
| 8.1  | System Overview .....  | 211 |
| 8.2  | Event Processing Model .....   | 214 |
| 8.3  | Rule Definition “Filter duplicate agent messages” .....              | 217 |
|      |  |     |
| 9.1  | Event Processing Model .....   | 227 |
| 9.2  | User Roles and Implementations .....                                 | 230 |



---

## List of Tables

|     |   |     |
|-----|---|-----|
| 1.1 | Guidelines for Design-Science Research [53]         | 30  |
| 2.1 | Event Types   | 41  |
| 3.1 | Stages of the Sense-and-Respond Loop [114, 131]     | 47  |
| 5.1 | Exemplary Pattern Definition                        | 113 |
| 5.2 | Exemplary Action Definition                         | 115 |
| 5.3 | Exemplary Business Pattern                          | 118 |
| 5.4 | Exemplary Business Action                           | 119 |
| 5.5 | Sources of Sense-and-Respond Rule Inconsistency     | 130 |
| 5.6 | Sources of Sense-and-Respond Rule Inconsistency     | 147 |
| 6.1 | Update Functions for Scores                         | 162 |
| 6.2 | Query Properties for Scores                         | 163 |
| 6.3 | Update Functions for Base Entities                  | 164 |
| 6.4 | Update Functions for Sets                           | 166 |
| 6.5 | Querying Properties for Sets                        | 166 |
| 7.1 | Pattern Definition “Short-Term Transaction”         | 205 |
| 7.2 | Pattern Definition “Balance Above Threshold”        | 206 |
| 8.1 | Event Types   | 212 |
| 8.2 | Infrastructural Rules                               | 216 |
| 8.3 | Rule Spaces   | 218 |
| 8.4 | Pattern Definitions                                 | 219 |
| 8.5 | Pattern Definitions (continued)                     | 220 |
| 8.6 | Pattern Definition “Task runtime exceeded per type” | 220 |
| 8.7 | Action Definitions                                  | 222 |
| 9.1 | Pattern Definitions                                 | 229 |

248 List of Tables

10.1 Guidelines for Design-Science Research [53] ..... 237

---

## Listings

|  |     |
|--|-----|
| 5.1 Exemplary Sense-and-Respond Rule Description . . . . . | 129 |
|--|-----|





---

## List of Algorithms

|     |                             |     |
|-----|-----------------------------|-----|
| 7.1 | Expand( $d, p, sub$ ) ..... | 199 |
|-----|-----------------------------|-----|



---

## References

- [1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, “The design of the Borealis stream processing engine,” in *CIDR '05: Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.
- [2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, August 2003.
- [3] A. Adi, D. Botzer, G. Nechushtai, and G. Sharon, “Complex Event Processing for financial services,” in *Proceedings of the IEEE Services Computing Workshops*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 7–12.
- [4] A. Adi and O. Etzion, “Amit – The situation manager,” *The VLDB Journal*, vol. 13, no. 2, pp. 177–203, May 2004.
- [5] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.
- [6] J. F. Allen, “Maintaining knowledge about temporal intervals,” *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, November 1983.
- [7] R. Altman, W. R. Schulte, M. Pezzini, and D. Sholler, “Predicts 2012: Cloud computing and event processing will be the key advances in application architecture,” Gartner Research, Stamford, CT, USA, Tech. Rep., 2011.

- [8] R. v. Ammon, C. Silberbauer, and C. Wolff, “Domain specific reference models for event patterns – for faster developing of business activity monitoring applications,” in *Proceedings of the VIP Symposia on Internet related research with elements of M+I+T++*, 2007.
- [9] Anon., Internal Customer Report, 2011, *details upon request*.
- [10] Apache Software Foundation, “Apache Tomcat 7,” Software, 2010.
- [11] M. Bali, *Drools JBoss Rules 5.0 Developer’s Guide*. Packt Publishing, 2009.
- [12] A. Barros, G. Decker, and A. Grosskopf, “Complex events in business processes,” in *Business Information Systems*, ser. Lecture Notes in Computer Science, W. Abramowicz, Ed. Springer Berlin / Heidelberg, 2007, vol. 4439, pp. 29–40.
- [13] M. Bichler, “Review of design science in information systems research, by a. hevner et al.” *Wirtschaftsinformatik*, vol. 48, no. 2, pp. 133–142, 2006.
- [14] C. Brett and M. Gualtieri, “Must you choose between business rules and Complex Event Processing platforms?” Forrester Research, Cambridge, MA, USA, Tech. Rep., 2009.
- [15] R. Bruns and J. Dunkel, *Event-Driven Architecture. Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Berlin Heidelberg: Springer-Verlag, 2010.
- [16] F. Bry and M. Eckert, “Rule-based composite event queries: The language XChange<sup>EQ</sup> and its semantics,” in *RR ’07: Proceedings of the 1st international conference on Web reasoning and rule systems*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 16–30.
- [17] —, “Rules for making sense of events: Design issues for high-level event query and reasoning languages,” in *Proceedings of the AAAI Spring Symposium: AI Meets Business Rules and Process Management*. Menlo Park, CA, USA: AAAI Press, 2008, pp. 12–16.
- [18] F. Bry, M. Eckert, and P.-I. Patranjan, “Reactivity on the web: Paradigms and applications of the language XChange,” *Journal of Web Engineering*, vol. 5, no. 1, pp. 3–24, 2006.
- [19] B. Burton, Y. Genovese, N. Rayner, R. Casonato, M. Smith, M. A. Beyer, T. Austin, B. Gassman, and D. Sommer, “Predicts 2011: Pattern-based strategy technologies and business practices gain momentum,” Gartner Research, Stamford, CT, USA, Tech. Rep., 2010.

- [20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture: A system of patterns*. New York, NY, USA: John Wiley & Sons, Inc., 1996.
- [21] Business Rules Group, “Business rules manifesto Version 2.0,” Business Rules Group, Tech. Rep., 2004.
- [22] R. Casati and A. C. Varzi, “50 years of events – An annotated bibliography 1947 to 1997,” Philosophy Documentation Center, Tech. Rep., 1997.
- [23] —, “Events,” in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., 2010. [Online]. Available: <http://plato.stanford.edu/archives/spr2010/entries/events>
- [24] A. Castro Alves, “New event-processing design patterns using CEP,” in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, S. Rinderle-Ma, S. Sadiq, and F. Leymann, Eds. Springer Berlin Heidelberg, 2010, vol. 43, pp. 359–368.
- [25] S. Chakravarthy and D. Mishra, “Snoop: An expressive event specification language for active databases,” *Data Knowl. Eng.*, vol. 14, no. 1, pp. 1–26, 1994.
- [26] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 2009.
- [27] K. M. Chandy and W. R. Schulte, *Event-Processing: Designing IT Systems for Agile Companies*. McGraw-Hill Professional, 2009.
- [28] D. Cohn and R. Hull, “Business artifacts: A data-centric approach to modeling business operations and processes,” *IEEE Data Engineering Bulletin*, vol. 32, no. 3, pp. 3–9, 2009.
- [29] D. G. Conway and G. J. Koehler, “Interface agents: Caveat mercator in electronic commerce,” *Decis. Support Syst.*, vol. 27, no. 4, pp. 355–366, 2000.
- [30] Coral8, “Complex Event Processing: Ten design patterns,” Coral8, Inc., Tech. Rep., 2006.

- [31] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, “Towards expressive publish/subscribe systems,” in *Advances in Database Technology - EDBT 2006*, ser. Lecture Notes in Computer Science, Y. Ioannidis, M. Scholl, J. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, and C. Boehm, Eds. Springer Berlin / Heidelberg, 2006, vol. 3896, pp. 627–644.
- [32] A. v. Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, 2000.
- [33] ebizQ, “Event processing market pulse,” ebizQ, Tech. Rep., 2007.
- [34] M. Eckert and F. Bry, “Aktuelles Schlagwort: Complex Event Processing (CEP),” *Informatik Spektrum*, vol. 32, no. 2, pp. 163–167, 2009.
- [35] Elmo, Gum, Heather, Holly, Mistletoe, and Rowan, *Notes Towards the Complete Works of Shakespeare*. Kahve-Society & Liquid Press, 2002.
- [36] EsperTech, “Esper,” Software.
- [37] O. Etzion and P. Niblett, *Event Processing in Action*. Stamford, CT, USA: Manning Publications Co., 2010.
- [38] Fair Isaac Corporation, “Blaze Advisor,” Software.
- [39] T. Faison, *Event-Based Programming: Taking Events to the Limit*. Berkely, CA, USA: Apress, 2006.
- [40] L. Fiege, “Visibility in event-based systems,” Ph.D. dissertation, Technische Universität Darmstadt, 2006.
- [41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [42] J.-P. Garbani and G. O’Donnell, “Market overview: IT process automation, Q3 2011,” Forrester Research, Cambridge, MA, USA, Tech. Rep., 2011.
- [43] B. Gassman, G. Herschel, A. Bitterer, J. Richardson, and N. Chandler, “Cool vendors in business intelligence and performance management,” Gartner Research, Stamford, CT, USA, Tech. Rep., 2008.
- [44] G. Gatzui and K. Dittrich, “SAMOS: An active object-oriented database system,” *IEEE Data Eng. Bulletin*, vol. 15, no. 4, pp. 23–26, 1992.
- [45] T. Gockel, *Form der wissenschaftlichen Ausarbeitung: Studienarbeit, Diplomarbeit, Dissertation, Konferenzbeitrag*. Springer-Verlag GmbH, 2008.

- [46] Google, “Google Web Toolkit 2.1.0,” Software, 2011.
- [47] I. Graham, *Business Rules Management and Service Oriented Architecture: A Pattern Language*. New York, NY, USA: John Wiley & Sons, Inc., 2007.
- [48] M. Gualtieri and J. R. Rymer, “The Forrester Wave: Complex Event Processing (CEP) platforms, Q3 2009,” Forrester Research, Cambridge, MA, USA, Tech. Rep., 2009.
- [49] S. H. Haeckel, *Adaptive Enterprise: Creating and Leading Sense-And-Respond Organizations*. Cambridge, MA, USA: Harvard Business Press, 1999.
- [50] Haley, “Haley Expert Rules,” Software, 2007.
- [51] B. v. Halle, *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. New York, NY, USA: John Wiley & Sons, Inc., 2001.
- [52] E. Hemingway, *Preface to “The Great Crusade”, by Gustav Regler*. Longmans, Green and Co., 1940.
- [53] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in Information Systems research,” *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.
- [54] A. Hinze, K. Sachs, and A. Buchmann, “Event-based applications and enabling technologies,” in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2009, pp. 1–15.
- [55] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [56] IBM, “WebSphere Business Events,” Software.
- [57] —, “IBM WebSphere ILOG JRules,” Software, 2010.
- [58] JBoss, “Drools 5.1,” Software, 2010. [Online]. Available: <http://www.jboss.org/drools>
- [59] —, “Drools Fusion,” Software, 2010. [Online]. Available: <http://www.jboss.org/drools/drools-fusion.html>



- [60] D. Jeffery, A. Kozlenkov, and A. Paschke, “State management and concurrency in event processing,” in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2009, pp. 23:1–23:4.
- [61] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, “A glossary of temporal database concepts,” *SIGMOD Rec.*, vol. 21, pp. 35–43, September 1992.
- [62] A. Kavelar, H. Obweger, J. Schiefer, and M. Suntinger, “Web-based decision making for Complex Event Processing systems,” in *Services '10: Proceedings of the IEEE 6th World Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 453–458.
- [63] I. Kellner and L. Fiege, “Viewpoints in complex event processing: industrial experience report,” in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2009.
- [64] R. A. Kowalksi and M. J. Sergot, “A logic-based calculus of events,” *New Generation Computing*, vol. 4, no. 1, pp. 67–95, 1986.
- [65] S. Kumaran, P. Nandi, T. Heath, K. Bhaskaran, and R. Das, “ADoc-oriented programming,” in *SAINT '03: Proceedings of the 2003 Symposium on Applications and the Internet*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 334–341.
- [66] K. C. Laudon and C. G. Traver, *Management Information Systems*, 12nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011.
- [67] P. Limbeck, M. Suntinger, and J. Schiefer, “SARI OpenRec – Empowering recommendation systems with business events,” in *DBKDA '10: Proceedings of the 2010 Second International Conference on Advances in Databases, Knowledge, and Data Applications*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 111–119.
- [68] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta, “E-Cube: Multi-dimensional event sequence processing using concept and pattern hierarchies,” Worcester Polytechnic Institute, Tech. Rep., 2009.
- [69] —, “E-Cube: Multi-dimensional event sequence processing using concept and pattern hierarchies,” in *ICDE '10: Proceedings of the 26th IEEE International Conference on Data Engineering*, 2010, pp. 1097–1100.

- [70] M. Liu and E. A. Rundensteiner, “Event sequence processing: new models and optimization techniques,” in *IDAR '10: Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research*. New York, NY, USA: ACM, 2010, pp. 7–12.
- [71] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
- [72] —, “What’s the difference between ESP and CEP?” 2006. [Online]. Available: <http://complexevents.com/?p=103>
- [73] —, “A short history of Complex Event Processing – part 1: Beginnings,” 2007. [Online]. Available: <http://complexevents.com/?p=321>
- [74] —, “A short history of Complex Event Processing – part 2: The rise of CEP,” 2007. [Online]. Available: <http://www.complexevents.com/2008/08/28/a-short-history-of-complex-event-processing-part-2-the-rise-of-cep>
- [75] J. Makkonen, H. Ahonen-Myka, and M. Salmenkivi, “Applying semantic classes in event detection and tracking,” in *ICON '02: Proceedings of International Conference on Natural Language Processing*, R. Sangal and S. M. Bendre, Eds., Mumbai, India, 2002, pp. 175–183.
- [76] P. Mangkorntong and F. A. Rabhi, “A high-level approach for defining & composing event patterns and its application to e-markets,” in *EDA-PS '07: Proceedings of The Second International Workshop on Event-driven Architecture, Processing and Systems*, 2007.
- [77] C. McGregor and J. Schiefer, “Correlating events for monitoring business processes,” in *Proceedings of the 6th International Conference on Enterprise Information Systems*, 2004, pp. 320–327.
- [78] Microsoft, “StreamInsight,” Software, 2010.
- [79] R. Milner, “A theory of type polymorphism in programming,” *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.
- [80] C. H. Mooney and J. F. Roddick, “Mining relationships between interacting episodes,” in *SDM '04: Proceedings of the 4th SIAM International Conference on Data Mining*, 2004, pp. 1–10.
- [81] T. Morgan, *Business Rules and Information Systems: Aligning IT with Business Goals*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

- [82] T. Moser, H. Roth, S. Rozsnyai, R. Mordinyi, and S. Biffl, “Semantic event correlation using ontologies,” in *OTM '09: Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1087–1094.
- [83] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- [84] E. Mui and K. Kwong, “Event Insight: SAP’s first Complex Event Processing engine for the business user,” in *SAP TechEd 2010*, 2010.
- [85] P. Nandi, D. König, S. Moser, R. Hull, V. Klicnik, S. Claussen, M. Kloppmann, and J. Vergo, “Data4BPM, Part 1: Introducing business entities and the Business Entity Definition Language (BEDL),” IBM Corp., Riverton, NJ, USA, Tech. Rep., 2010.
- [86] A. Nigam and N. S. Caswell, “Business artifacts: An approach to operational specification,” *IBM Syst. J.*, vol. 42, pp. 428–445, July 2003.
- [87] H. Obweger, J. Schiefer, P. Kepplinger, and M. Suntinger, “Discovering hierarchical patterns in event-based systems,” in *Proceedings of the 2010 IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 329–336.
- [88] H. Obweger, J. Schiefer, M. Suntinger, F. Breier, and R. Thullner, “Complex Event Processing *off the Shelf* – Rapid development of event-driven applications with solution templates,” in *MED '11: Proceedings of the 19th IEEE Mediterrean Conference on Control and Automation*, 2011.
- [89] H. Obweger, J. Schiefer, M. Suntinger, and P. Kepplinger, “Model-driven rule composition for event-based systems,” *Int. J. Business Process Integration and Management*, vol. 5, no. 4, 2011.
- [90] H. Obweger, J. Schiefer, M. Suntinger, P. Kepplinger, and S. Rozsnyai, “User-oriented rule management for event-based applications,” in *DEBS '11: Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2011, pp. 39–48.
- [91] H. Obweger, J. Schiefer, M. Suntinger, and R. Thullner, “Entity-based state management for Complex Event Processing applications,” in *Rule-Based Reasoning, Programming, and Applications*, ser. Lecture Notes in Computer Science, N. Bassiliades, G. Governatori, and A. Paschke, Eds. Springer Berlin / Heidelberg, 2011, vol. 6826, pp. 154–169.

- [92] H. Obweger, M. Suntinger, J. Schiefer, and G. Raidl, “Similarity searching in sequences of complex events,” in *RCIS '10: Proceedings of the 2010 Fourth International Conference on Research Challenges in Information Science*, 2010, pp. 631–641.
- [93] Oracle, “Open ESB IEP SE,” Software, 2008.
- [94] —, “Oracle Complex Event Processing,” Software, 2011.
- [95] Organization for the Advancement of Structured Information Standards (OASIS), “OASIS Web Services Business Process Execution Language 2.0,” 2007. [Online]. Available: <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html/>
- [96] A. Paschke, “ECA-RuleML/ECA-LP: A homogenous Event-Condition-Action logic programming language,” in *RuleML '06: Proceedings of the International Conference on Rules and Rule Markup Languages for the Semantic Web*, 2006.
- [97] —, “Design patterns for Complex Event Processing,” in *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, 2008.
- [98] A. Paschke and A. Kozlenkov, “Rule-based event processing and reaction rules,” in *Rule Interchange and Applications*, ser. Lecture Notes in Computer Science, G. Governatori, J. Hall, and A. Paschke, Eds. Springer Berlin / Heidelberg, 2009, vol. 5858, pp. 53–66.
- [99] A. Paschke and P. Vincent, “A reference architecture for event processing,” in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2009, pp. 1–4.
- [100] N. W. Paton and O. Díaz, “Active database systems,” *ACM Comput. Surv.*, vol. 31, pp. 63–103, March 1999.
- [101] L. Perrochon, E. Jang, and D. C. Luckham, “Enlisting event patterns for cyber battlefield awareness,” in *DARPA Information Survivability Conference and Exposition*. Los Alamitos, CA, USA: IEEE Computer Society, 2000.
- [102] R. G. Ross, *Principles of the Business Rule Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [103] H. Roth, “Event data warehousing,” Ph.D. dissertation, Vienna University of Technology, 2012 (to appear).

- [104] H. Roth, J. Schiefer, H. Obwegger, and S. Rozsnyai, “Event data warehousing for Complex Event Processing,” in *RCIS '10: Proceedings of the Fourth International Conference on Research Challenges in Information Science*, 2010, pp. 203–212.
- [105] S. Rozsnyai, “Managing event streams for querying complex events,” Ph.D. dissertation, Vienna University of Technology, 2008.
- [106] S. Rozsnyai, H. Obwegger, and J. Schiefer, “Event Access Expressions: A business user language for analyzing event streams,” in *AINA '11: Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications*, 2011, pp. 191–199.
- [107] S. Rozsnyai, J. Schiefer, and H. Roth, “SARI-SQL: Event query language for event analysis,” in *Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 24–32.
- [108] S. Rozsnyai, J. Schiefer, and A. Schatten, “Concepts and models for typing events for event-based systems,” in *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2007, pp. 62–70.
- [109] —, “Solution architecture for detecting and preventing fraud in real time,” in *ICDIM '07: Proceedings of the 2nd International Conference on Digital Information Management*, 2007, pp. 152–158.
- [110] R. Sandhu, R. Boppana, R. Krishnan, J. Reich, T. Wolff, and J. Zachry, “Towards a discipline of mission-aware cloud computing,” in *CCSW '10: Proceedings of the 2010 ACM workshop on Cloud computing security*. New York, NY, USA: ACM, 2010, pp. 13–18.
- [111] G. Saurer, J. Schiefer, and A. Schatten, “Testing complex business process solutions,” in *Proceedings of the First International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 593–600.
- [112] J. Schiefer, H. Obwegger, and M. Suntinger, “Correlating business events for event-triggered rules,” in *Rule Interchange and Applications*, ser. Lecture Notes in Computer Science, G. Governatori, J. Hall, and A. Paschke, Eds. Springer Berlin / Heidelberg, 2009, vol. 5858, pp. 67–81.
- [113] J. Schiefer, S. Rozsnyai, C. Rauscher, and G. Saurer, “Event-driven rules for sensing and responding to business situations,” in *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2007, pp. 198–205.

- [114] J. Schiefer and A. Seufert, “Management and controlling of time-sensitive business processes with sense & respond,” in *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-1*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 77–82.
- [115] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel, “Distributed heterogeneous event processing: Enhancing scalability and interoperability of CEP in an industrial context,” in *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2010, pp. 150–159.
- [116] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, NY, USA: John Wiley & Sons, Inc., 2000.
- [117] M. Seiriö and M. Berndtsson, “Design and implementation of an ECA rule markup language,” in *Rules and Rule Markup Languages for the Semantic Web*, ser. Lecture Notes in Computer Science, A. Adi, S. Stoutenburg, and S. Tabet, Eds. Springer Berlin / Heidelberg, 2005, vol. 3791, pp. 98–112.
- [118] S. Sen and N. Stojanovic, “GRUVE: A methodology for Complex Event Pattern life cycle management,” in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, B. Pernici, Ed. Springer Berlin / Heidelberg, 2010, vol. 6051, pp. 209–223.
- [119] M. Shanahan, “The Event Calculus explained,” in *Artificial intelligence today*, M. J. Wooldridge and M. Veloso, Eds. Springer Berlin / Heidelberg, 1999, pp. 409–430.
- [120] G. Sharon and O. Etzion, “Event-processing network model and implementation,” *IBM Systems Journal*, vol. 47, no. 2, pp. 321–334, 2008.
- [121] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA, USA: MIT Press, 1996.
- [122] StreamBase, Inc., *StreamBase 7.0.3 Documentation*, 2011.
- [123] M. Suntinger, H. Obwegger, J. Schiefer, and M. E. Gröller, “Event tunnel: Exploring event-driven business processes,” *IEEE Comput. Graph. Appl.*, vol. 28, pp. 46–55, September 2008.
- [124] M. Suntinger, H. Obwegger, J. Schiefer, and M. E. Gröller, “The event tunnel: Interactive visualization of complex event streams for business process pattern analysis,” in *IEEE Pacific Visualization Symposium 2008*, 2008.

- [125] M. Suntinger, J. Schiefer, H. Roth, and H. Obweiger, “Data warehousing versus Event-driven BI: Data management and knowledge discovery in fraud analysis,” in *Proceedings of the International Conference on Software, Knowledge, Information Management and Applications*, 2008, pp. 129–134.
- [126] Sybase, Inc., “Beyond relational operators: Programming with FlexStreams in the Sybase Aleri Streaming Platform,” Sybase, Inc., Dublin, CA, USA, Tech. Rep., 2010.
- [127] Sybase Inc., “Sybase Aleri Streaming Platform,” Software, 2011.
- [128] TIBCO, “BusinessEvents,” Software, 2008.
- [129] Y. Turchin, A. Gal, and S. Wasserkrug, “Tuning complex event processing rules using the prediction-correction paradigm,” in *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2009, pp. 1–12.
- [130] UC4, “UC4 Automation Engine V9.00,” Software, 2011.
- [131] UC4 Senactive, *UC4 Decision*, 2011.
- [132] —, “UC4 Decision 9.0,” Software, 2011.
- [133] R. Vecera, “Efficient indexing, search and analysis of event streams,” Master’s thesis, Vienna University of Technology, 2007.
- [134] K. Vidačković, I. Kellner, and J. Donald, “Business-oriented development methodology for Complex Event Processing: Demonstration of an integrated approach for process monitoring,” in *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2010, pp. 111–112.
- [135] K. Vidačković, T. Renner, and S. Rex, “Marktübersicht Real-Time Monitoring Software,” Fraunhofer IAO, Tech. Rep., 2010.
- [136] P. Vincent, “Fair Isaac Blaze Advisor Structured Rules Language - a commercial rules representation,” in *W3C Workshop on Rule Languages for Interoperability*, 2005.
- [137] S. White, A. Alves, and D. Rorke, “WebLogic event server: A lightweight, modular application server for event processing,” in *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2008, pp. 193–200.

- [138] A. Widder, R. v. Ammon, P. Schaeffer, and C. Wolff, "Combining discriminant analysis and neural networks for fraud detection on the base of Complex Event Processing," in *DEBS' 08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, 2008.
- [139] World Wide Web Consortium (W3C), "RDF Vocabulary Description Language 1.0: RDF Schema," 2004. [Online]. Available: <http://www.w3.org/TR/rdf-schema/>
- [140] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 407–418.
- [141] S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The Aurora and Medusa projects," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 3–10, 2003.
- [142] D. Zimmer and R. Unland, "On the semantics of complex events in active database management systems," in *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 392–399. [Online]. Available: <http://dl.acm.org/citation.cfm?id=846218.847253>
- [143] R. Zmud, "Editor's comments," *MIS Quarterly*, vol. 21, no. 2, pp. xxi–xxii, 1997.





---

## Acknowledgments

I would like to thank my current and former teammates and colleagues at UC4 Senactive GmbH, in particular Josef Schiefer, Christian Plaichner, Martin Suntinger, Albert Kavelar, Peter Kepplinger, Heinz Roth, Philip Limbeck, Robert Thullner, Florian Breier, Christoph Bonitz, Lukas Maczejka, Martin Sturm, Manuel Messerer, Dorel Coban, Suzanne Martini, and Szabolcs Rozsnyai. Without you guys, this would not have been possible. Many thanks also to Christian Huemer for his support through this process, knowing exactly when to contribute his knowledge and insights.

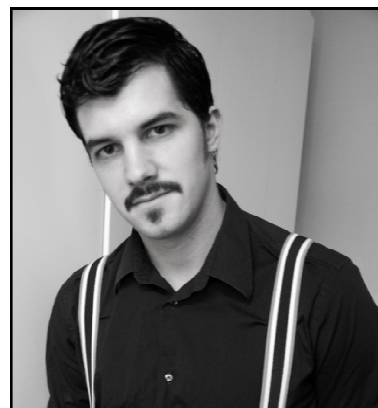
Hannes Obweger



# Curriculum Vitae

**Hannes Obweger**

Staudingergasse 13/24  
1200 Vienna, Austria  
+43 650 6293437  
ho@obweger.org



## Personal Data

---

|                       |                             |
|-----------------------|-----------------------------|
| <b>Day of birth</b>   | September 05, 1985          |
| <b>Place of birth</b> | St. Veit a.d. Glan, Austria |
| <b>Nationality</b>    | Austrian                    |

## Education

---

|              |   |
|--------------|---|
| 2009 – today | <b>Doctoral Programme in Computer Sciences</b><br><b>Vienna University of Technology</b>  |
| 2007 – 2009  | <b>Master's Programme in Software Engineering and Internet Computing (Dipl.-Ing.)</b><br><b>Vienna University of Technology</b><br><i>Pass with distinction</i><br><br><b>Master Thesis:</b> Similarity Searching in Complex Business Events and Sequences thereof<br><i>Awarded the INITS-Award 2009</i> |
| 2004 – 2009  | <b>Bachelor's Programme in Journalism and Communication Studies (Bakk.phil.)</b><br><b>University of Vienna</b><br><i>Pass with distinction</i>   |
| 2004 – 2007  | <b>Bachelor's Programme in Media and Computer Science (Bakk.techn.)</b><br><b>Vienna University of Technology</b><br><i>Pass with distinction</i>   |
| 1999 – 2004  | <b>College of Electronics specializing in Technical Computer Science (Matura)</b><br><b>Höhere Technische Bundeslehranstalt Mössingerstraße, Klagenfurt</b><br><i>Pass with distinction</i>   |
| 1995 – 1999  | <b>Gymnasium</b><br><b>Bundesrealgymnasium Spittal/Drau</b>   |

## Work Experience

---

|                           |  |
|---------------------------|--|
| 2006 – today              | <b>UC4 Senactive GmbH (formerly: Senactive IT Dienstleistungs GmbH), Vienna, Austria</b><br>Software development and research in the fields of Complex Event Processing and workload automation. |
| 2007<br><i>Internship</i> | <b>SEZ AG</b><br>Development and maintenance of the company's Sharepoint-based Intranet application.   |

|                              |  |
|------------------------------|--|
| 2006<br><i>Internship</i>    | <b>SEZ AG</b><br>Development and maintenance of the company's Sharepoint-based Intranet application.   |
| 2005<br><i>Project-based</i> | <b>ALRO Control Systems AG (Switzerland)</b><br>Design, implementation and rollout of a Java-based document administration tool.<br>Development of an interactive atlas of Swiss water plants. |
| 2004<br><i>Project-based</i> | <b>ALRO Control Systems AG (Switzerland)</b><br>Design, implementation and rollout of a Java-based document administration tool.   |
| 2003<br><i>Internship</i>    | <b>KELAG</b><br>Internship in electrical engineering   |
| 2002<br><i>Internship</i>    | <b>KELAG</b><br>Internship in electrical engineering   |
| 2001<br><i>Internship</i>    | <b>KELAG</b><br>Internship in electrical engineering   |

### Additional Qualifications

---

|   |   |
|---|---|
| <b>Business English Certificate (BEC) Vantage</b> | The BEC Vantage is an intermediate-level Cambridge ESOL exam, at Level B2 of the Council of Europe's Common European Framework of Reference for Languages. <sup>1</sup> |
|---|---|

### Language Skills

---

|                |               |
|----------------|---------------|
| <b>German</b>  | Mother tongue |
| <b>English</b> | Fluent        |

### Personal Interests

---

Traveling, motorbiking, blues music

---

<sup>1</sup> <http://www.candidates.cambridgeesol.org>

# List of Publications

## Journal Articles

---

- 2011      **Model-Driven Rule Composition for Event-Based Systems**  
*International Journal for Business Processing Integration and Management (IJBPIIM)*  
*Volume 5, Number 4*  
with Josef Schiefer, Martin Suntinger, and Peter Kepplinger
- 2008      **Event Tunnel: Exploring Event-Driven Business Processes**  
*Computer Graphics and Applications*  
*Volume 28, Number 5*  
with Martin Suntinger, Josef Schiefer, and M. Eduard Gröller

## Conference and Workshop Papers

---

- 2011      **Complex Event Processing "off the Shelf" – Rapid Development of Event-Driven Applications with Solution Templates**  
*19<sup>th</sup> Mediterranean Conference on Control and Automation*  
with Josef Schiefer, Martin Suntinger, Florian Breier, and Robert Thullner
- 2011      **Entity-Driven State Management for Complex Event Processing Applications**  
*5<sup>th</sup> International Conference on Rule-based Reasoning, Programming, and Applications*  
with Josef Schiefer, Martin Suntinger, and Robert Thullner
- 2011      **User-Oriented Rule Management for Event-Based Applications**  
*5<sup>th</sup> ACM International Conference on Distributed Event-Based Systems*  
with Josef Schiefer, Martin Suntinger, Peter Kepplinger, and Szabolcs Rozsnyai
- 2011      **Proactive Business Process Compliance Monitoring with Event-Based Systems**  
*6<sup>th</sup> International Workshop on Vocabularies, Ontologies and Rules for The Enterprise*  
with Robert Thullner, Szabolcs Rozsnyai, Josef Schiefer, and Martin Suntinger
- 2011      **Event Access Expressions - A Business User Language for Analyzing Event Streams**  
*25<sup>th</sup> IEEE International Conference on Advanced Information Networking and Applications*  
with Szabolcs Rozsnyai and Josef Schiefer
- 2010      **Web-Based Decision Making for Complex Event Processing Systems**  
*6<sup>th</sup> World Congress on Services*  
with Albert Kavelar, Josef Schiefer, and Martin Suntinger
- 2010      **Discovering Hierarchical Patterns in Event-Based Systems**  
*2010 IEEE International Conference on Services Computing*  
with Josef Schiefer, Peter Kepplinger, and Martin Suntinger
- 2010      **Event Data Warehousing for Complex Event Processing**  
*4<sup>th</sup> International Conference on Research Challenges in Information Science*  
with Heinz Roth, Josef Schiefer, and Szabolcs Rozsnyai
- 2010      **Similarity Searching in Sequences of Complex Events**  
*4<sup>th</sup> International Conference on Research Challenges in Information Science*  
with Martin Suntinger, Josef Schiefer, and Günther Raidl
- 2010      **Trend-Based Similarity Search in Time-Series Data**  
*2<sup>nd</sup> International Conference on Advances in Databases, Knowledge and Data Applications*  
with Martin Suntinger, Josef Schiefer, and Günther Raidl

- 2009 | **Correlating Business Events for Event-Triggered Rules**  
*International Symposium on Rule Interchange and Applications (RuleML'09)*  
with Josef Schiefer and Martin Suntinger
- 2008 | **Data Warehousing versus Event-Driven BI:  
Data Management and Knowledge Discovery in Fraud Analysis**  
*International Conference on Software, Knowledge, Information Management and Applications*  
with Martin Suntinger, Josef Schiefer, and Heinz Roth
- 2008 | **The Event Tunnel: Interactive Visualization of Complex Event Streams for Business  
Process Pattern Analysis**  
*IEEE Pacific Visualization Symposium*  
with Martin Suntinger, Josef Schiefer, and M. Eduard Gröller

## Patents

---

- 2009 | **Method Of Visualizing Sets Of Correlated Events On A Display**  
with Josef Schiefer and Martin Suntinger
- 2009 | **Method Of Detecting A Reference Sequence Of Events In A Sample Sequence Of Events**  
with Josef Schiefer, Martin Suntinger, and Christian Rauscher