An Interface-based Modular Approach for Designing Distributed Event-based Systems

by

Jun Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A Distributed Event-based System (DEBS) exhibits its desired behavior through its functional components collaborating with each other *via* event exchanging. Due to loose-coupling and flexibility, DEBS applications have become increasingly popular. Indeed, such systems are expected to appear in various application domains such as large-scale Internet applications and ubiquitous computing.

Notwithstanding their popularity, current DEBS applications are still often developed in an informal process and are not modularized. On the individual event level, current DEBS developers can define what events a component can accept and publish, and, by registering event handlers, what action an event can trigger. Currently, developers lack structuring mechanisms for representing event interactions and dependencies in a modular way. While current research has made fruitful contributions to various aspects in the DEBS paradigm, such as, event delivery, event detection and composition, event visibility, its emphasis is on the individual event level.

In this thesis, we advocate that by designing a new DEBS metamodel with extended behavioral interfaces and high-level structure mechanisms, we can (1) define an interface-based modular approach to model and design DEBS applications, (2) implement a prototype framework on a P2P network that provides built-in support to our proposed interface-based DEBS development, and (3) provide case studies illustrating the interface-based development process and the applicability of our proposed approach.

Acknowledgements

I would like to thank my supervisor, Prof. Paulo Alencar, who has provided me with invaluable support in producing this work. I would also like to thank Prof. Donald Cowan and Prof. Daniel Berry for reading my thesis and for providing helpful comments.

I would like to thank Rolando Maldonado Blanco for his advice and help.

Dedication

To my parents and my wife.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In component-based software engineering (CBSE), an application system is decomposed into functional or logical components with well-defined interfaces used for communication across the components. These components usually take the form of a collection of objects, and provide a higher level of design abstractions than objects [37, 24, 12, 13, 1]. The components can be bound together to provide services to form a higher-level component.

A component can be treated as a black box and abstracted using component interfaces. The Interface Definition Language (IDL) is a specification language used to describe a software component's interface in a language-neutral way, enabling communication between software components without a shared language. Current IDL is often used to describe Remote Procedure Call (RPC). For example, an interface definition written in OMG IDL defines the interface and fully specifies each operation's parameters [11].

Together with their externally visible properties and composition relations, components constitute the software architecture which can be formally captured in Architecture Description Language (ADL) [20]. Based on software modules and communication designs, systems can be classified into different architecture styles such as client-server, peer-to-peer, event-based implicit-invocation [20]. Event-based implicit-invocation is one of the architecture styles that exhibits high cohesion and loose coupling [10].

A Distributed Event-based System (DEBS) is an event-based implicit-invocation system comprised of distributed functional components interacting with each other

via events. An event represents a happenings of interest in the system or the executing environment.

In DEBS, a functional component can act as a publisher, subscriber, or both. Events are published by publisher components and are dispatched by an event delivery mechanism to these subscriber components that are interested in the occurrence of these events. Upon receiving an event, subscriber components react to the arrival event by invoking certain actions. These subscriber components can, in turn, act as publishers to publish other events to the system.

By interacting *via* events, publishers do not directly know subscribers, but subscribers' functionality is implicitly invoked by publishers *via* events. Due to its low coupling between functional components, DEBS is suitable for the development of applications in systems with a large number of functional components. Such systems are expected in ubiquitous computing environments [40], and large-scale web applications [26].

The implicit invocation of functionality *via* events, as well as the autonomy, heterogeneity, and potentially large number of components, make the development and maintenance of applications in DEBS difficult. In particular, because a DEBS exhibits its desired system behavior through the individual behavior of its functional components, it is crucial to have such component behavior modeled precisely at design time and regulated at runtime.

1.2 Current Problems

Despite increasing popularity, current DEBS development is not modular and is still an informal process poorly supported by current software engineering methodologies [7, 28].

Many current systems (including frameworks, middleware, etc.) are able to produce and consume events; however, events in such systems are not treated as first-class citizens, and are implemented in a diverse way in terms of structure and representation. In CORBA, events can be defined as part of IDL as any data structure, but, before events can be used, these IDLs must be compiled to generate helper classes as part of the method invocation [12, 19]. EJB relies on Java Message Service (JMS) to provide event support. Events in EJB are modeled as JMS messages and can be asynchronously handled by Message Driven Bean (MDB) using either a JMS Queue or a Topic. EJB events are represented using

one of JMS' predefined message types [24, 22]. Other systems, such as Hermes [29], allow an event type to be defined as an XML Schema, but they require a one-to-one (or one-to-many) mapping between an event type and an entity (or many entities) in the underlying programming language. In JINI, although events are defined as a generic event Java class, they are transported as serialized Java objects *via* Java RMI which only works on a Java platform [23, 31]. In these systems, the consumer component needs to find a way to understand received events and such event processing is left to the application programmer. We believe such ad-hoc event processing can be improved if the DEBS event can be supported as a first-class construct.

The traditional signature-based component interface definition focuses on the contract of one method in request and response semantics [11]. Such an interface definition can help define a single action associated with an individual event. However, it lacks the power to describe the components' behavior precisely in DEBS in terms of event receiving, processing and publishing. Implicit invocation causes functional components to lose control of their event processing flow; components announce events, but do not know who will respond, what the response will be, or when it will come. In designing a DEBS application, it is not enough to just define that interface B will do action *a1*, *a2* and *a3* when it receives events *e1*, *e2* and *e3*, respectively. To precisely define interface B's behavior, we must also specify at what state, from whom and in what order this interface B should receive these events. Therefore, the temporal order among event instances is critical and must be precisely captured in a component interface at design time and must be honored at runtime. To make the current interface model suitable for DEBS, it needs to be enhanced by taking into account behavior in terms of event receiving, processing and publishing. Interface composition mechanisms are also needed to define how these behavior-enhanced interfaces can be composed hierarchically.

Current DEBS development only has limited framework support. A framework is a reusable, semi-complete application containing dynamic and static components that can be customized to produce user-specific applications. A framework is usually designed for a particular application domain such as user interfaces, P2P, telecommunications, etc [6]. A framework is expected to be reused in application development to reduce the cost and improve the quality of software. Though there are many successful frameworks, such as, Java's RMI, Spring Framework, implementations of OMG's CORBA, Microsoft's MFC and DCOM, Sun's JXTA [35, 34, 12, 21, 36], these frameworks are not designed for DEBS domain and only provide limited event support on the individual-event level. Therefore, we believe

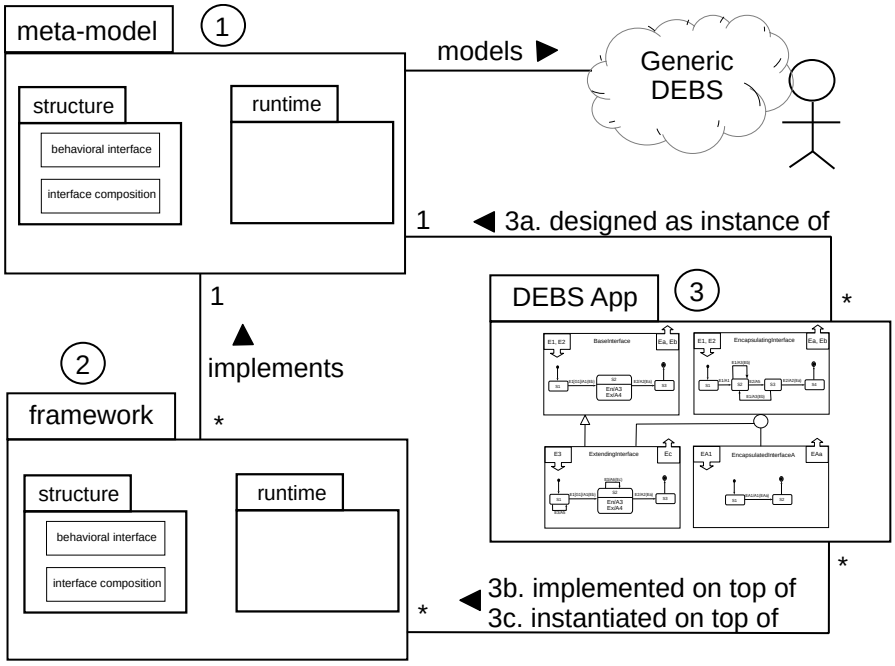


Figure 1.1: Approach Overview

the DEBS development can be better supported if we have a supporting framework that has built-in DEBS development support in a modular way.

Overall, on the individual event level, current DEBS developers can define what events a component can accept and publish, and, by registering event handlers, what action an event can trigger. Currently, developers lack structuring mechanisms for representing event interactions and dependencies in a modular way. While current research has made fruitful contributions to various aspects in the DEBS paradigm, such as, event delivery [25, 29, 16, 4], event detection and composition [2, 4, 30], event visibility [29], its emphasis is on the individual event level. As observed in [26], few hierarchical structuring mechanisms exist for the development of applications on DEBS.

1.3 Proposed Approach

To alleviate the aforementioned problems, we propose an interface-based modular approach to enhance modularity in the DEBS development. As shown in Figure 1.1, the proposed approach consists of a DEBS metamodel, a supporting framework and an interface-based development process.

1. The metamodel defines a language for modeling DEBS with a focus on behavior-enhanced interface, interface composition mechanisms and first-class event and event schema constructs.

2. The supporting framework is designed for DEBS domain and provides built-in support to our proposed interface-based DEBS development process.

3. The development process defines three steps involved in developing DEBS applications with behavior-enhanced interfaces:

   (a) decomposing a DEBS application by using interfaces, compositions and event schemas.

   (b) implementing interfaces by providing event actions and guard condition tests.

   (c) instantiating a DEBS application by assembling the required interface implementation and deploying to the supporting framework.

1.4 Contributions

In this thesis, we advocate that by designing a new DEBS metamodel with extended behavioral interfaces and high-level structure mechanisms, we can (1) define an interface-based modular approach to model and design DEBS applications, (2) implement a prototype framework on a P2P network that provides built-in support to our proposed interface-based DEBS development, and (3) provide case studies illustrating the interface-based development process and the applicability of our proposed approach.

With the behavior-enhanced interface, our approach enables developers to model component behavior precisely on the interface level in terms of event receiving, processing and publishing, thereby improving modularity in DEBS development. To accomplish this goal, we have provided the following contributions:

1. A new metamodel for DEBS that focuses on behavior-enhanced interface, interface composition mechanisms and first-class event and event schema constructs.

2. A supporting prototype framework on top of a P2P network that supports developers to build their DEBS applications based on the proposed interface-based DEBS design.

3. A representation of the interface-based DEBS development process *via* case studies and illustration of the applicability of our proposed approach.

1.5 Thesis Outline

Chapter 1 describes the introduction and briefly explains our work. Chapter 2 reviews the background and related work. Chapter 3 provides a detailed explanation of our proposed DEBS metamodel. Chapter 4 focuses on the explanation of our supporting prototype framework implementation. Chapter 5 covers the case studies and shares our experience. Finally, Chapter 6 concludes the thesis and articulates the directions of future work.

Chapter 2

Background and Related Work

2.1 Software Component Models

2.1.1 Structure Models

Sun Microsystems' Enterprise JavaBeans (EJB) [24] is a server-side component architecture for the Java Platform, Enterprise Edition (formerly known as J2EE). Based on the JavaBeans framework, EJB allows programmers to concentrate on particular business problems without worrying about transactions, threading and process control, security and other non-functional properties. The EJB container hosts components and offers lifecycle operations as well as additional services such as transactions. Before version 3.0, an EJB component consists of a class that implements an EJB interface (Entity Bean, Session Bean and Message Bean) and two other Java classes that implement Remote Interface and Home Interface, respectively. In the EJB 3.0, entity beans are superseded by the Java Persistence API, and all Enterprise JavaBeans are Plain Old Java Objects (POJO), with proper annotations.

OMG's CORBA Component Model (CCM) [13] has been adopted by OMG to extend and subsume the CORBA distributed object model [12]. In CCM, a CORBA component is introduced as a new CORBA meta type. In particular, a CORBA component is defined as a series of attributes and several implemented (or provided) interfaces called "facets". A CORBA component communicates with each other *via* "receptacles", receive events *via* "event sinks" interface and send out events *via* "event sources" interface.

Microsoft designed COM as a collocated component programming model to en-

able interprocess communication and dynamic object creation. Later, DCOM extends COM to support communication between distributed components by providing a runtime that is used to marshal and unmarshal requests and replies. DCOM has been deprecated in favor of Microsoft .NET, which provides a large body of precoded solutions to common software development requirements and manages the execution of programs written specifically for the framework. In Microsoft .NET, almost any class file is a component and is controlled by containers [1].

2.1.2 Behavior Models

UML 2 state machine diagrams, formerly referred to as state diagrams, statechart diagrams, or state-transition diagrams, depict the dynamic behavior of an entity based on its response to events, showing how the entity reacts to various events depending on the current state [15]. In this thesis, we use statechart diagrams to model the behavior of component interfaces in terms of event receiving, handling and publishing.

Cicalese et al. [3] points out that the current IDLs do not formally specify the behavior of the software component's operations and proposes a Java extension that enhances Java remote method invocation interfaces with Eiffel-style preconditions, postconditions, and invariants. However, this improvement focuses on only one operation and does not cover the temporal aspect of event processing control.

Interface automata formally models the temporal aspects of software component interfaces. Alfaro et al. [5] employs an automata-based language in an optimistic approach to capture assumptions about the order in which the components are called and the order in which the components call external methods. They also propose an approach to check the compatibility of two component interfaces. The interface automata approach differs from our proposed statechart approach in the following ways: (1) Interface automata interacts through the synchronization of input and output events, and, internally, actions of concurrent automata are interleaved asynchronously, whereas in our proposed statecharts, the interaction between interfaces and internal substatecharts is done asynchronously; (2) In interface automata, messages are not queued, and the arrival of a message while on a state not prepared to handle the message would indicate an incompatibility between the environment and the automaton, whereas in statecharts, the message would be queued until the statechart is at a state ready to handle the message.

State Chart eXtensible Markup Language (SCXML) [39] is a Public Working

Draft under review by the World Wide Web Consortium (W3C). SCXML provides a generic state-machine based execution environment based on CCXML and Harel State Tables. In this thesis, we use SCXML to describe the interface behavior.

2.2 Event Models

In CORBA [12], events can be typed or untyped and are defined in a component's IDL. Untyped events are attributes of the CORBA datatype *any* that can be cast to any datatype. On the other hand, typed event data is represented as and passed by means of typed parameters as defined in its IDL, which can be defined in any desired manner. In CORBA, an extra step is needed to compile IDLs to generate the stub and skeleton code to manipulate the event.

In EJB, event handling is done by using JMS [22]. A message in JMS is modeled as a fixed structure consisting of a header, several properties and a body. A message is created by a message producer and sent to message consumers *via* message queue which provides temporal independence between message producers and message consumers. Message processing is done by invoking registered message handlers.

In Microsoft's COM, DCOM, and .NET, events are treated, defined and created as normal objects and event processing is done *via* registered event handlers [1].

In Sun Microsystems' JINI architecture, events are the mechanisms used for asynchronous communication [23]. JINI models events as a generic RemoteEvent classes consisting of four fields: a remote reference of the source of the event, an event identifier, a sequence number, a serialized "handback" object as the payload. Event handling is bound to the Java platform and is done by invoking classes that implement RemoteEventListener *via* Java RMI. Our proposed event model also provides generic modeling of schemas and their events, but, unlike JINI, our model supports arbitrary structures in an event schema and is programming language neutral.

In Hermes [29], an event type is defined as an XML Schema and an event is represented as an XML document. XML Schema parsing is done using an Apache Xerces Parser. Event attribute accessing is done *via* XPath language. An event type is statically bound to a Java class which extends XMLDynamicBinder abstract class. An object of this class represents an event instance and can marshal and unmarshal its value *via* the toXML and fromXML method and can also export its XMLSchema *via* toXMLSchema. However, Hermes does not support event schema

to be created dynamically. As pointed out in Hermes' paper, this limit is caused because a fromXMLSchema method cannot be provided in Java as the language does not support construction of new classes at runtime.

In SARI [32], an event type library and its metamodel is provided. SARI's work is similar to ours in that we both provide an event metamodel. In SARI, attribute multiplicity is not specified, whereas in our metamodel, each attribute has a build-in lower bound and an upper bound. SARI supports event typing such as "inheritance" and "exheritence", whereas ours is part of a bigger generic DEBS metamodel and focuses more on the event structure.

2.3 Software System Meta-modeling

A model is an abstraction of phenomena in the real world. A metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest [14] and, thus, is a higher level abstraction highlighting properties of those models. A model is an instance of and always conforms to its metamodel.

OMG's Meta-Object Facility (MOF) [14] is an extensible model-driven integration framework for defining, manipulating and integrating metadata and data in a platform-independent manner. MOF-based standards are in use for integrating tools, applications and data. A number of technologies standardized by OMG, including UML, MOF, CWM, SPEM, XMI, and various UML profiles, use MOF and MOF-derived technologies for metadata-driven interchange and metadata manipulation.

Zachariadis et al. [41, 42] proposes a lightweight local component metamodel called SATIN to address adaptability issues related to a mobile system. This paper is a good example of how to practice meta-modeling: it covers designing the metamodel on MOF, designing metamodel notation, implementing the metamodel as middleware, and implementing case study applications.

2.4 Peer-to-Peer Systems

Clay Shirkey (The Accelerator Group) gives an intuitive definition of P2P as [27]:

> Peer-to-peer is a class of applications that take advantage of resources storage, cycles, content, human presence available at the edges of the

> Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers.

JXTA, introduced by Sun Microsystems, Inc., is a set of open, generalized peer-to-peer protocols that allow any connected device (cell phone to PDA, PC to server) on the network to communicate and collaborate. JXTA defines a three-layer P2P software architecture (platform, services and applications) as consisting of a set of XML-based protocols and a number of abstractions and concepts such as peer groups, pipes, and advertisements. Peers are organized in peer groups; a peer group can assemble several modules; resources are shared *via* advertisements; and application level communications are usually done *via* pipes [36, 18]. Our provided implementation is built on top of JXTA.

2.5 Event-based System Engineering

With respect to structuring DEBS, we are only aware of Fiege's proposal to use event visibility as a structuring abstraction. Fiege [8, 9] proposes the use of scope to model event visibility based on publish and subscribe semantics; the visibility of an event determines the range of the delivery of the event and, in turn, determines the range of components that can produce and react to the event. Effectively, this approach improves the event delivery mechanism but does not cover the methodologies for the identification and modeling of the structural and other properties of the DEBS.

Chapter 3

A Metamodel for DEBS

3.1 Metamodel Overview

Focusing on first-class event constructs, enhanced behavioral component interfaces, and interface composition mechanisms, our proposed metamodel models the essential elements in a generic DEBS with respect to static structure and runtime semantics.

The static structure of the metamodel is depicted in Figure 3.1 and Figure 3.2. An event schema defines the data structure to which all its events must conform. A component interface defines its input and output event schemas and the behavior specified as a finite state machine. Two composition mechanisms, extend and encapsulate, can be used to construct new interfaces from existing ones in a modular way. An extending interface will have its own features and those features as defined in its extended interfaces, whereas an encapsulating interface can hide and coordinate all its encapsulated interfaces. A component interface can have multiple implementations.

Runtime semantics are illustrated in Figure 3.3. A reactive component instance manages the life cycle of its hosted component interface implementation instances, each of which represents an instantiated component interface. According to its interface definition, a component interface implementation instance can receive, process and publish events. An event is dispatched *via* an event bus and is delivered to those component interface implementation instances that are interested in that dispatched event.

In the following sections, we first explain the generic event model. Then, we present the behavior-enhanced interface model, followed by two interface composi-

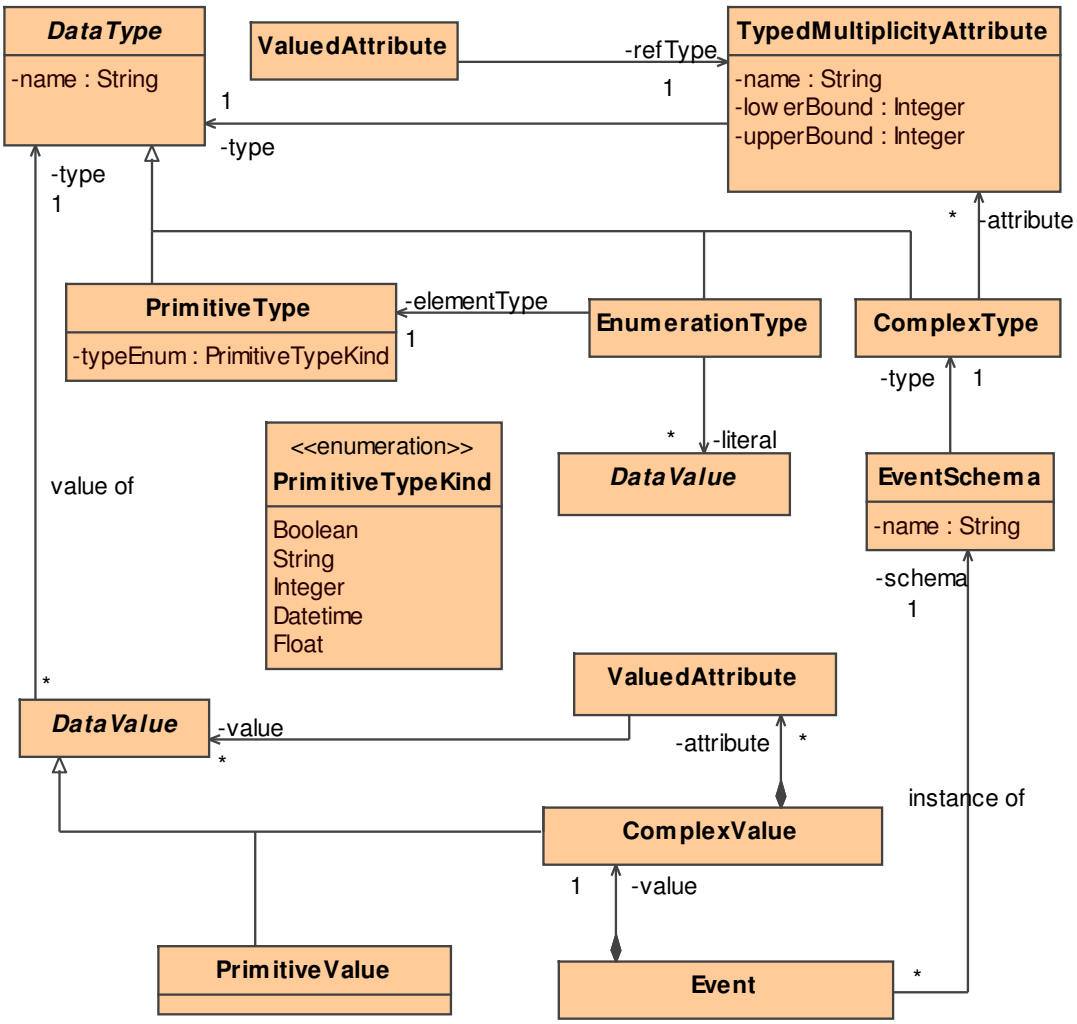


Figure 3.1: DEBS Event Schema Diagram

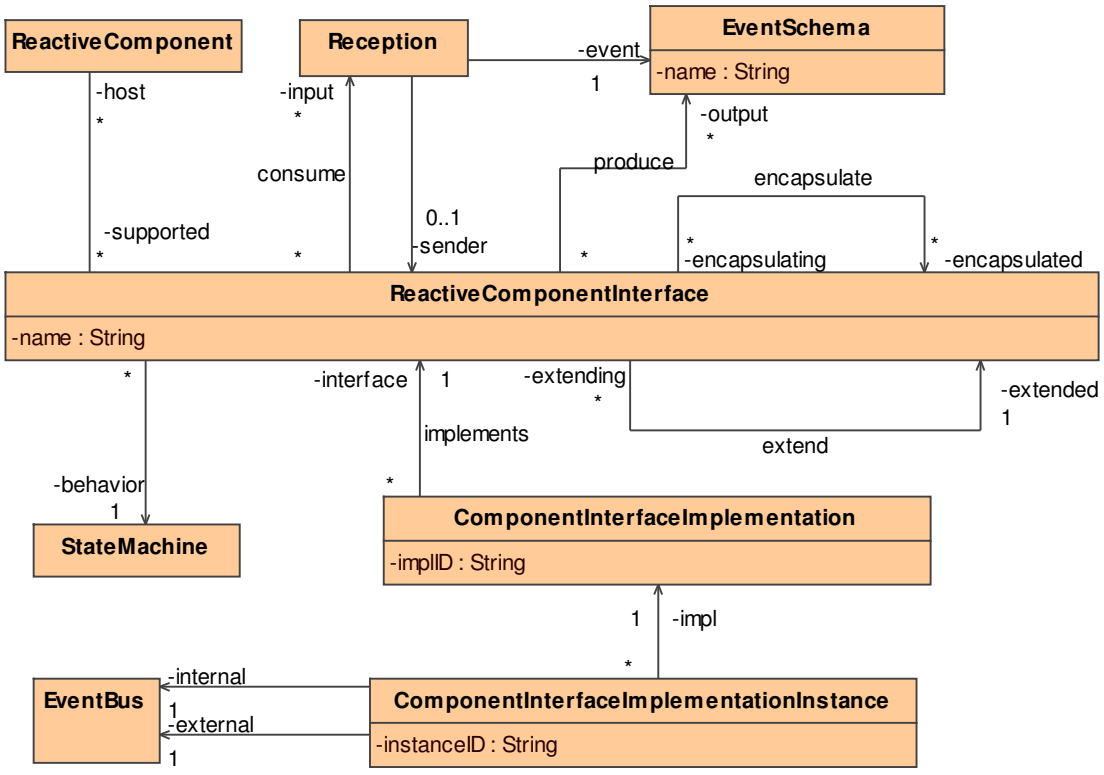


Figure 3.2: DEBS Component Interface Diagram

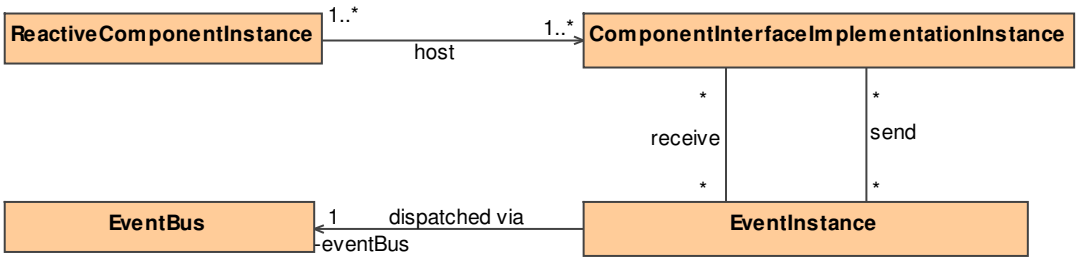


Figure 3.3: DEBS Runtime Diagram

tion mechanisms. Finally, we introduce diagrammatic notation, which can be used during DEBS development.

3.2 Generic Event Model

In our metamodel, EventSchema and Event are modeled as first-class citizens (Figure 3.1).

EventSchema Similar to a Java class defining the type of its Java object instances, an EventSchema defines the data structure of its event instances used within a DEBS. Each EventSchema has a unique name distinguishing one schema from another. An EventSchema must associate with exactly one ComplexType as its type. At runtime, an EventSchema can be dynamically defined and removed in a DEBS, and all EventSchemas must be first defined and registered in a DEBS before their event instances can be used.

ComplexType A ComplexType represents a user-defined data type which can have other nested data types. A ComplexType is identifiable by its name (inherited from DataType), and its nested data types are specified through its associations with multiple TypedMultiplicityAttribute.

TypedMultiplicityAttribute A TypedMultiplicityAttribute models an attribute in an event schema. Each attribute has a name unique in its defined event schema. The lowerBound and the upperBound specify the number of times the values of this attribute may occur in an event instance. The lowerBound is greater than or equal to zero and must be less than or equal to the upperBound. The TypedMultiplicityAttribute is optional if its lowerBound is zero, otherwise, it is mandatory. A ComplexType associated with no TypedMultiplicityAttribute is an empty type and used only to define an empty EventSchema which does not possess any attributes.

DataType A DataType is the ancestor of all possible attribute types. It is abstract and has a type name which identifies it from others. The semantics of DataType are defined by its subtypes: PrimitiveType, EnumerationType and ComplexType.

PrimitiveType A subtype of DataType, PrimitiveType specifies the primitive types that can be used to define an attribute type. The available primitive types are defined in PrimitiveTypeKind.

PrimitiveTypeKind A PrimitiveTypeKind enumerates the available primitive types that can be used to define an attribute type. The available primitive types are Boolean, String, Integer, Datetime, Float.

EnumerationType An EnumerationType is a subtype of DataType. Each EnumerationType has a unique name and must associate with one PrimitiveType as its element type of predefined values specified *via* its associated multiple DataValue.

Event As the essential communication block in a DEBS, an event represents a runtime instance of its event schema. Every event must be associated with exactly one event schema specifying the data structure of the event. An event has one ComplexValue which represents the values in an event. Events should only be constructed from their event schema to guarantee correctness. An event is published by a ComponentInterfaceImplementationinstance and delivered *via* an EventBus to one or more ComponentInterfaceImplementationinstances that are interested in this event.

Also, in a DEBS, at time t, an event instance is said to be *valid* if and only if: (1) its associated schema exists in the DEBS at time t; (2) each attribute value in valued-attributes is a valid instance conforming to its attribute type as defined in its associated event schema. Therefore, if an event schema is dropped from a DEBS at time t, then at $t+1$, all its event instances will be considered as *invalid*.

An event schema defines the static data structure but not the associated event processing semantics of its event instances. The semantics of when to send and receive an event and how to react to an event is defined in an interface's behavior (details are explained in Section 3.3).

ValuedAttribute A ValuedAttribute models an attribute value at runtime. A ValuedAttribute has a reference to its defining TypedMultiplicityAttribute which specifies the type and multiplicity of the multiple DataValues associated with this ValuedAttribute. If the attribute type is a primitive type, the data value of this attribute must be a primitive value that conforms to the type. If the attribute type is an enumeration type, the value must be one of those pre-defined values. If the attribute

type is a complex type, the data value must be a complex value with all nested data values in accordance with their associated types. Such value-type matching is performed recursively until a primitive value is matched with its primitive type. Also, the number of DataValues must be inclusively bound by the lowerBound and upperBound as specified in its associated TypedMultiplicityAttribute. Therefore, a ValuedAttribute is valid if and only if it has the correct number of compatible values as specified in its associated TypedMultiplicityAttribute.

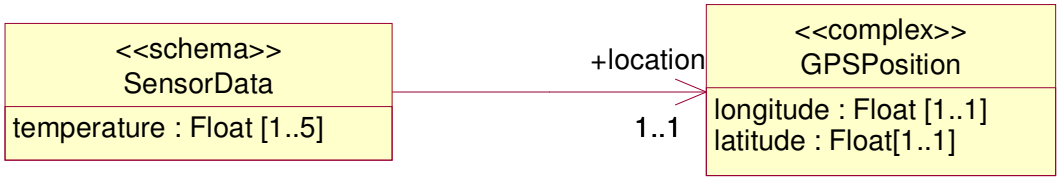
DataValue A DataValue represents an abstract data value of its associated data type. The actual value is specified by its subtypes, PrimitveValue and ComplexValue.

PrimitiveValue A PrimitiveValue models a primitive value of an associated primitive type. Depending on a particular metamodel implementation, a primitive value can be coded as a string representation which can later be interpreted, or directly mapped to a native representation in its underlying programming language. For example, in Java, a value of a Datatime can be represented as an object of java.util.Time.

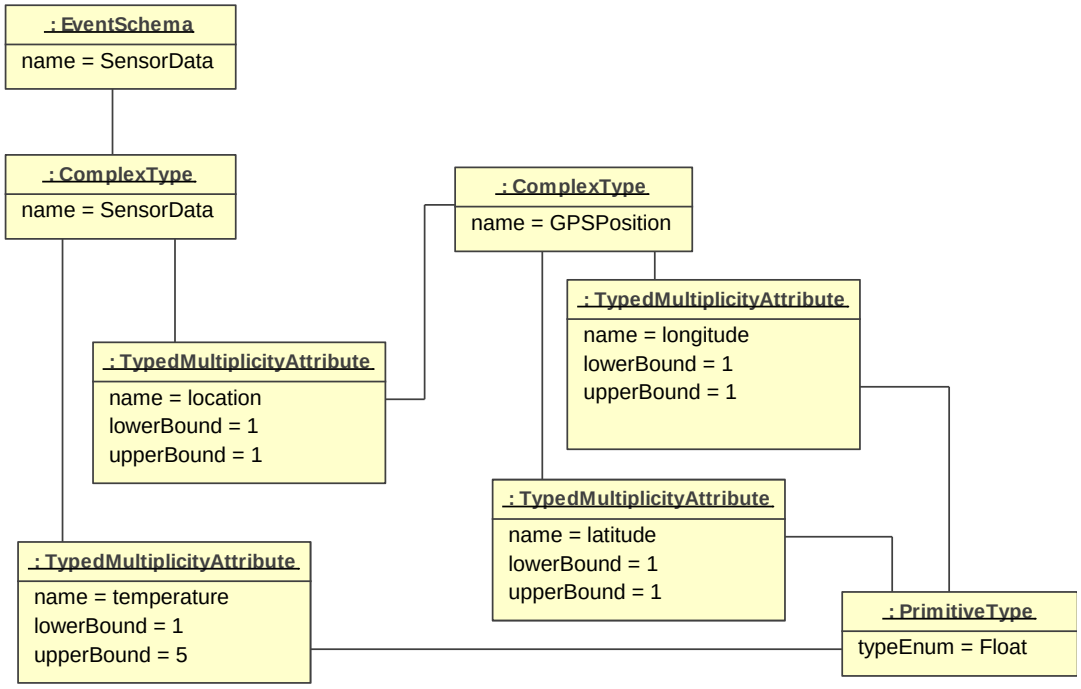
ComplexValue A ComplexValue extends a DataValue and represents a value of a ComplexType. A ComplexValue always associates itself with one ComplexType and contains a collection of ValuedAttributes, each of which represents a particular attribute value in the ComplexValue. A ComplexValue is valid if and only if (1) the data type associated with the ComplexValue is a type of ComplexType, (2) for every mandatory TypedMultiplicityAttribute in the associated ComplexType, the ComplexValue contains a valid ValuedAttribute that refers to the TypedMultiplicity-Attribute, and (3) for every optional TypedMultiplicityAttribute, it is valid to have no ValuedAttributes referring to this TypedMultiplicityAttribute (but if the optional TypedMultiplicityAttribute does have a ValuedAttribute referring to it, the ValuedAttribute must be valid).

3.2.1 Examples

Based on the metamodel above, an event schema and its event instances can be created dynamically. For instance, in our case study 5.2, we describe a temperature sensor system consisting of multiple sensors and one information center. Each sensor has its predefined GPS position and, from time to time, will report its



(a) SensorData Event Schema



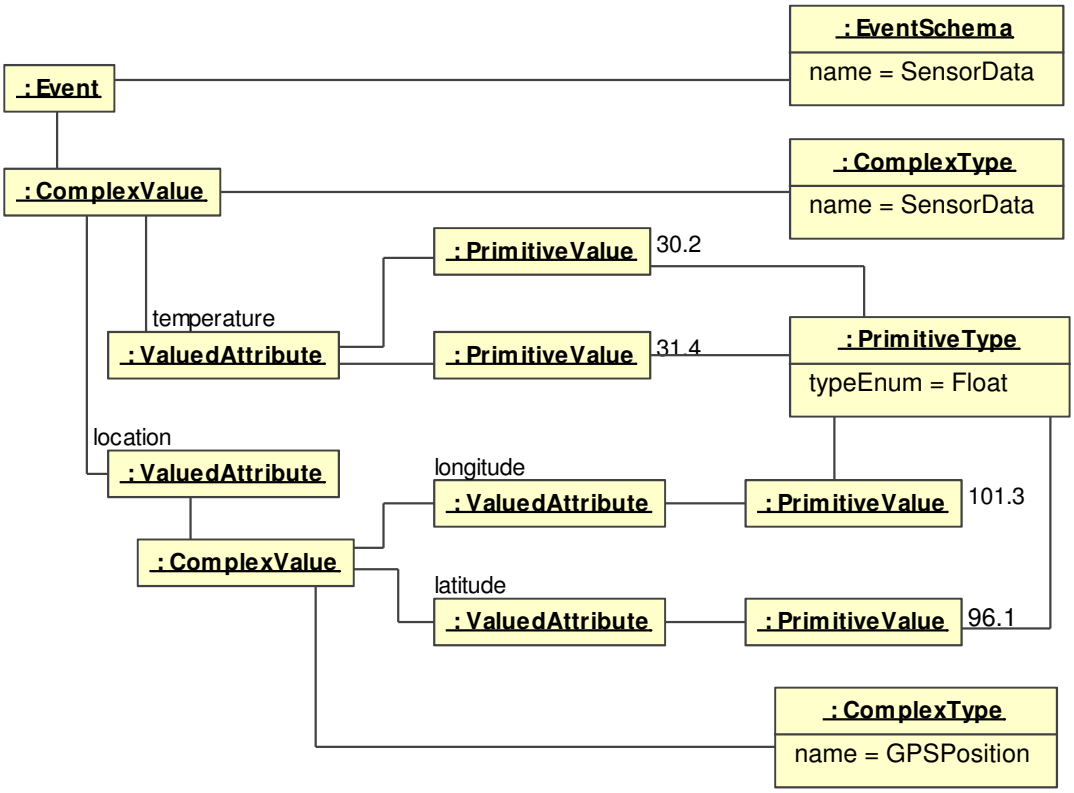
(b) SensorData Schema as Metamodel Instances

Figure 3.4: SensorData Event Schema Example

collected temperature data *via* events to an information center. An event must carry the sensor's GPS position and can carry one to five temperature data. Based on our event schema model, we can define a SensorData event schema with its specification represented diagrammatically in Figure 3.4(a).

Such an event schema has a temperature attribute of float type and has an attribute location which is typed as a complex type called GPSPosition. Any value of the SensorData can have one to five temperature values and must have one GPSPosition value. The GPSPosition complex type has two attributes: longitude and latitude, both of which are of type float. Any value of a GPSPosition must have one longitude value and one latitude value. Figure 3.4(b) illustrates how such event schema specification can be dynamically modeled as instances of the metamodel elements described above.

(a) SensorData Event



(b) SensorData Event Instance as Metamodel Instances

Figure 3.5: SensorData Event Example

From the event schema, events can be created and values can be assigned dynamically. Such events can also be modeled as instances of metamodel elements. For instance, Figure 3.5(a) depicts a SensorData event instance which has two temperature values, 30.2 and 31.4. The SensorData event also has one location value with longitude at 101.3 and latitude at 96.1, and Figure 3.5(b) illustrates the modeling of this event as instances of the metamodel elements described above. Due to space constraints, we use four labels to denote the actual four links between four ValuedAttribute instances and their corresponding TypedMultiplicityAttribute instances.

3.3 Behavior-enhanced Interface Model

Reactive component interfaces and interface composition mechanisms are the core part of our proposed metamodel toward DEBS modularization. As a first-class building block, a reactive component interface modularizes the discrete event manipulation by defining (1) what events it can receive and publish, (2) if necessary, from whom it should receive such events, (3) when it can receive and publish events, and (4) how to process an event. The composition mechanisms define how reactive component interfaces can be composed together (e.g., extends, encapsulates) to build more complicated interfaces. The metamodel elements of the behavior-enhanced interface model are outlined in Figure 3.2.

ReactiveComponentInterface The ReactiveComponentInterface represents the building blocks in a DEBS and regulates event processing through its behavior specification. Each ReactiveComponentInterface possesses a unique interface name within its defining DEBS. The ReactiveComponentInterface specifies that it will receive incoming events where event schema and sender ReactiveComponentInterface match those defined in its associated Reception. Also, the ReactiveComponentInterface specifies all the event schemas that this interface will send out through the produce association with EventSchema. The ReactiveComponentInterface can encapsulate other ReactiveComponentInterfaces and may extend other interfaces. These two associations (i.e., encapsulte and extend) represent the interface composition mechanisms, whose semantics are discussed in detail in Section 3.4.

Reception Each Reception is associated with one event schema and optionally one sender ReactiveComponentInterface. If an event schema is not associated with

a sender interface, the defining interface will receive all events of this event schema sent from all interfaces.

StateMachine In general, an interface behavior specifies an interface protocol specification and answers the question of "*when* and *how* to react to *what* events". Each state, together with its associated triggering events, defines the *when* and *what*: certain events can only be received when this interface is at a particular state. The transitions associated with each state, together with its guard conditions and actions, specifies the *how*: a received event will be processed according to associated actions and will cause the interface to transit to its next state as defined in transition if the associated guard conditions are satisfied.

In our metamodel, the behavior of an interface is defined using a Finite State Machine. We reuse UML's state machine metamodel with customized semantics, so the StateMachine metamodel is omitted here and readers are referred to UML 2.0 specifications for detailed information [15].

The following semantic changes are made to overcome some drawbacks of the UML state machine as observed by Simons in [33]:

- The UML state machine allows transitions to go directly to a state inside a composite state from outside. Such boundary crossing transitions violate encapsulation and, therefore, are not allowed in our metamodel.
- When an event triggers multiple transitions of a state and its composing state, the UML state machine gives priority to the inner-most transitions, and such choice again infringes the encapsulation. Therefore, in our metamodel, we follow the semantics defined in the Harel state machine [17] to let the outer-most transitions have precedence over the inner ones.

In addition, the semantic variation point of handling unexpected events in the UML state machine is also specified in our metamodel: if an interface in any state receives an unexpected event (i.e., an event not expected to receive at this state), the interface state machine will be forced to transit to a special error state indicating this error occurrence. However, what to do after getting into this error state (e.g., backtrack to the previous state before the error occurred, after the error is fixed, or totally lock up the interface) is implementation specific and not specified in the metamodel.

ComponentInterfaceImplementation A ReactiveComponentInterface can have multiple implementations (e.g., for different platforms, programming languages). A ComponentInterfaceImplementation has a unique implementation ID and possesses a reference to a ReactiveComponentInterface which it implements.

ComponentInterfaceImplementationInstance The ComponentInterfaceImplementationInstance is the instantiated instance of its corresponding ComponentInterfaceImplementation. At runtime, a ComponentInterfaceImplementation can have multiple instances. In particular, a ComponentInterfaceImplementationInstance is constructed every time a ComponentInterfaceImplementation is loaded into a DEBS. Every ComponentInterfaceImplementationInstance has an instanceID uniquely identifying itself, and every ComponentInterfaceImplementationInstance is associated with two EventBuses: external and internal. A ComponentInterfaceImplementationInstance perceives and reacts to the environment where it resides by receiving and publishing events to its external event bus, whereas its internal event bus represents a private environment shared only by an encapsulating instance and its encapsulated instances (details are explained in Section 4.2.6).

ReactiveComponent A ReactiveComponent can support more than one ReactiveComponentInterface. By support, we mean that at runtime, for each supported ReactiveComponentInterface, the ReactiveComponent will have one of its ComponentInterfaceImplementations loaded. The ReactiveComponent in our metamodel is not treated as a monolithic implementation of all its supported interfaces. Instead, it represents a container at runtime where interface implementation instances can be loaded and run. More detailed discussion of this ReactiveComponent implementation is covered in Section 4.3.2.

3.4 Interface Composition Mechanisms

Component interfaces represent the primitive building block in a DEBS, whereas the interface composition mechanisms define how such building blocks can be glued together to construct new building blocks in a modular way. In this metamodel, we propose two essential composition mechanisms: extend and encapsulate.

3.4.1 Extend

Similar to inheritance in OO-programming, extend is a vertical composition mechanism which defines how the features (i.e., input and output event schemas, behaviors) of the extending component interface can be extended with features of the extended component interfaces. The extend mechanism provides a means to build a DEBS incrementally in a modular way. For example, we can first define a generic component interface which handles common issues required in a particular system (e.g., start and stop, bookkeeping), and then we can incrementally build other application-specific component interfaces on top of this generic component interface.

An interface can directly and indirectly extend more than one other interface and can be both directly and indirectly extended by more than one interface. However, along the path of the extension, no loop is allowed (i.e., an interface cannot directly or indirectly extend itself).

Semantically, if an interface i extends another interface j, i will have its own features plus all the features defined in j, including features from indirectly extended interfaces. By features, we mean input and output events, and behavior state machine. Specifically, let i extend j and i_{ext} be the imaginary resulting interface, then i will conceptually have identical characteristics to i_{ext} which is described below:

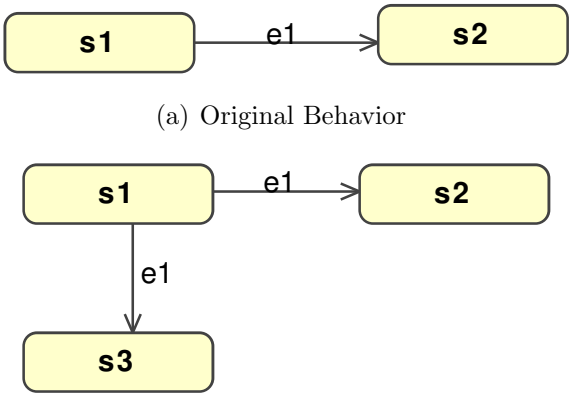
1. i_{ext} can receive input events which are the union of i's own input events and all input events defined in all interfaces that i extends.

2. i_{ext} can send output events which are the union of i's own output events and all output events defined in all interfaces that i extends.

3. the behavior of i_{ext} will be the behavior as described by the state machine resulting from a conservative combination of i's state machine and the state machine of all interfaces that i extends.

By conservative combination, we mean that if i extends j, j's behavior must remain intact and the delta changes introduced by i on top of j (e.g., new event schemes, new transition relations) should not conflict with the existing definition of j, and the resulting state machine after the interface extending process must be deterministic.

For instance, Figure 3.6(a) shows part of an interface behavior which is extended in a non-conservative extension as shown in Figure 3.6(b). The behavior extension



(a) Original Behavior

(b) Extended Behavior with Non-deterministic Transitions

Figure 3.6: A Typical Example of Non-conservative Behavior Extension

introduces non-deterministic transitions to state $s1$ when $e1$ is received. While outside the scope of this thesis, one possible way to deal with this problem could be to enhance IDE to provide help during the behavior extension process at design time. For now, we assume this extension process is done by developers manually and, therefore, expect a framework implementation to be able to detect and report such non-conservative behavior extension at runtime.

In terms of the interface compatibility, i's implementation is downward compatible with all the interfaces that i extends. Specifically, for any interface k that i extends, wherever k's implementation is expected, i's implementation can be used instead.

3.4.2 Encapsulate

As the name implies, this encapsulate mechanism is inspired by information hiding in OO-programming. It is a horizontal composition mechanism between an encapsulating interface and its encapsulated interfaces, and describes how a new complex interface can be derived from encapsulating other smaller interfaces. Similar to *facet* in the OO design pattern, this encapsulation rule is useful in cases where a new interface can be derived by composing other existing interfaces without exposing the composed interfaces to the outside world.

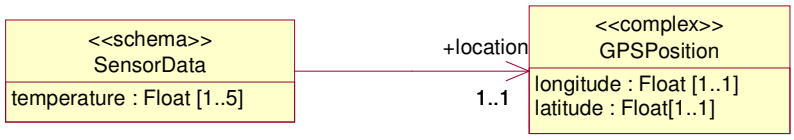
An interface can directly encapsulate more than one other interface and can be directly encapsulated by more than one other interface. However, along the path of the encapsulation, an interface cannot directly or indirectly encapsulate itself.

Semantically speaking, the encapsulating interface acts as *facade*, *coordinator* and *proxy* for all its encapsulated interfaces. At runtime, an interface implementation instance has a pair of event buses: external and internal. The external event bus is the outside world for the interface and the internal event bus is its private world shared by its encapsulated interfaces. It is always true that the internal event bus of an encapsulating interface instance is the external event bus of its encapsulated interface instances.

In this way, an encapsulating interface works as a *facade* which communicates with the outside world by receiving events from and publishing events to its associated external event bus. Upon arrival of an external event, an encapsulating interface will act as *coordinator* with its behavior as defined in the behavior state machine definition: it can process the event directly, or it can forward the event (possibly after pre-processing and transformation) to its encapsulated interfaces *via* its internal event bus. An encapsulating interface can publish any events to its internal event bus as long as the corresponding event schemas are defined in the input event schemas of its encapsulated interfaces.

Upon assembly, an encapsulated interface implementation instance will be assigned an external event bus from its encapsulating interface. Then, all encapsulated interfaces will collaborate with other colleague-interfaces by exchanging events *via* their external event bus, and usually, such collaboration is under the control of the encapsulating interface *via* its internal event bus. Similar to handling external events, the encapsulating interface can handle events sent from its encapsulated interfaces: it can process the event directly, or it can, like a *proxy*, forward events (possibly after pre-processing and transformation) to the outside world *via* its external event bus. An encapsulating interface can receive any event from its internal event bus as long as the corresponding event schemas are defined in its encapsulated interfaces output event schemas.

An encapsulating interface only hides and coordinates its directly-encapsulated children interfaces, each of which can, in turn, hide and coordinate its own encapsulated interfaces. Interface encapsulating differs from normal state machine composition in that state machine composition results in a bigger, monolithic state machine from other state machines (usually, the result state machine is run as a single instance in the program space), whereas interface encapsulating provides a modular approach to the composition of existing interfaces. Thus, the implementation instances of encapsulated interfaces can share the program space with their encapsulating interface, or they can have their own totally different executing environment.



(a) Event Schema Notation

(b) Event Notation

Figure 3.7: Notation – Event Schema and Event

3.5 Notation

Along with the metamodel, we propose a diagrammatic notation to represent the event schema, event, component interface, component interface instance and the extend and encapsulate composition mechanisms. The notation can be regarded as an adaptation of the well-known UML notation and can be used to help model DEBS applications with proposed behavior-enhanced interfaces.

EventSchema Figure 3.7(a) shows an event schema called SensorData. The SensorData event schema has a temperature attribute of float type and has an attribute location which is typed as a complex type called GPSPosition. Any value of SensorData can have one to five temperature values and must have one GPSPosition value. The GPSPosition complex type has two attributes: longitude and latitude, both of which are of type float. Any value of GPSPosition must have one longitude value and one latitude value.

Event Figure 3.7(b) depicts a SensorData event instance which has two temperature values, 30.2 and 31.4, and one location value with a longitude of 101.3 and a latitude of 96.1.

ReactiveComponentInterface Figure 3.8 depicts a reactive component interface called ReactiveComponentInterface. The input event schemas of this interface are E1 and E2 shown in the upper-left corner. The output event schemas of this interface are Ea and Eb shown in the upper-right corner. The behavior of this event

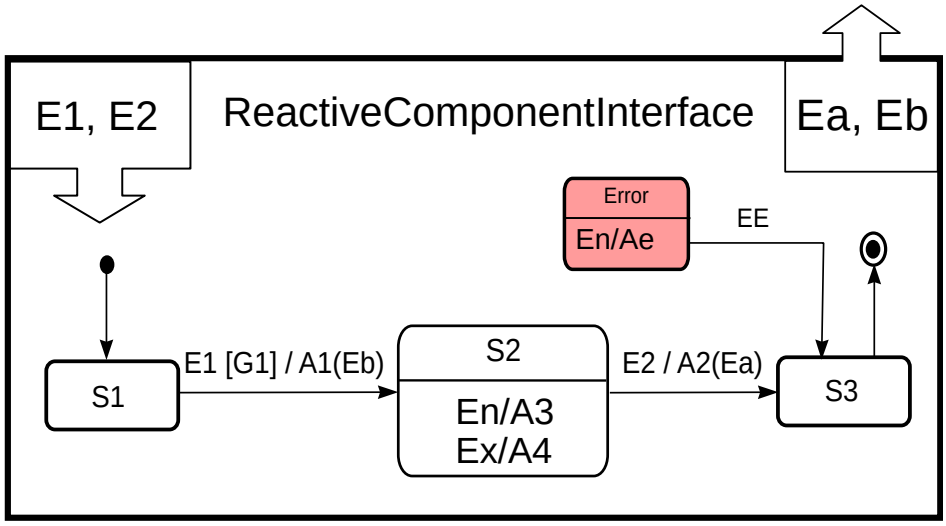


Figure 3.8: Notation – Reactive Component Interface

interface is shown as a statechart in the middle. The initial state of this interface is S1. When the interface receives an event of E1 at state S1, if the transition guard G1 is satisfied, it will perform action A1 (this action will send out an event of Eb) and transit to state S2. Upon entering S2, the entry action A3 will be executed. When the interface receives E2 at state S2, it will perform action A2 (this action will send out an event of Ea) and transit to state S3. When this interface transits out from S2, exit action A4 will be invoked. Once entering S3, it will transit to a final state automatically.

Also shown in Figure 3.8 is a special error state called Error. As we mentioned before, all unexpected events will cause this interface to transit to this Error state. The action Ae is invoked every time when the Error state is entered. From this Error state, only an event EE will cause the interface to transit back to state S3.

Extend The interface extension is represented as the same notation used for representing class extension in UML. For instance, Figure 3.9 shows the case where an interface, ExtendingInterface, extends another interface, BaseInterface, with the following delta changes:

1. E3 is added to its input event schemas and Ec is added to its output event schemas.

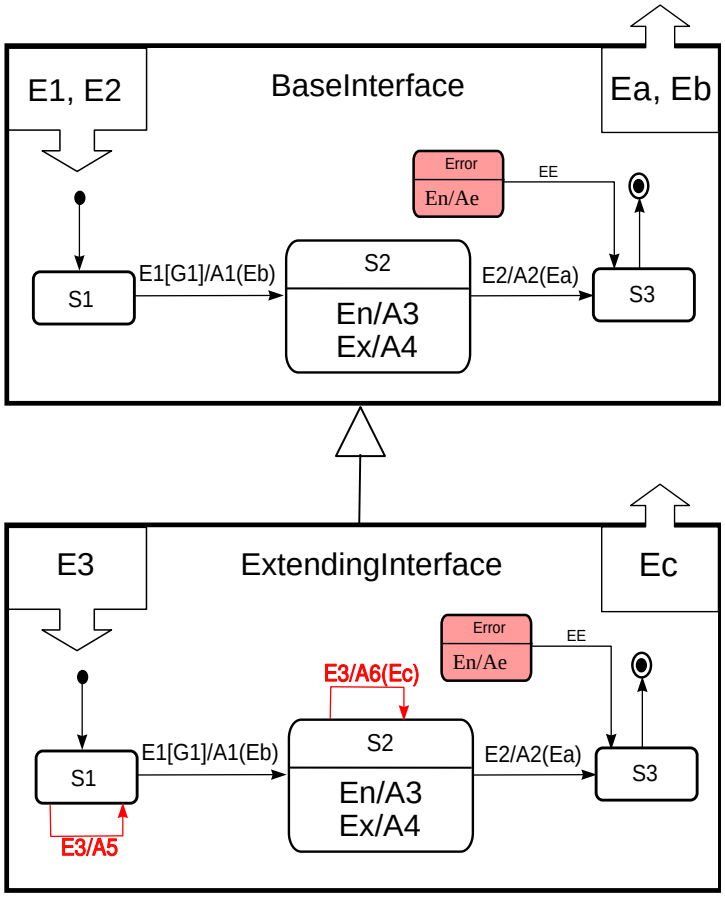


Figure 3.9: Notation – Interface Extension

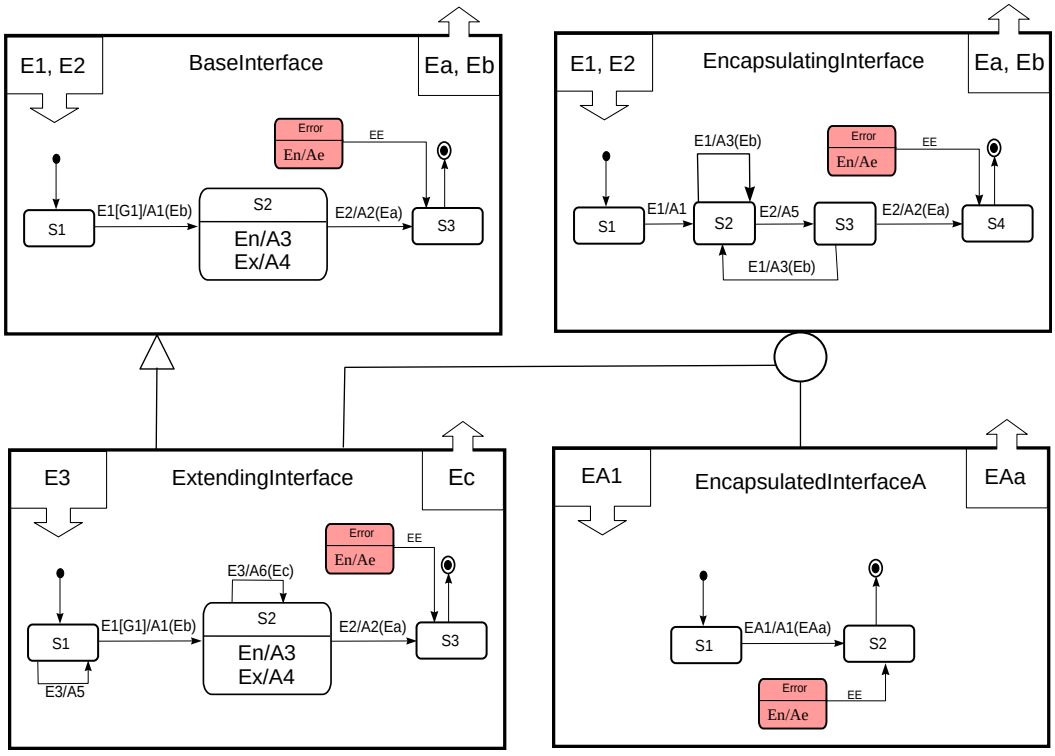


Figure 3.10: Notation – Interface Encapsulation

2. At state S1, when E3 is received, action A5 will be executed and the state will remain in S1.

3. At state S2, when E3 is received, action A6 will be executed (this action will send out an event of Ec) and the state will remain in S2.

Encapsulate The interface encapsulation is denoted as a circle attached to the bottom of the box representing an encapsulating interface. An encapsulated interface is connected by a line that connects the circle attached to the encapsulating interface and the box representing this encapsulated interface. As shown in Figure 3.10, an interface, EncapsulatingInterface, encapsulates two other interfaces, ExtendingInterface and EncapsulatedInterfaceA. The ExtendingInterface in turn extends a BaseInterface. As described above, the EncapsulatingInterface will hide and coordinate the ExtendingInterface and EncapsulatedInterfaceA according to its behavior definition.

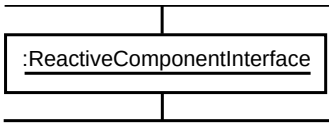


Figure 3.11: Notation – Interface Implementation Instance

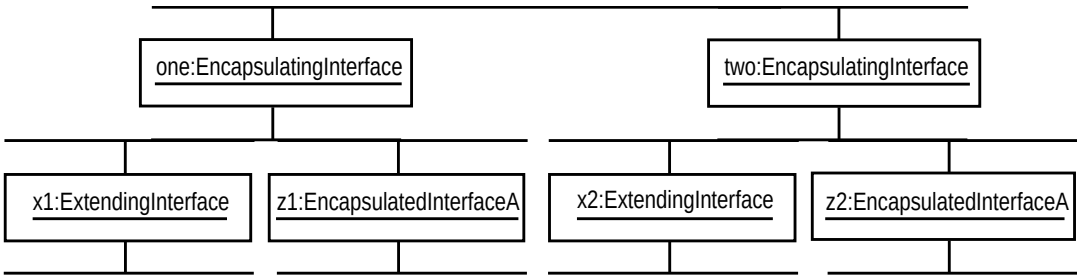


Figure 3.12: Notation – Interface Implementation Instance Diagram

ComponentInterfaceImplementationInstance Figure 3.11 shows the notation of an interface implementation instance. Each box represents an interface implementation instance at runtime. The line above the box represents the external event bus, whereas the one below represents the internal one. An instance diagram for Figure 3.10 is shown in Figure 3.12 which illustrates the interface implementation instance diagram of two encapsulating interface implementation instances. All six interface instances could co-exist in the same host machine, or could reside in six physically different hosts.

Chapter 4

Framework Implementation

4.1 Framework Overview

The previous chapter introduces a DEBS metamodel with generic event and enhanced behavioral component interfaces. In this chapter, we introduce a prototype framework implementation – **G**eneric **E**vent-based fra**M**ework (GEM), based on the JXTA network. After being deployed in participating hosts, GEM forms an overlay network for DEBS over JXTA. GEM is designed for DEBS domain and provides built-in support to our proposed interface-based DEBS development process. Specifically, GEM provides: (1) generic event management, (2) behavior-extended interface definition and composition, (3) runtime interface behavior enforcement. Also, XML is extensively used in our framework to define event schema, interface and interface behavior, interface implementation definition, and to marshal event instances.

The high-level overview of the framework architecture is depicted in Figure 4.1. Based on the underlying JXTA network, GEM's Framework Core Service provides core framework services such as component interface addressing and event delivering to all higher-level components, including the system core component and application components. The framework core service is implemented as a JXTA service and can be loaded into a JXTA peer group.

On top of the Framework Core Service, we have one core component and multiple application components. Currently, the core component hosts two built-in service interfaces (SchemaServiceInterface and InterfaceServiceInterface) that are loaded during framework bootstrap. The SchemaServiceInterface provides the framework event schema service, and the InterfaceServiceInterface provides the component interface

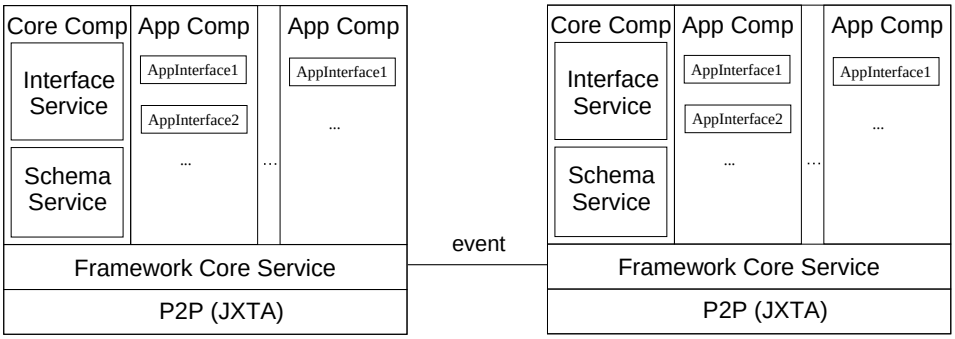


Figure 4.1: High Level Overview of Framework Architecture

service (details are described in Section 4.3.3 and 4.3.4). We can request the framework to create multiple application components which can assemble application specific component interfaces.

The steps to bootstrap GEM is outlined below. First, the JXTA network is launched and then the Framework Core Service is loaded into the peer group. Once loaded, the Framework Core Service will take over the rest of GEM bootstrap. It loads in built-in event schemas, built-in service interfaces and their default implementation, and assembles SchemaServiceInterface and InterfaceServiceInterface into a framework core component. The bootstrap completes when the framework core component starts and then the framework is instantiated and ready to go.

In the following section, we first introduce the framework core service and then explain the framework built-in event schemas and built-in interfaces with their default implementation.

4.2 Framework Core Service

The part of the framework implementation relating to the interfaces, interface implementation and components is shown in Figure 4.2. The FrameworkCoreService is implemented as a JXTA service and can be loaded into a hosting JXTA peer group. The framework core service is the only part that interacts with the underlying JXTA network and provides the essential framework services to the other parts of the framework. The main services provided by the framework core service includes, but is not limited to, event schema loading, interface loading, interface implementation loading, event handling (e.g., dispatching, receiving and forwarding to its target interfaces) and component handling (e.g., start and stop components).

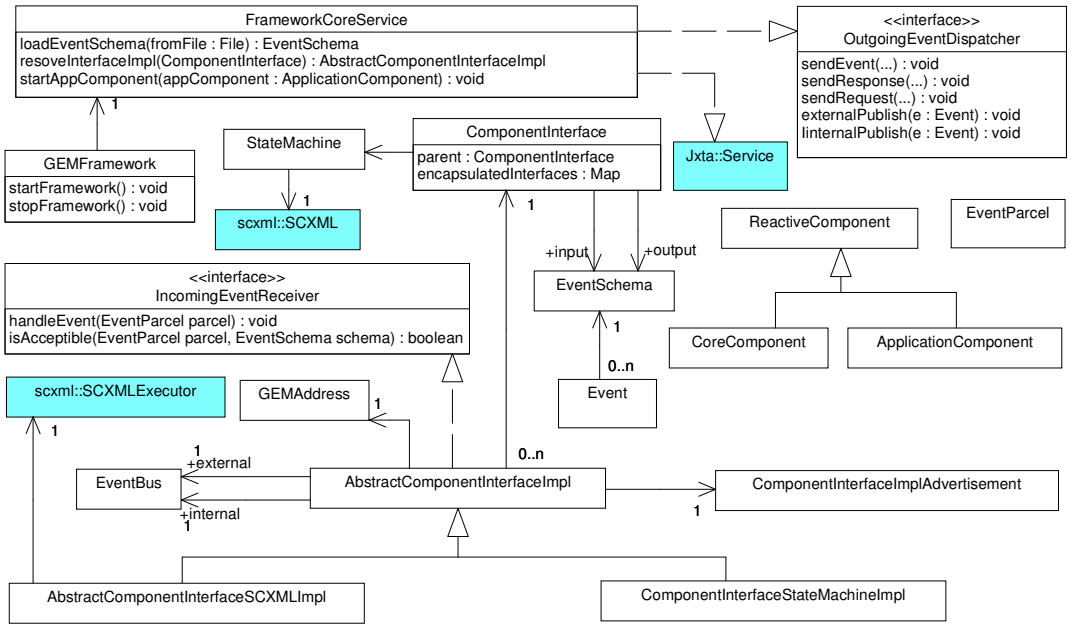


Figure 4.2: Framework Core

4.2.1 Interface Addressing and Event Delivery Semantics

When requested to dispatch an event, the FrameworkCoreService relies on JXTA's pipe service for the actual event delivery. GEM employs a novel interface addressing system which allows an event's target interface (or interfaces) to be addressed as: (1) Instance Specific, (2) Implementation Specific, (3) Interface Specific, and (4) Definition Specific. In addition, the framework also allows an event to be delivered as: (1) a normal event, (2) a request, or (3) a response.

Effectively, GEM serves as a generic event-based framework with flexible ways to deliver events. In fact, the pure DEBS event delivery semantics are achieved by **always** sending normal events addressed as definition specific. With its flexible event delivery semantics, GEM allows developers the flexibility to explore their DEBS design in different styles, either as a pure DEBS, or as an event/RPC-hybrid system. The two built-in service interfaces in our framework are implemented in an event/RPC-hybrid style, whereas the two case studies are all implemented in a pure DEBS style.

In the following subsections, we will introduce the interface addressing system and then the event delivery mode.

Interface Addressing System

In our framework, every interface implementation instance can be uniquely identified using an interface address – GEMAddress. A GEMAddress consists of an interface name, an interface implementation ID and an interface implementation instance ID. The interface name is the unique name of an interface in the framework; the interface implementation ID uniquely identifies a particular interface implementation, and the interface implementation instance ID is assigned to an implementation instance when it is loaded into the framework. In URL format, a GEMAddress can be represented as gem://interface-name/interface-impl-id/interface-impl-instance-id. For example, four cases are shown below:

1. gem://gem.core.EventSchemaServiceInterface/scxml/6bcc0ae8-0d23-495c-8cad-dec7ad857cbd represents a unique interface implementation instance. The interface instance is an instance of the sxcml implementation of the interface gem.core.EventSchemaServiceInterface. The interface instance is assigned with a UUID 6bcc0ae8-0d23-495c-8cad-dec7ad857cbd when it is loaded into the framework.

2. gem://gem.core.EventSchemaServiceInterface/scxml/* represents all scxml implementation instances of interface gem.core.EventSchemaServiceInterface.

3. gem://gem.core.EventSchemaServiceInterface/*/* represents implementation instances of interface gem.core.EventSchemaServiceInterface without concerning itself with a particular implementation.

4. gem://*/*/* represents all interface implementation instances in a DEBS.

Based on the interface addressing system mentioned above, the framework can deliver an event to its target interface (or interfaces) in four different ways: (1) Instance Specific, (2) Implementation Specific, (3) Interface Specific, and (4) Definition Specific.

Instance Specific An event with a target interface address having a format of case 1 can be uniquely delivered to an individual interface implementation instance. The framework provides this semantic to support directly addressed interactions among interface instances. The instance specific interface addressing assumes that the sender interface knowns its target interface instance, and wants its event to be seen and reacted to by its specified target interface instance. It does not want to be

bothered by unexpected responses from other unknown interface instances. Usually, this is used together with other combinations. For example, in our framework implementation, once an interface is loaded into the framework, its hosting reactive component interface will use this method to deliver a Start event to its hosted interface instances to notify them of the start of their life cycle. Similarly, the Stop event is also sent in this way to its hosted interfaces when the hosting component wants to end their life cycle.

Implementation Specific An event with a target interface address having a format of case 2 can be delivered to all interface implementation instances of a specific interface implementation. The implementation specific interface addressing semantic is helpful in cases where the sender interface wants to constrain the event to those interfaces which are implemented in a particular way (e.g., must be implemented by a software company with a good reputation). In our framework, if an interface wants to find a particular event schema and also knows there are instances of *scxml* implementation available, it can send a schema request to those SchemaServiceInterface instances implemented in *scxml* under the address of gem://gem.core.EventSchemaServiceInterface/scxml/*.

Interface Specific An event with a target interface address having a format of case 3 can be delivered to all interface implementation instances of a specific interface. The interface specific interface addressing semantic is similar to case 2 with no specific preference over a particular implementation. For instance, if an interface wants to find a particular event schema and does not care which potential implementation answers this question, it can send a schema request to SchemaServiceInterface under the address of gem://gem.core.EventSchemaServiceInterface/*/*.

Definition Specific An event with a target interface address having a format of case 4 (i.e., gem://*/*/*) can be delivered to all interface implementation instances which are interested in this event. The definition specific interface addressing is the standard event delivery semantic in a pure DEBS. In this way, an interface knows nothing about the target interfaces. A sender interface simply tells the framework the event bus to which it wants this event published, and the framework is in charge of forwarding this event to all those interface instances which are currently listening to that event bus and are interested in the event.

Finally, we want to emphasize some conditions that all target interface instances

must satisfy before they can actually receive an event e of event schema E sent by sender interface I. Specifically, the target interface instances must:

1. Be addressed in one of the four ways mentioned above.
2. Be listening to the event bus to which the event e is delivered.
3. Be interested in the event e (i.e., have event schema E defined as one of its input event schemas. If an expected sender interface J is also defined, J must match the actual sender interface I by name).

Event Delivery Mode: Normal Event, Request or Response

The framework provides three fundamental function calls, sendEvent, sendRequest and sendResponse, to allow an event to be dispatched as a normal event, request or response.

The first one (i.e., sending as normal events) is the standard way of delivering events in a pure DEBS. For the convenience of DEBS development, we also provide externalPublish(Event) and internalPublish(Event) to allow interfaces to easily publish events to their external or internal event bus, respectively. However, we expect that, in many cases, RPC-oriented semantics are also very helpful, so our framework also has built-in request-response RPC support (i.e., sendRequest and sendResponse) for system function implementation, experiments and for convenient system development.

Through the use of Event, Request and Response meta-events (Figure 4.3), these event delivery modes are implemented in the FrameworkCoreService and can be used to develop event-based applications either as a pure DEBS or as a hybrid. In particular, if all interfaces constrain their event sending only to externalPublish(Event) or internalPublish(Event), we will have a pure DEBS. Otherwise, we have an event/RPC-hybrid system.

As normal event In this way, event sending is performed asynchronously. The semantic of sending an event as a normal event is that the sender interface instance continues its work and does not expect any response from other interfaces. The FrameworkCoreService provides a method sendEvent(ID busID, GEMAddress from, GEMAddress to, Event event) to send an event instance to its target interface or interfaces *via* event bus busID.

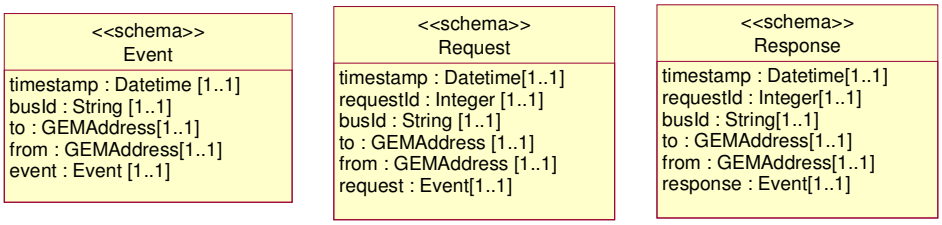


Figure 4.3: Meta Event Schemas

Listing 4.1: Event Schema Definition – gem.service.core.event

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.core.event">
    <Attr Name="timestamp"      Type="DATETIME" Format="yyyy-MM-
        dd'T'HH:mm:ss.SSSZ" />
      <Attr Name="busId"          Type="String"/>
      <Attr Name="to"             Type="GEMAddress"/>
      <Attr Name="from"           Type="GEMAddress"/>
    <Attr Name="event"          Type="Event"/>
  </ComplexType>

  <EventSchemaName>gem.service.core.event</EventSchemaName>
</EventSchema>
```

Every event dispatched as a normal event is packed as a payload in an event instance of a special gem.service.core.event event schema. The definition is shown in code List 4.1. An event of gem.service.core.event consists of one timestamp to indicate the time the event was sent, one busID to indicate the event bus to which this event instance was sent (the busID is also the event bus from which the event instance is received), one to and one from GEMAddress to address the sender and receiver component interface, one event instance which is the actual event instance that was sent out as a payload. Indeed, the event payload is generic and reflective. Any event instance can be carried as a payload in a gem.service.core.event event.

As request In this way, sending is performed synchronously. The semantic of sending an event as a request is that the sender interface **does** expect response events. The sender interface will be blocked until a response event comes back or

times out. If the event is addressed to a group of interfaces, the first response will wake up the blocked sender interface instance and the rest of the responses will be discarded. In the event no response comes back, the sender will be awakened by a time out and an exception will be thrown to indicate the failure. The method Event sendRequest(ID busID, GEMAddress from, GEMAddress to, Event event) can be used to send a request event addressed to its target interface or interfaces.

Listing 4.2: Event Schema Definition – gem.service.core.request

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.core.request">
    <Attr Name="timestamp"      Type="DATETIME" Format="yyyy-MM-
        dd'T'HH:mm:ss.SSSZ" />
      <Attr Name="requestId"      Type="Integer"/>
      <Attr Name="busId"          Type="String"/>
      <Attr Name="to"             Type="GEMAddress"/>
      <Attr Name="from"           Type="GEMAddress"/>
    <Attr Name="request"        Type="Event"/>
  </ComplexType>

  <EventSchemaName>gem.service.core.request</EventSchemaName>
</EventSchema>
```

Similar to sending a normal event, every event sent out as a request is packed as a payload in an event instance of a special gem.service.core.request event schema. The definition is shown in List 4.2. An event of gem.service.core.request consists of one timestamp to indicate the time the event was sent, one requestId to uniquely identify a request with respect to its sender interface, one busID to indicate the event bus to which the event instance is sent (the busID is also the event bus from which the event instance is received), one to and one from GEMAddress to represent the sender and receiver component interface, one request of event instance which is the actual event instance that was sent out as a payload. Indeed, the request payload is generic and reflective. Any event instance can be carried as a payload in a gem.service.core.request event.

As response Once a request event is received, the receiver interface instance returns a response event to its sender. The method sendResponse(ID busID, GEMAd-

	as event	as request	as response
Instance Specific	Yes	Yes	Yes
Implementation Specific	Yes	Yes	No
Interface Specific	Yes	Yes	No
Definition Specific	Yes	Yes	No

Table 4.1: Combinations of Supported Event Deliveries

dress from, GEMAddress to, Event event, long origReqID) allows a response event to be sent back to its original sender interface with specified origReqID.

Listing 4.3: Event Schema Definition – gem.service.core.response

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.core.response">
      <Attr Name="requestId"      Type="Integer"/>
      <Attr Name="busId"          Type="String"/>
    <Attr Name="timestamp"      Type="DATETIME" Format="yyyy-MM-
      dd'T'HH:mm:ss.SSSZ" />
      <Attr Name="to"             Type="GEMAddress"/>
      <Attr Name="from"           Type="GEMAddress"/>
    <Attr Name="response"       Type="Event"/>
  </ComplexType>

  <EventSchemaName>gem.service.core.response</EventSchemaName>
</EventSchema>
```

Like a normal event or request, every event sent out as a response is packed as a payload in an event instance of a special gem.service.core.response event schema. The definition is shown in code List 4.3. An event of gem.service.core.response consists of one timestamp to indicate the time the event was sent, one requestId that matches the corresponding request ID to which this event is a response, one busID to indicate the event bus to which this event instance was sent (the busID is also the event bus from which this event instance is received), one to and one from GEMAddress to represent the sender and receiver component interface, one response of the event instance which is the actual event instance sent out as a payload. The response payload is generic and reflective: any event instance can be carried as a payload in a gem.service.core.response event.

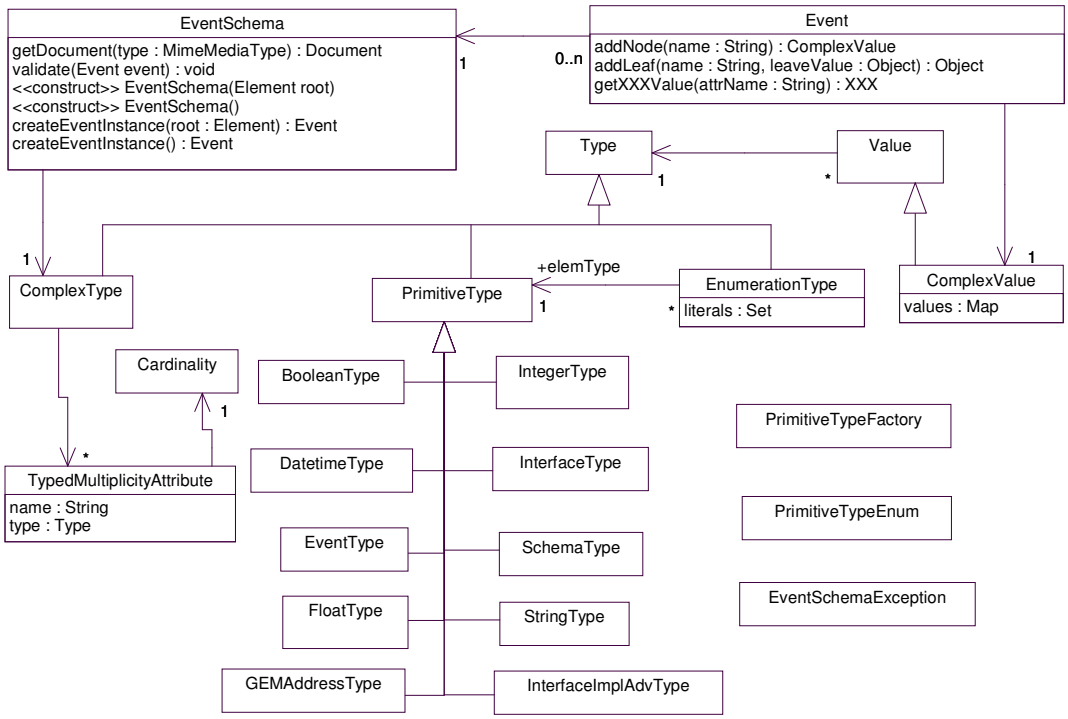


Figure 4.4: Framework: Schema and Event Model

The twelve possible event delivery combinations are summarized in Table 4.1. As shown in the table, the framework allows an event to be delivered as a normal event or request in all four interface addressing styles. A response can only be sent to a particular interface instance. Again, recall that if we always send normal events in the definition specific interface addressing format, we will get a pure DEBS; otherwise, we will have an event/RPC-hybrid system.

4.2.2 Event Schemas and Events

In GEM, everything is event related, such as schema creation, event instance creation, event deliver and event process. It is highly desirable in such a system, that different event schemas be created, published and removed dynamically, and their event instances be created and validated accordingly. Also, events should be delivered in a neutral, platform independent way. Therefore, the design goals of this part of the framework are flexibility and heterogeneity. The part of the framework which implements the generic schema and event metamodel is depicted in Figure 4.4.

Each Eventschema has exactly one associated ComplexType. A ComplexType has

multiple TypedMultiplicityAttribute, each of which defines the name and type of a particular attribute and associates with one Cardinality to describe the number of times an attribute can occur in an Event instance.

The Type of an attribute can be either ComplexType, PrimitiveType or EnumerationType. The current framework supports ten most-often-used primitive types. The first five are normal primitive types: BooleanType, IntegerType, DatetimeType, FloatType and StringType. The other five are complex types: InterfaceType, EventType, SchemaType, GEMAddressType and InterfaceImplAdvType. They are the most-often-used five complex types in our proposed framework and, thus, are promoted to be the primitive type.

Every EnumerationType is associated with exactly one PrimitiveType representing its element type. Also, the EnumerationType holds a set of literal values of all valid values in this type. All actual values have the same type as defined in association elemType.

Listing 4.4: Event Schema Sample - SchemaRegister

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.event.SchemaRegister">
    <Attr Name="schema"    Type="Schema" minOccur="1" maxOccur="*
       "/>
  </ComplexType>

  <EventSchemaName>gem.service.event.SchemaRegister</
     EventSchemaName>
</EventSchema>
```

The built-in event schema service interface in our framework defines an input event schema called SchemaRegister. Every SchemaRegister event can contain multiple event schema definitions and can be sent to an event schema service interface to register these event schemas in the framework. The SchemaRegister event schema can be defined in XML as shown in Listing 4.4.

Listing 4.5: Event Schema Sample - SensorData

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="GPSPosition">
```

```
    <Attr Name="longitude" Type="FLOAT" />
    <Attr Name="latitude" Type="FLOAT" />
  </ComplexType>
  <ComplexType Name="gemdemo.sensor.SensorData">
    <Attr Name="location" Type="GPSPosition" />
    <Attr Name="temperature" Type="FLOAT" minOccur="1" maxOccur=
        "5" />
  </ComplexType>

  <EventSchemaName>gemdemo.sensor.SensorData</EventSchemaName>
</EventSchema>
```

Listing 4.5 shows a more complex example of an event schema definition. The example shown in Listing 4.5 is the XML definition of the gemdemo.sensor.SensorData event schema as defined in Diagram 3.4(a) in the event schema section of the DEBS metamodel reported in Section 3.2.1. From its definition, we know that every SensorData event has exactly one location and one to five temperature data. Each location, in turn, must have exactly one longitude and one latitude value.

Once an event schema is defined in XML, it can be loaded into GEM, thereby creating event schemas. As shown in Figure 4.4, every Event has exactly one associated EventSchema and one ComplexValue which has a map representing all pairs of attribute names and their values in Event. An attribute can itself be a ComplexValue if its attribute type is of ComplexType. In this way, the event value forms a hierarchical structure.

In GEM, event instances can only be created from their associated event schemas to ensure the validity of all event instances. The event method Object addLeaf(String name, Object leaveValue) adds a leaf value (i.e., an attribute whose type is of PrimitiveType or EnumerationType specified by name). For example, assume we have loaded the SchemeRegister event schema and two other event schemas (Start and Stop). The code snippet below 4.6 shows how to create a SchemaRegister event from its event schema and how to populate the event with event schemas Start and Stop. The populated SchemeRegister event can later be dispatched to the event schema service to register these two event schemas.

Listing 4.6: Creating SchemaRegister Event Instance

```
EventSchema start, stop;
// ... loading start, stop event schema
Event schemaRegisterSchemaEvent = schemaRegisterSchema.
```

```
    createEventInstance();
// set a leaf attribute value
schemaRegisterSchemaEvent.addLeaf("schema", start);
schemaRegisterSchemaEvent.addLeaf("schema", stop);

// validity check
schemaRegisterSchemaEvent.validate();
```

Another event method ComplexValue addNode(String name) is used to add a node attribute (i.e., an attribute whose type is of ComplexType); this method returns a ComplexValue which can then be used to populate the leave attributes or to add other node attributes recursively. An example of how to populate a SensorData (4.5) is illustrated in Listing 4.7.

Listing 4.7: Creating SensorData Event Instance

```
Event sensorDataEvent = sensorDataSchema.createEventInstance();

// add 3 temperature data (we can have 1 to 5 temperature)
sensorDataEvent.addLeaf("temperature", new Float(37.0));
sensorDataEvent.addLeaf("temperature", new Float(35.0));
sensorDataEvent.addLeaf("temperature", new Float(36.5));

// add a node attribute "location"
ComplexValue locationValue = sensorDataEvent.addNode("location")
    ;

// populate "location" by adding its leaving values
locationValue.addLeaf("longitude", new Double(34.4));
locationValue.addLeaf("latitude", new Double(89.3));

// validity check
sensorDataEvent.validate();
```

During the value assignment, type checking and upper bound cardinality is enforced by the framework. Once populated, the lower bound of attributes in an event can be checked by invoking method void validate() defined in Event. The method validate() will recursively go through each attribute and check whether values assigned to an attribute match its type and whether an attribute occurrence satisfies its attribute's cardinality. In case of error, an EventSchemaException which specifies the error reasons will be thrown out.

Once populated, events can be dispatched to their target interfaces. As mentioned earlier, GEM transports all events in XML form. Events are first marshaled at the sender side from objects into XML, and are then unmarshaled back from XML into objects at the receiver side. Listing 4.8 illustrates a marshaled SchemaRegister event which was populated as shown in 4.6.

Listing 4.8: Event Sample - A SchemaRegister Event

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Event>
<Event Schema="gem.service.event.SchemaRegister">
  <schema>
    <EventSchema>
      <EventSchemaName>gem.interface.commmand.generic.start</
          EventSchemaName>
    </EventSchema>
  </schema>
  <schema>
    <EventSchema>
      <EventSchemaName>gem.interface.commmand.generic.stop</
          EventSchemaName>
    </EventSchema>
  </schema>
</Event>
```

4.2.3 Component Interface Definition

Recall that in our metamodel, every component can extend another interface and can encapsulate other interfaces. The interface behavior is described in a statechart variant. GEM imposes the following constraints on a component interface. In particular, an interface can: (1) extend at most one other interface, (2) encapsulate more than one other interface, (3) associate with exactly one behavior StateMachine defined in SCXML.

A component interface is treated and shared as a resource. A component interface can be defined and published as a ComponentInterfaceAdvertisement in XML form. For example, the corresponding interface definition of the Sensor interface 5.2 is shown in Listing 4.9.

The Sensor interface extends gem.core.GenericComponentInterface and will receive input gemdemo.sensor.IntervalChg events from any interface and will publish

gemdemo.sensor.SensorData events to its external event bus. The interface behavior definition is between the behavior tags and will be discussed later in Section 4.2.4.

Listing 4.9: Sensor Interface Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
  <InterfaceName> gemdemo.sensor.Sensor </InterfaceName>
  <Extend> gem.core.GenericComponentInterface </Extend>
  <InputEventSchema>
    <SchemaName> gemdemo.sensor.IntervalChg </SchemaName>
    <SenderInterfaceName>anyone</SenderInterfaceName>
  </InputEventSchema>

  <OutputEventSchema>
    <SchemaName>gemdemo.sensor.SensorData</SchemaName>
  </OutputEventSchema>
  <Behavior>
    <scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
        http://gem/CORE" initialstate="loaded" version="1.0">
       <state id="error" final="true">
       <onentry>
         <gem:do action="error"/>
       </onentry>
       <transition event="gem.interface.commmand.generic.stop">
         <target next="stopped"/>
       </transition>
       </state>

     <state id="loaded">
       <onentry>
         <gem:do action="load"/>
       </onentry>
       <transition event="gem.interface.commmand.generic.start"
          >
         <target next="started"/>
         <gem:do action="start"/>
       </transition>
     </state>
```

```
      <state id="started">
        <transition event="internal::timer">
          <target next="started"/>
          <gem:do action="reportData" outputEventName="gemdemo.
              sensor.SensorData"/>
        </transition>
        <transition event="gemdemo.sensor.IntervalChg">
          <target next="started"/>
          <gem:do action="changeInterval"/>
        </transition>
        <transition event="gem.interface.commmand.generic.stop">
          <target next="stopped"/>
        </transition>
      </state>

      <state final="true" id="stopped">
        <onentry>
          <gem:do action="stop"/>
        </onentry>
      </state>

    </scxml>
  </Behavior>
</ComponentInterfaceAdvertisement>
```

4.2.4 Component Interface Behavior Definition

The behavior of a component interface is described using SCXML with customized semantics. SCXML stands for State Chart eXtensible Markup Language and is a W3C standard providing a generic state-machine based execution environment based on CCXML (Call Control XML) and Harel State Tables. SCXML syntax is self-explanatory and, therefore, omitted here. Readers are referred to the SCXML website [39] for specification details.

The default event handling semantic in SCXML is to silently discard an event if it is not expected at the current state. In our framework, this semantic is modified so that any unexpected event will cause the interface behavior state machine to transit to a special error state indicating the detection of an invalid event sequence during the interface event exchanging. The error state is reserved in every interface

behavior statechart, and upon entering this state, the interface implementation instance can only be stopped with a Stop event.

Recall the Sensor interface behavior as defined in Figure 5.2. The Sensor interface extends the ComponentGenericInterface, and its behavior is described as follows: after being loaded into the framework, a component interface implementation is in loaded state. Its life cycle starts when it receives a Start event, transiting to state started. Once started, the interface will report temperature data every time an internal timer goes off. Also, at this state, the Sensor interface can change temperature collecting intervals when it receives an IntervalChg event. A Stop event will then bring the state machine to the final state stopped. At any state, once unexpected events occur, the interface will transit to the error state from which it can only be stopped.

The corresponding SCXML definition of the above Sensor interface behavior is embedded between the behavior tags as shown in Listing 4.9. The current framework represents every interface behavior as a complete SCXML, no matter if it is extending or not. It would be ideal if when an interface behavior extends another one, the behavior extension could be represented as delta behavior changes in terms of states, transitions, and action changes. However, such optimization is outside the scope of this thesis and is left for future work. Finally, when an interface XML definition is created, it can be loaded into the framework and published to interface services.

4.2.5 Interface Implementation Definition

A component interface can have multiple types of implementation (e.g., for different programming languages such as Java, C++, or for different hardware platforms such as Unix, PC, cellphone). An implementation is represented as ComponentInterfaceImplAdvertisement, which is an adaptation of JXTA's ModuleImplAdvertisement.

Listing 4.10 shows a particular SCXML implementation of a built-in interface called gem.core.GenericComponentInterface. The definition says that this particular interface implementation requires JDK 1.4.1, JXTA V2.0 and Commons SCXML V0.6. The gem.GenericComponentInterfaceSCXMLImpl is the Java class realizing this implementation and can be found at http://gem/core/gem.jar.

Listing 4.10: SCXML Implementation of GenericComponentInterface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:MIA>
```

```
<jxta:MIA type="ComponentInterfaceImplAdv" xmlns:jxta="http://
    jxta.org">
  <interfaceName>
    gem.core.GenericComponentInterface
  </interfaceName>
  <ImplName>scxml</ImplName>
  <Behavior>Commons SCXML V0.6</Behavior>
  <Comp>
    <Efmt> JDK1.4.1 </Efmt>
    <Bind> V2.0 Ref Impl </Bind>
  </Comp>
  <Code> gem.GenericComponentInterfaceSCXMLImpl </Code>
  <PURI> http://gem/core/gem.jar </PURI>
</jxta:MIA>
```

An interface implementation advertisement is automatically generated by the framework and should not be tempered with. An interface implementation provider can create the above implementation advertisement by invoking method ComponentInterfaceImplAdvertisement buildInterfaceImplAdv(ComponentInterface componentInterface, String code, String uri) in FrameworkCoreServer. Once created, the implementation advertisement can be used to populate an InterfaceImplAdvRegister event, which can be sent to InterfaceServiceInterface to register this implementation. Designers can then invoke the AbstractComponentInterfaceImpl resolveInterfaceImpl(ComponentInterface anInterface) method in FrameworkCoreService to search and load a compatible implementation for the specified component interface. These steps are shown in the code snippet identified in Listing 4.11 below.

Listing 4.11: Building Publishing and Loading Component Interface Implementation

```
// build interface implementation advertisement
ComponentInterfaceImplAdvertisement genericInterfaceImplAdv =
  buildInterfaceImplAdv(genericCompInterface,
     GenericComponentInterfaceSCXMLImpl.class.getName(), "http
     ://gem/core/gem.jar");

// build an InterfaceImplAdvRegister event to publish this
   interface implementation advertisement
EventSchema interfaceImplRegisterSchema = coreService.
   findEventSchemaByName("gem.service.interface.
   InterfaceImplAdvRegister");
```

```
Event interfaceImplRegisterSchemaEvent =
    interfaceImplRegisterSchema.createEventInstance();
interfaceImplRegisterSchemaEvent.addLeaf("interfaceImplAdv",
    genericInterfaceImplAdv);

// send register event to register this interface implementation
coreService.sendEvent(demoComponent.getAssignedEventBusID(),
    myGEMAddress, FrameworkCoreService.
    colleagueInterfaceServiceAddress,
    interfaceImplRegisterSchemaEvent);

// later, implementation is resolved and loaded into the
    framework
GenericComponentInterfaceSCXMLImpl
    aGenericComponentInterfaceSCXMLImpl =
   resolveInterfaceImpl(genericCompInterface);

....
```

4.2.6 Interface Implementation Instance

An interface implementation instance represents a loaded interface implementation in GEM. An implementation instance will have an associated state machine executing engine populated using its interface behavior definition. The instance will have an assigned external EventBus and, if encapsulating others, it can request an internal EventBus as well. Also, the instance will get a UUID to uniquely identify itself. After set-up is complete, the interface instance is ready to react to incoming events from the external and internal buses, and this interface instance can be addressed using any one of the four interface addressing formats as mentioned earlier in Section 4.2.1.

GEM currently supports an interface to be easily implemented as an *scxml* implementation. Specifically, developers implement an interface by implementing the event actions and guard condition tests associated with transitions as defined in the interface definition. For each event action *xyz*, developers should implement a method *void doActionXyz(Event e)*, where e is the trigger event of the transition. For each guard condition test *pqr*, developers should implement a method *boolean*

testPqr(Event e), where e is the trigger event of the transition. These implemented actions and tests will be used by a state machine executing engine at runtime.

GEM integrates Apache Commons SCXML as the default built-in state machine executing engine [38]. Every time an interface implementation is loaded into the framework, its interface behavior definition is used to populate a state machine executing engine, which is then associated with the implementation instance. GEM will make sure that the implementation instance will receive only those events it wants and will only send events as specified. Once an event is received, the implementation instance can then ask its associated executing engine to handle the event. The engine will, in turn, rely on its associated implementation instance to provide the actual guard condition tests and event handling actions.

4.3 Framework Built-in Component Interfaces

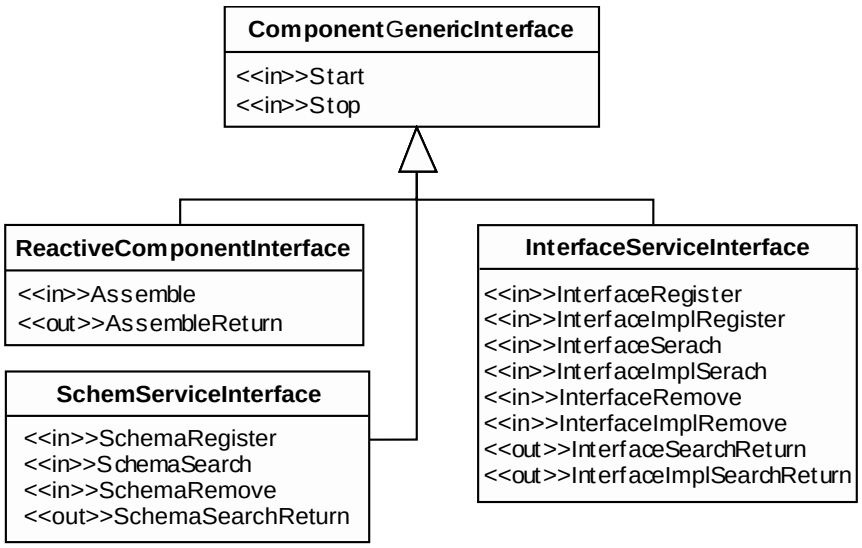
This section explains the rest of the framework, which is recursively designed following our proposed interface-based development process in an event/RPC-hybrid style. In particular, the rest of the framework is decomposed into four built-in component interfaces as shown in Figure 4.5(a). The event schemas used by these four interfaces are shown in Figure 4.5(b). In this section, we will employ notation introduced in section 3.5 wherever space-constraint is not a problem.

4.3.1 ComponentGenericInterface

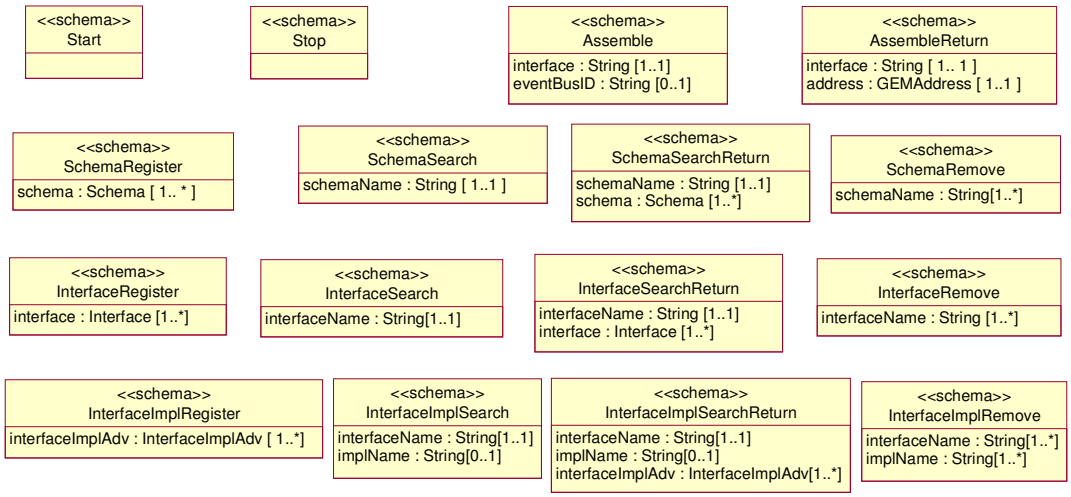
Similar to the Object class in the JAVA programming language, this interface resides at the very top of the interface-tree. It defines the basic behavior all interfaces in the framework must preserve during their life cycle, and it is the ancestor of all other component interfaces.

As shown in Figure 4.6, this ComponentGenericInterface defines the common interface behaviors as follows:

1. An interface has a special error state. At any state, when it receives an unexpected event, the interface will transit to its error state and will execute error action. Once at error state, the interface can only accept a Stop event, which stops the interface life cycle. At error state, all events except for Stop will be treated as unexpected events and will, again, cause the interface to transit back to its error state.



(a) Component Interfaces



(b) Interface Event Schemas

Figure 4.5: Framework Built-in Interfaces and Event Schemas

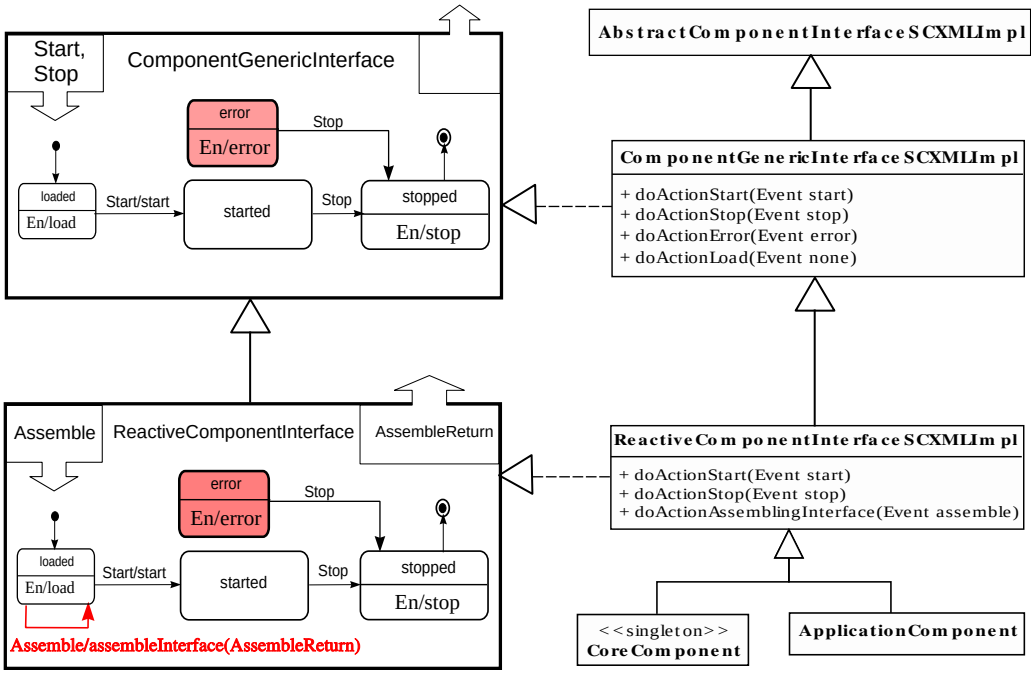


Figure 4.6: Component Generic Interface and Reactive Component Interface

2. An interface's life cycle starts once it is loaded into the framework (at loaded state). An interface can run action load to perform initialization. Once loaded, and after it receives a Start event, it executes action start and then becomes stated. It is expected that this stated state is expanded by sub-interfaces to reflect application specific behaviors.

3. At started, when it receives a Stop event, the interface stops running and transits to state stopped.

4. The stopped state is a final state and formally represents the end of the life cycle. When it enters this state, the interface will call action stop to perform finalization tasks before it quits.

The ComponentGenericInterface has a default SCXML implementation – ComponentGenericInterfaceSCXMLImpl. The method doActionLoad, doActionStart, doActionStop and doActionError defines the actual semantics of the corresponding action load, start, stop and error in ComponentGenericInterface. The ComponentGenericInterfaceSCXMLImpl is expected to be extended by other interface SCXML implementations to achieve behavioral extension.

4.3.2 ReactiveComponentInterface

In our framework, a reactive component is treated as a generic container which can assemble other component interfaces and manage their life cycles. The ReactiveComponentInterface defines the behavior of such a reactive component. In addition to Start and Stop, this interface can also react to an Assemble event and will send out an AssembleReturn event. It extends ComponentGenericInterface behaviors with the following delta changes as shown in Figure 4.6:

- At loaded, when an Assemble event is received, which contains an interface name to be assembled and an optional event bus ID, this ReactiveComponentInterface will invoke action AssemblingInterface to load the interface implementation into the framework. The life cycle of the loaded interface will be managed by this ReactiveComponentInterface. Once an implementation of the specified interface is loaded, this interface will send out an AssembleReturn event to return the GEMAddress of the newly assembled interface instance.
- At loaded, when notified to start, the start action in ReactiveComponentInterface will notify all its assembled interface instances to start.
- Similarly, at started, when notified to stop, the stop action in ReactiveComponentInterface will notify all its assembled interface instances to stop.

For the ReactiveComponentInterface interface, GEM provides an SCXML implementation named ReactiveComponentInterfaceSCXMLImpl, from which both CoreComponent and ApplicationComponent are derived. On each framework host, there is only one CoreComponent and there can be multiple ApplicationComponents as needed. By default, a CoreComponent assembles two core interfaces, SchemaServiceInterface and InterfaceServiceInterface, both of which will be discussed later.

4.3.3 SchemaServiceInterface

This built-in interface models GEM's event schema registry and provides event schema services *via* events. In particular, other interfaces can send events to this interface to request services such as registering or unregistering event schemas, searching existing event schemas. In GEM, an event schema must be registered before its events can be created and used. After an event schema has been removed, all its ongoing events will be treated as illegal and not understandable until this event schema is registered again.

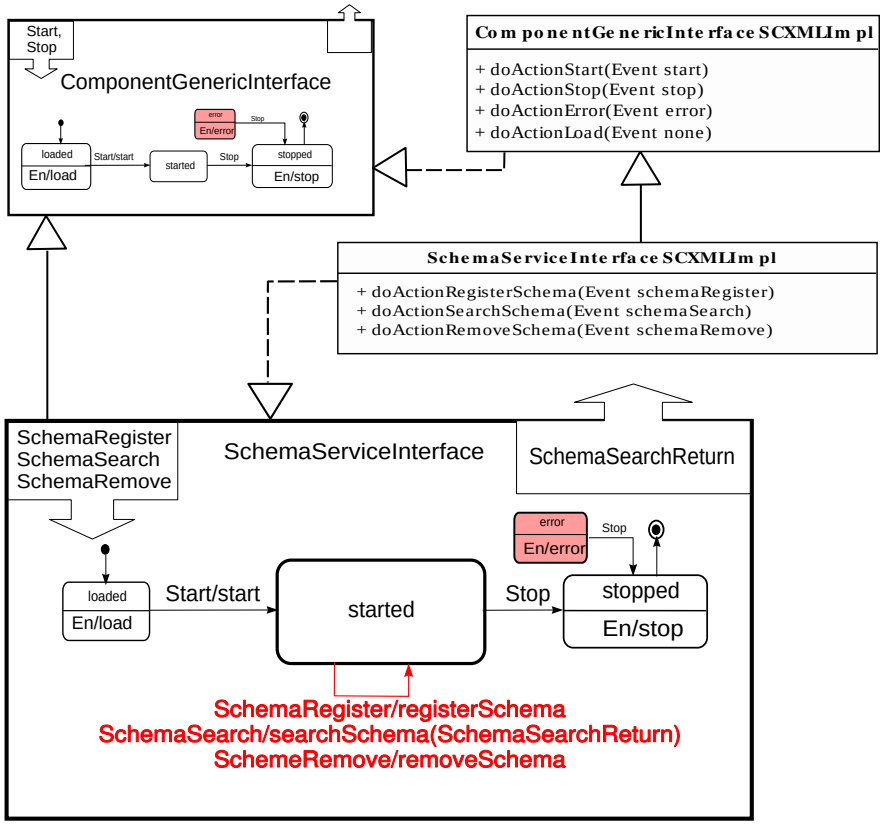


Figure 4.7: Built-in Interface: Event Schema Service

The SchemaServiceInterface is pre-loaded in the core component during framework bootstrapping. Though an event can be directly addressed to an individual SchemaServiceInterface instance, by default, an event is sent as a group to all instances of all types of implementation of this interface using interface specific addressing 4.2.1 (i.e., gem://gem.core.EventSchemaServiceInterface/*/*). A single SchemaRegister event, for example, will cause its embedded event schemas to be registered in all available SchemaServiceInterface implementation instances. Similarly, a single SchemaSearch event will be answered by all available SchemaServiceInterface implementation instances. The requester will usually receive the answer from the nearest SchemaServiceInterface instance with the answer. The remaining answers are silently discarded.

The SchemaServiceInterface extends ComponentGenericInterface with delta changes as shown in Figure 4.7. In addition to Start and Stop, this interface will receive SchemaRegister, SchemaSearch and SchemaRemove events, and will publish SchemaSearchReturn events. The event schemas are defined in Figure 4.5(b). The started state is extended with the following additional transitions:

1. A SchemaRegister event contains multiple event schemas to be registered. When a SchemaServiceInterface receives the event, it will call action registerSchema to register the event schemas embedded in the event.

2. A SchemaSearch event contains an event schema name to be searched. When a SchemaServiceInterface receives the event, it will call action searchSchema to perform the search and will return its search result in a SchemaSearchReturn event to the sender interface. The interface instance will not answer the search if it cannot find the specified event schema.

3. A SchemaRemove event contains multiple event schema names to be removed from the registry. When a SchemaServiceInterface receives the event, it will call action removeSchema to remove the event schemas from its registry

4.3.4 InterfaceServiceInterface

This built-in interface models GEM's registry for component interfaces and their implementations. It provides services to other interfaces *via* event exchanging. In particular, other interfaces can send events to this interface to request services such as registering or unregistering interface, search interface, registering or unregistering interface implementation, and search interface implementation.

In GEM, to use an interface, its associated event schemas, interface definition and implementation must be registered. If a required event schema is unexpectedly removed from the framework, the interface will not be able to receive or send corresponding events and, therefore, may not be able to function properly. The current framework does not check the dependency between event schemas and interfaces during event schema removal. This feature is left for future work.

The InterfaceServiceInterface is pre-loaded in the core component during framework bootstrapping. Similar to SchemaServiceInterface, an interface usually sends an event to all InterfaceServiceInterface instances using interface specific addressing 4.2.1. In this way, a single InterfaceRegister event will cause embedded interfaces to be registered in all available InterfaceServiceInterface implementation instances.

The InterfaceServiceInterface extends ComponentGenericInterface, and the delta changes are shown in Figure 4.8. In addition to Start and Stop, this interface introduces six more input event schemas and two more output event schemas. These event schemas are defined in Figure 4.5(b). The started state is extended with the following additional transitions:

1. An InterfaceRegister event contains at least one interface definition to be registered. When an InterfaceServiceInterface receives the event, it will call action registerInterface to register embedded interfaces in its registry.

2. An InterfaceImplRegister event contains at least one interface implementation definition to be registered. When an InterfaceServiceInterface receives the event, it will call action registerInterface to register embedded interface implementation definitions in its registry. The interface implementation registration fails if its implemented interface definition is not found.

3. An InterfaceRemove event contains at least one interface name to be removed. When an InterfaceServiceInterface receives the event, it will call action removeInterface to remove the specified interfaces from its registry.

4. An InterfaceImplRemove event contains at least one pair of interfaces and its implementation names to be removed. When an InterfaceServiceInterface receives the event, it will call action removeImplInterface to remove from its registry the specified interface implementation if the interface name and implementation name match the ones specified in the event.

5. An InterfaceSearch event contains one interface name to be searched. When an InterfaceServiceInterface receives the event, it will call action searchInterface to

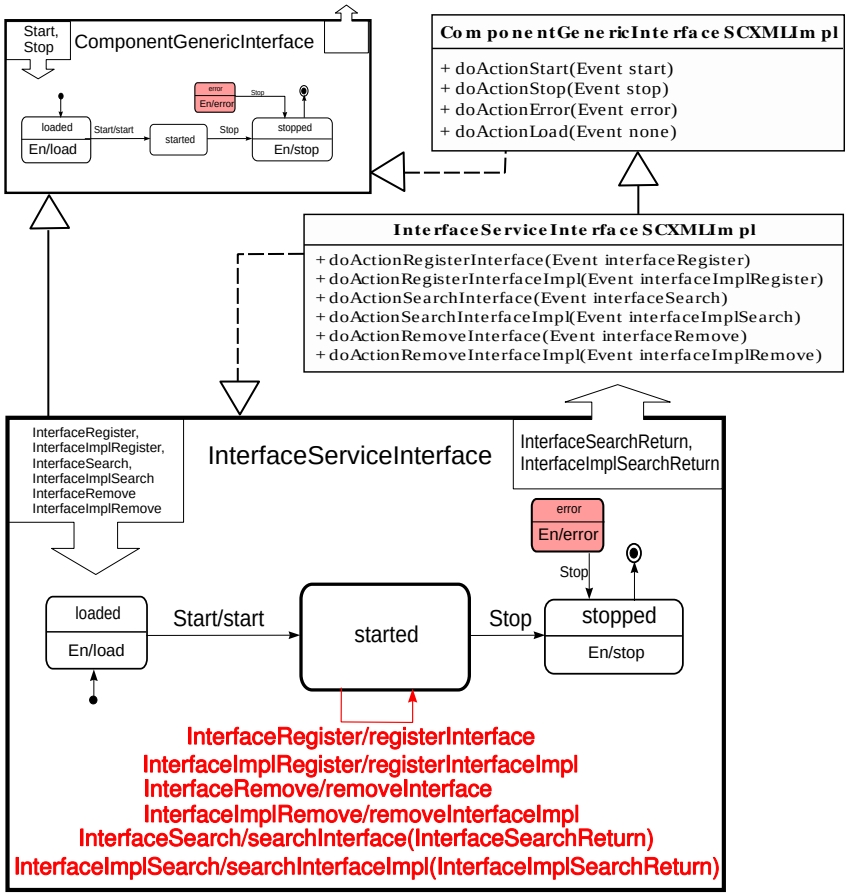


Figure 4.8: Built-in Interface: Interface Service

search the specified interface in its registry. If the specified interface is found, this InterfaceServiceInterface interface will send out an InterfaceSearchReturn event to the original sender interface.

6. An InterfaceImplSearch event contains one interface name and one optional implementation name to be searched. When an InterfaceServiceInterface receives the event, it will call action searchImplInterface to search the specified interface implementation in its registry. If the implementation name is not specified in the event, all types of implementation for the given interface are returned; otherwise, only those implementation definitions whose name matches the specified name are returned. The InterfaceServiceInterface will send out an InterfaceImplSearchReturn event to the original sender interface if the specified implementations are found.

The framework provides a default SCXML implementation InterfaceServiceInterfaceSCXMLImpl which implements the required actions: doActionRegisterInterface, doActionRegisterInterfaceImpl, doActionRemoveInterface, doActionRemoveInterfaceImpl, doActionSearchInterface and doActionSearchInterfaceImpl.

Chapter 5

Case Studies

In this chapter, we illustrate our proposed interface-based DEBS development process *via* two case studies – *Temperature Sensor System* and *e-Promotion System*. By constraining our interface implementation to only use *externalPublish(Event)* for event publishing (i.e., an event is always published as a normal event and can be received by all interfaces that are interested in it), we have specifically designed these two applications in a pure DEBS style.

5.1 Proposed Development Process

With the proposed DEBS metamodel and its supporting framework implementation, we can now build DEBS applications with behavior-enhanced interfaces. The metamodel enables us to model DEBS applications as its instances; the framework implementation provides an environment where DEBS applications can be designed, implemented and deployed. In particular, a DEBS application can be developed in the following steps:

1. decomposing a DEBS application by using interfaces, compositions and event schemas.

2. implementing interfaces by providing event actions and guard condition tests.

3. instantiating a DEBS application by assembling the required interface implementation and deploying to the supporting framework.

5.2 Temperature Sensor System

The *Temperature Sensor System* is a simple DEBS system collecting temperature data from its sensors. In this system, there is one information center and multiple temperature sensors. Each sensor reports temperature data to the information center at certain intervals. The information center collects the temperature data from sensors and can also command sensors to change the interval between their two data reports.

In the rest of this section, we first discuss the event schemas design of the system. Then, we explain the component interfaces of the system in detail. The complete schema definitions and component interface definitions can be found in Listing B.1 and Listing B.2.

5.2.1 Event Schemas

The *Temperature Sensor System* is simple and only needs two event schemas: SensorData and IntervalChange. The event schema design is shown in Figure 5.1.

SensorData This event schema represents the temperature data report published by a Sensor interface. It has a temperature attribute of float type and has an attribute location which is typed as a complex type called GPSPosition. Any value of SensorData can have one to five temperature values and must have one GPSPosition value. The GPSPosition complex type has two attributes, longitude and latitude, both of which are of type float. Any value of GPSPosition must have one longitude value and one latitude value.

IntervalChange This event schema represents an interval change published by an information center. An IntervalChange event has one interval of Integer type indicating the new time interval between two data reports expected by the information center interface. In particular, an event with interval set to 1000 means an expected interval of 1000ms.

5.2.2 Component Interfaces

We decompose the *Sensor* system into two interfaces, Sensor and InforCenter. The design of these two interfaces is shown in Figure 5.2.

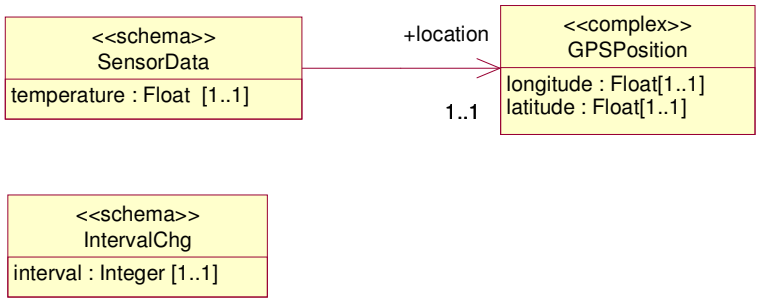


Figure 5.1: Sensor System Event Schemas

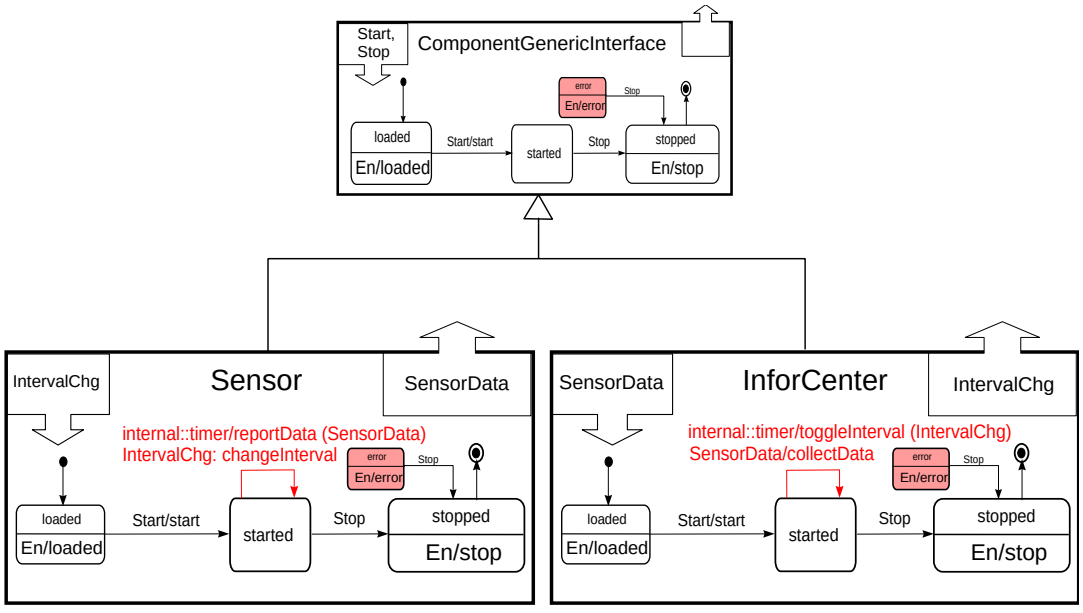


Figure 5.2: Sensor System Component Interfaces

Sensor This interface models the temperature sensor which will publish its temperature data report from time to time. The Sensor interface extends ComponentGenericInterface by adding the following transitions to started state:

1. At started state, when the internal timer goes off, the Sensor interface will invoke reportData action to publish its temperature report *via* a SensorData event which can pack at most five temperature data.

2. At started state, the sensor will change its report interval by invoking action changeInterval when it receives an event of IntervalChg which carries the new interval.

InforCenter This interface models the information center which will collect the temperature report and will toggle the report interval from time to time. The InforCenter interface also extends the built-in ComponentGenericInterface and will receive SensorData and publish IntervalChange events. The delta changes are shown as below:

1. At started state, when the internal timer goes off, the InforCenter will call toggleInterval action to publish a new temperature report interval *via* an IntervalChg event.

2. At started state, when a SensorData event has been received, the InforCenter will record this report by invoking collectData action.

5.2.3 Component Interface Implementation

We chose to implement the above component interfaces in SCXML implementation which specifies that each action *xyz* in a component interface will have its corresponding action function *doActionXyz(Event e)* in its SCXML implementation. The function name is the action name with the first letter capitalized and preceded with *doAction* and the event *e* is the trigger event that causes the action *xyz* to be invoked.

We use SensorSCXMLImpl and InforCenterSCXMLImpl to implement Sensor interface and InforCenter interface, respectively. Both interface implementations specialize the built-in ComponentGenericInterfaceSCXMLImpl which provides a default internal timer and the default logic of being loaded, stopped, and error handling,

which can be overridden by its subclasses, if necessary. In our simple system, SensorSCXMLImpl and InforCenterSCXMLImpl use only externalPublish(Event) to publish events, and do not override the default actions. The interface implementation design is shown in Figure 5.3.

SensorSCXMLImpl The action reportData in Sensor interface is implemented *via* the function doActionReportData(Event timer) in SensorSCXMLImpl where timer is the internal timer event. The action changeInterval in Sensor interface is implemented *via* the function doActionChangeInterval(Event intervalChg) in the SensorSCXMLImpl where intervalChg is the IntervalChg event received by Sensor interface.

InforCenterSCXMLImpl The action collectData in InforCenter interface is implemented *via* the function doActionCollectData(Event sensorData) in InforCenterSCXMLImpl where sensorData is the SensorData event received by InforCenter interface. The action toggleInterval in InforCenter interface is implemented *via* the function doActionToggleInterval(Event timer) in InforCenterSCXMLImpl where the timer is the internal timer event. Also, the doActionToggleInterval(Event timer) publishes an IntervalChg event as defined in InforCenter interface.

5.2.4 Component Interface Deployment

Figure 5.4 shows an instance of the Temperature Sensor System which consists of one InforCenter interface implementation instance and multiple Sensor interface implementation instances communicating *via* their external event bus. We do not specify the hosting reactive component of these interface implementation instances. We assume the case where each interface implementation instance is deployed to its own ApplicationComponent which is remotely located to each other.

5.3 e-Promotion System

The *e-Promotion System* is a location-aware DEBS application developed with our proposed behavior-enhanced interfaces. As depicted in Figure 5.5, this system consists of a shopping mall and its shoppers:

- the shopping mall can count the shoppers currently in the mall

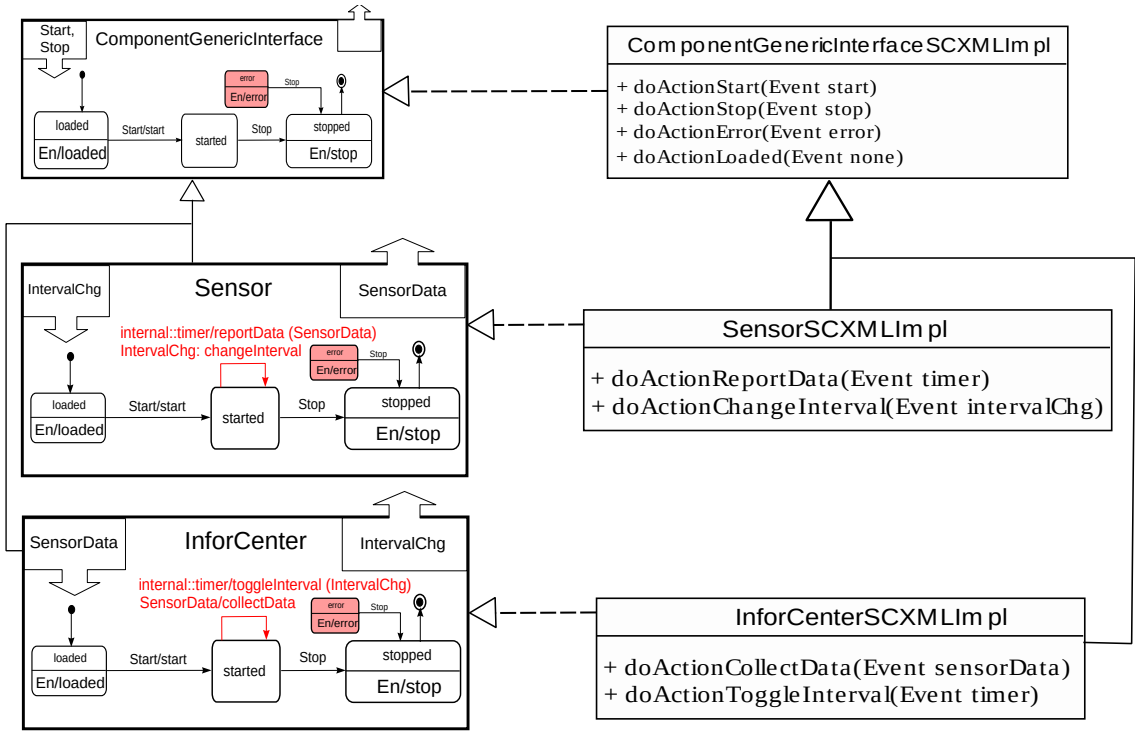


Figure 5.3: Sensor System Interface SCXML Implementation Diagram

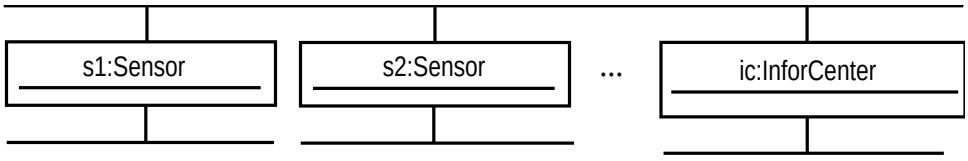


Figure 5.4: Sensor System Interface Instance Diagram

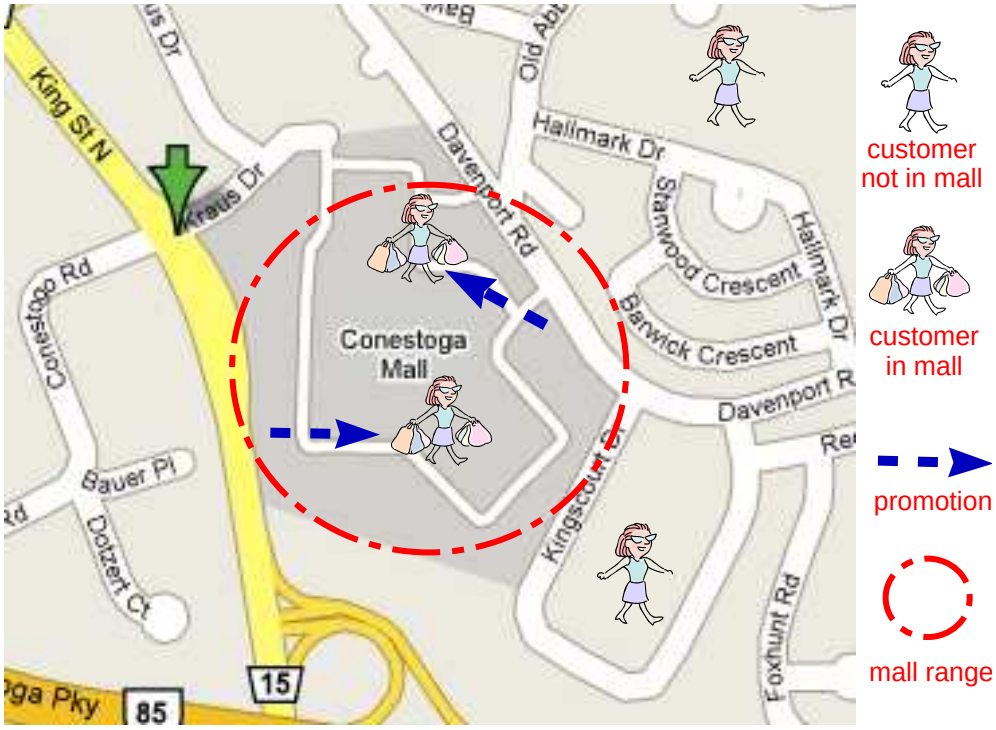


Figure 5.5: e-Promotion System Overview

- the shopping mall can send its promotion information to customers currently in the mall
- shoppers will get promotion information when they are in the mall
- shoppers will not get promotion information when they are not in the mall

5.3.1 System Design Overview

Figure 5.6 illustrates the interface-based DEBS design of the *e-Promotion System*, which is decomposed into three reactive component interfaces: PromotionBroadcaster, PromotionReceiver and PersonalGPS, where

- PromotionBroadcaster interface represents a promotion broadcaster and can be installed in a shopping mall
- PromotionReceiver interface represents a promotion receiver and can be deployed in a shopper's mobile device (e.g., cellphone, blackberry)

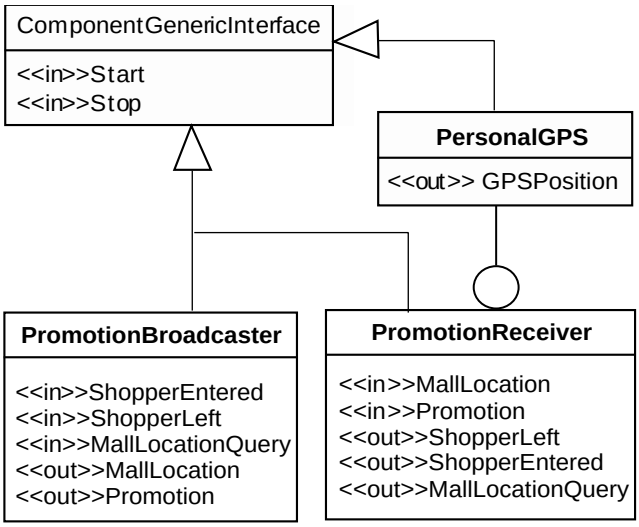


Figure 5.6: e-Promotion System Interfaces Overview

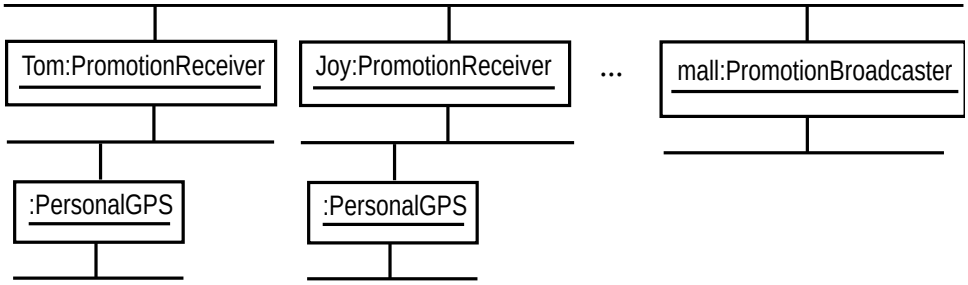


Figure 5.7: e-Promotion System Instance Diagram

- PersonalGPS interface represents the device that provides the current GPS position. The GPS device can be the same as the shopper's mobile device or it can be a third-party mobile device.

The instance diagram Figure 5.7 shows an example of an e-Promotion System. In this diagram, instances of PromotionReceiver (possessed by shoppers) and the instance of PromotionBroadcaster (owned by a mall) communicate *via* their external event bus. Each PromotionReceiver obtains its current location from its encapsulated PeronslGPS instance (represents a GPS device).

In the rest of this section, we discuss the event schema design of the system, then explain the component interfaces of the system in detail. The complete schema definitions and component interface definitions can be found in Listing C.1 and Listing C.2.

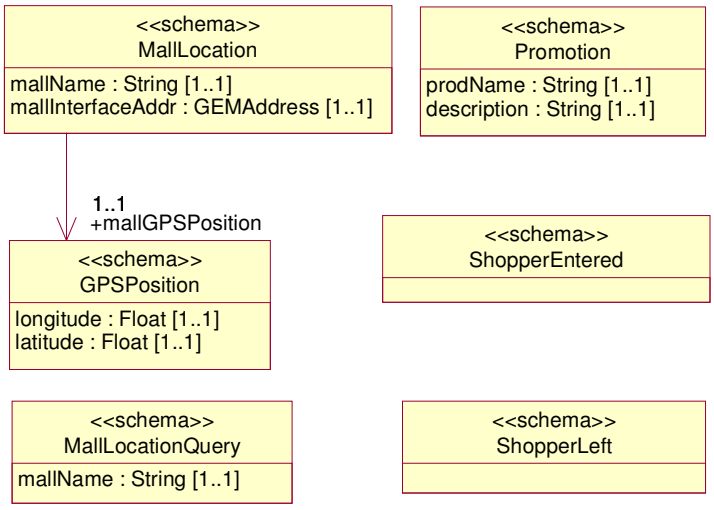


Figure 5.8: e-Promotion System Event Schemas Diagram

5.3.2 Event Schemas

As shown in Figure 5.8, the *e-Promotion system* uses six event schemas: GPSPosition, MallLocationQuery, MallLocation, Promotion, ShopperEntered and ShopperLeft.

GPSPosition This event schema defines the structure of a GPS position with two attributes, longitude and latitude. Both attributes are of float type and must occur exactly once in every GPSPosition event.

MallLocationQuery This event schema defines the mall location query event published from PromotionReceiver interface. It defines only one attribute: mallName the name of the mall whose GPS position a receiver interface is looking for. Every event instance of this schema must have exactly one mallName.

MallLocation This event schema defines the type of the mall location event published from the PromotionBroadcaster interface. It defines three attributes:

1. mallName is a String and is the name of the mall whose GPS location is represented in this event
2. mallInterfaceAddr is of type GEMAddress and is the PromotionBroadcaster interface address associated with the mall specified by mallName

3. mallGPSPosition is the GPS position of the mall specified in mallName. The mallGPSPosition attribute is a complex type gemdemo.promotion.GPSPosition as defined above.

All attribute must appear exact once in every event instance of this type.

Promotion This event schema defines the type of a promotion event published from the PromotionBroadcaster interface. It defines two attributes: prodName and description. Both attributes are of String type and both must appear exact once in every event instance of this type.

ShopperEntered A ShopperEntered event represents that a shopper has entered the mall and is published by a PromotionReceiver interface. The ShopperEntered schema has no attributes.

ShopperLeft A ShopperLeft event represents that a shopper has left the mall and is published by a PromotionReceiver interface. The ShopperLeft schema has no attributes.

5.3.3 PromotionBroadcaster

The PromotionBroadcaster interface models a promotion broadcaster in a shopping mall. The PromotionBroadcaster extends the ComponentGenericInterface, and will receive ShopperEntered, ShopperLeft and MallLocationQuery events and will publish MallLocation and Promotion events.

Figure 5.9 depicts the detailed interface definition. The delta interface behavior of PromotionBroadcaster is described as below:

1. At started, when the PromotionBroadcaster receives a MallLocationQuery event, it will call action answerLocation which will publish its predefined location *via* a MallLocation event.

2. At started, when the PromotionBroadcaster receives a ShopperEntered event, it will call action increaseCount to increase the count of shoppers in the mall.

3. At started, when the PromotionBroadcaster receives a ShopperLeft event, it will call action decreaseCount to decrease the count of shoppers in the mall.

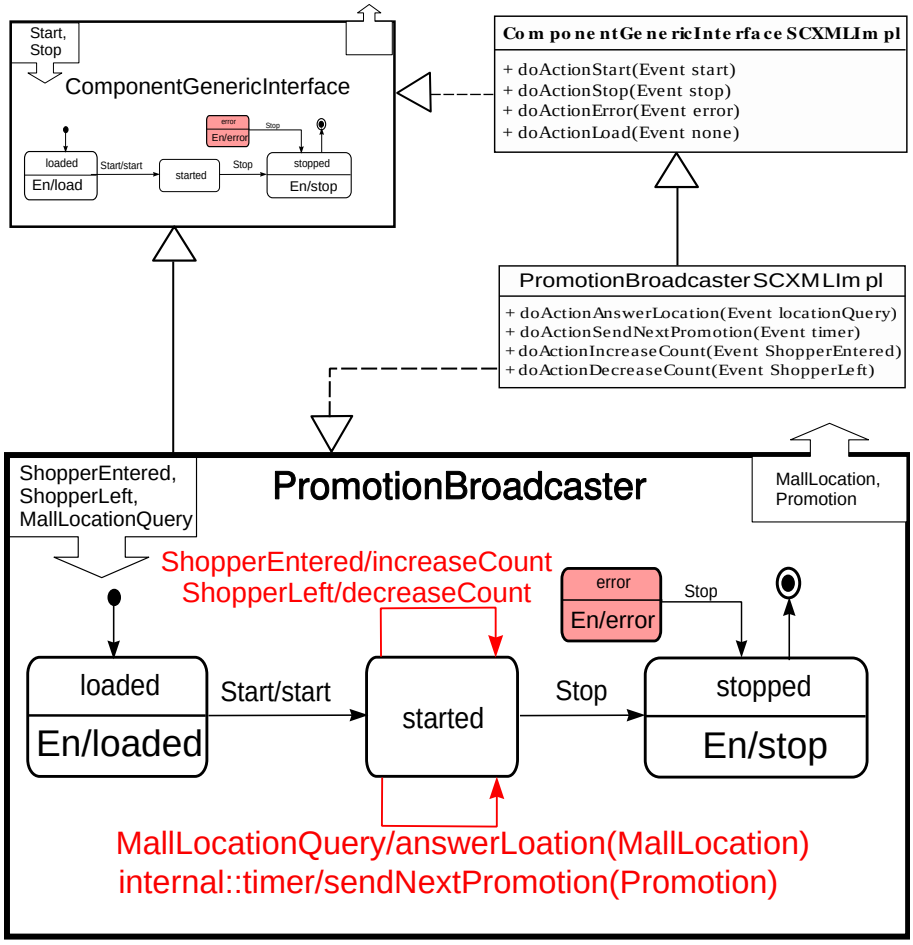


Figure 5.9: e-Promotion System: PromotionBroadcaster Interface

4. At started, when an internal timer goes off, the PromotionBroadcaster will call action sendNextPromotion which will publish a Promotion event provided there are shoppers in the mall.

The PromotionBroadcaster interface is implemented by PromotionBroadcasterSCXMLImpl, an SCXML implementation that extends ComponentGenericInterfaceSCXMLImpl. The PromotionBroadcasterSCXMLImpl implements four required actions: doActionAnswerLocation, doActionSendNextPromotion, doActionIncreaseCount and doActionDecreaseCount.

5.3.4 PromotionReceiver

The PromotionReceiver interface represents a promotion receiver which can be deployed in a shopper's mobile device (e.g., cellphone, blackberry). As shown in Figure 5.6, this interface extends ComponentGenericInterface. Also, the PromotionReceiver interface encapsulates a PersonalGPS interface from which it can obtain a GPSPosition event representing its current location.

Figure 5.10 illustrates the interface definition details. The PromotionReceiver interface will receive MallLocation and Promotion events and will publish MallLocationQuery, ShopperEntered and ShopperLeft events. The PromotionReceiver expands the started state into a composite one with the following delta behavior changes:

1. At state started, a PromotionReceiver automatically enters mallLocating state. Each time it enters this state, it will call action locateMall to publish a MallLocationQuery. If it receives a GPSPosition (published by its encapsulated PersonalGPS interface) or a Promotion event, it will transit back to this state. When it receives a MallLocation event, it will save this mall location by invoking action saveMallLocation and then transits to state mallLocated.

2. At state mallLocated, when it receives a GPSPosition event, if the condition test !inRange returns true (i.e., this receiver is not in the mall), the receiver will transit to state outside. If it receives a GPSPosition event and the condition test inRange returns true (i.e., a customer enters the mall from outside), the receiver will first execute action enterMall and then transit to state inside. The action enterMall publishes a ShopperEntered event. It will stay in this state if it receives MallLocation or Promotion events.

3. At state inside, if a receiver receives a Promotion event, it will call showPromotion to display this promotion information. If it recieves MallLocation events,

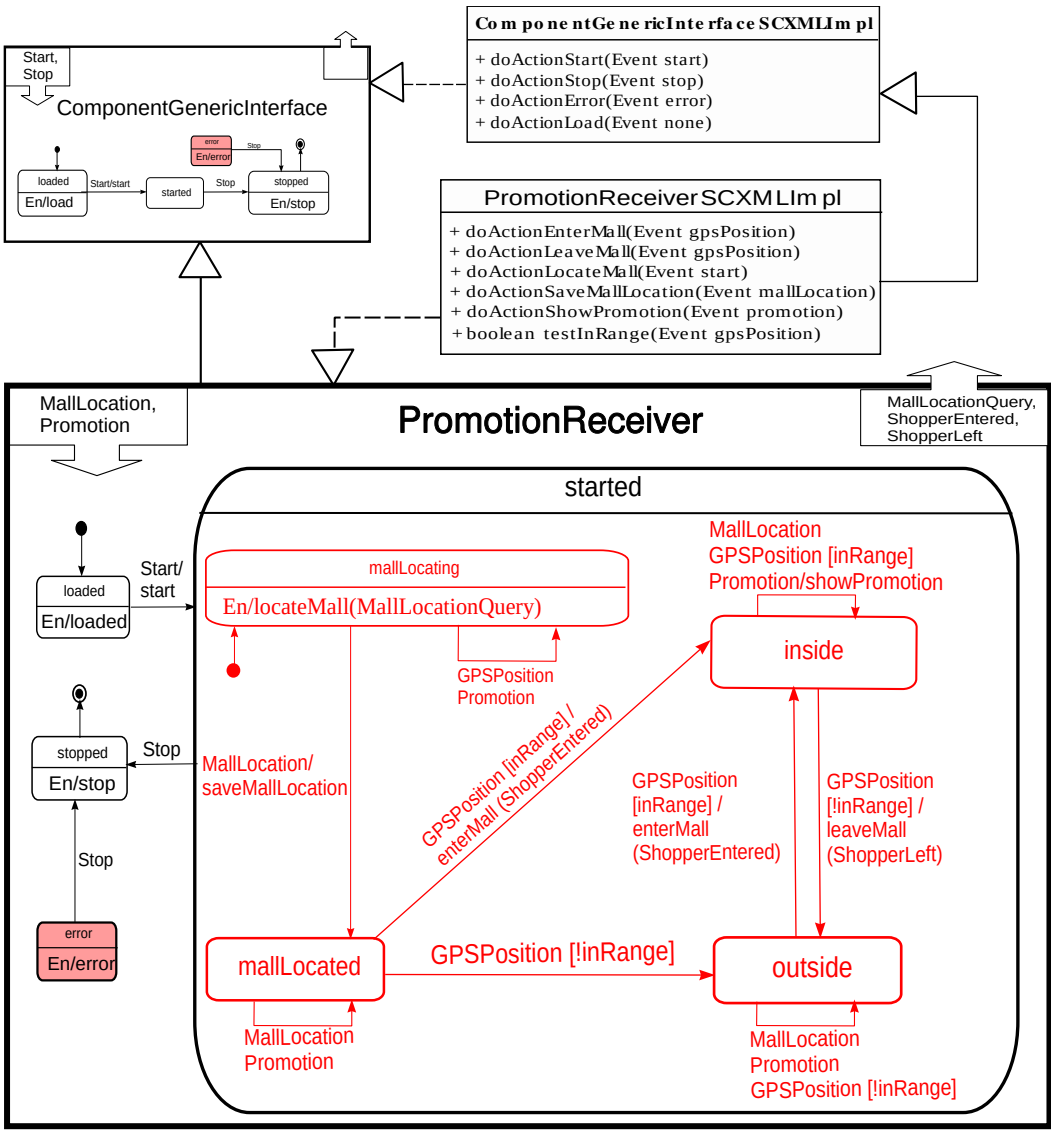


Figure 5.10: e-Promotion System: PromotionReceiver Interface

it will stay in this state. If the receiver gets a GPSPosition and the condition test inRange returns true (i.e., the shopper is still in the mall), it stays in inside. If it receives a GPSPosition event and it detects that it is not in the mall (i.e., !inRange returns true), it knows that the shopper has left the mall. It will first invoke leaveMall and then transit to state outside. The action leaveMall will publish a ShopperLeft event.

4. At state outside, if the receiver receives a GPSPosition event and this receiver is still outside the mall, it will do nothing and just remain in outside. If it receives MallLocation or Promotion events, if will also stay in this state. If the receiver receives a GPSPosition event and the condition test inRange is true (i.e., this receiver moves from outside to inside the mall), the receiver will first call enterMall and then transit to state inside. The action enterMall will publish a ShopperEntered event.

The PromotionReceiver interface is implemented by PromotionReceiverSCXMLImpl, an SCXML implementation that extends ComponentGenericInterfaceSCXMLImpl. The PromotionReceiverSCXMLImpl implements five actions and one condition test: doActionEnterMall, doActionLeaveMall, doActionLocateMall, doActionSaveMallLocation, doActionShowPromotion and testInRange.

5.3.5 PersonalGPS

In this e-Promotion system, the device that provides the current GPS position is abstracted as a PersonalGPS interface, no matter whether the device is in the same mobile device or in a separate one. The PersonalGPS interface is particularly designed to be like a service for its colleague interfaces: every second, it publishes a GPSPosition event to its external event bus so that all other interfaces which are also attached to this event bus can receive this location information if they want.

The PersonalGPS interface extends ComponentGenericInterface and will send out GPSPosition events. The behavior of this interface is very simple: once started, when it receives an internal::timer event, it will execute action reportLocation which will populate a GPSPosition event with the current location and publish the event.

The PersonalGPS interface is designed to be encapsulated by the PromotionReceiver interface. Recall in Section 3.4.2, we mention that an encapsulating interface instance sets up a private environment that is shared only by itself and its encapsulated interface instances. In this particular case, the PromotionReceiver interface

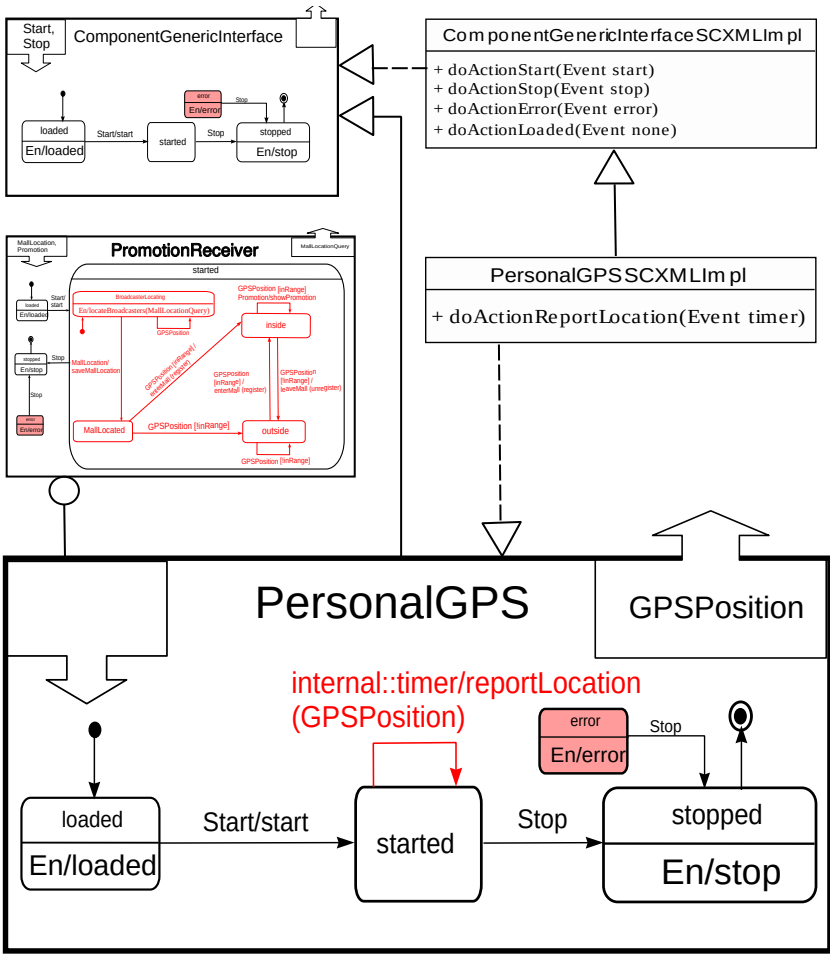


Figure 5.11: e-Promotion System: PersonalGPS Interface

will hide the PersonalGPS interface by assigning its internal event bus to the PersonalGPS's external event bus. Therefore, all GPSLocation events from an encapsulated PersonalGPS interface instance will only be published in a private environment and will only be received by its encapsulating PromotionReceiver interface instance.

The PersonalGPS interface is implemented by PersonalGPSSCXMLImpl, an SCXML implementation that extends ComponentGenericInterfaceSCXMLImpl. The method doActionReportLocation implements the action reportLocation in interface, which will publish a GPSPosition event as defined.

5.4 Experience Summary

We see a promising result of modularization using behavior-enhanced interfaces. As illustrated in the interfaces design, event handling is not less-organized action anymore. Indeed, event receiving, processing and publishing are now precisely modeled in an interface. In addition, we found this behavior-enhanced interface also serves as a very good documentation mechanism and unambiguously conveys the desired interface behaviors.

The DEBS design is simplified with the help of GEM. With GEM, all we have to do is to define event schemas, define interfaces, implement interface and deploy. In fact, for the first two, we only need to construct XML files. There is a little coding in interface implementation which requires us to implement the actions and guard condition tests as defined in the interface behavior. The coding involved in the deployment is very predictable. We just need to launch the framework, deploy the interface implementation and then we can tell the framework to run the application. GEM handles the rest of the work such as event dispatching and interface behavior enforcement.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Current DEBS development is not modular and is still an informal process poorly supported by current software engineering methodologies [7, 28]. In this thesis, we advocate that by designing a new DEBS metamodel with extended behavioral interfaces and high-level structure mechanisms, we can (1) define an interface-based modular approach to model and design DEBS applications, (2) implement a prototype framework on a P2P network that provides built-in support to our proposed interface-based DEBS development, and (3) provide case studies illustrating the interface-based development process and the applicability of our proposed approach.

We design a new DEBS metamodel with emphases on (1) event and event schema as first-class constructs, (2) behavior-enhanced interface modeled as input event schemas, output event schemas and a finite state machine specifying interface behavior in terms of event receiving, processing and publishing, and (3) interface composition mechanisms which define how complex interfaces can be composed from simple interfaces hierarchically.

We provide GEM, a prototype framework on top of the JXTA network that supports developers to build their DEBS applications based on the proposed interface-based DEBS design. GEM is designed for DEBS domain and provides built-in support to interface-based DEBS development. Also, with flexible event delivery mechanisms, GEM becomes generic and flexible and it allows a DEBS application to be designed either as a pure DEBS or as an event/RPC-hybrid system.

Finally, we illustrate our proposed interface-based development process *via* two case studies. Our first-hand experience gained from these case studies shows a

promising result of applying our proposed interface-based approach in designing DEBS applications. Overall, the DEBS design is better modularized with behavior-enhanced interfaces, and the design implementation can be facilitated with GEM, which also guards the system behavior at runtime.

6.2 Future Work

Fine-Grained Event Access Control The current proposed generic DEBS metamodel focuses on essential elements required in a DEBS in terms of its static structure and runtime behavior description without concerning the event access control. In fact, by associating events with component interfaces, this metamodel effectively provides an implicit, simple event access control mechanism, i.e., whether an interface can publish or consume an event is statically specified in its definition. However, in real application systems, we believe it desirable to have a more fine-grained event access control mechanism. A generic property-based access control would specify that the ability of an interface accessing an event depends not only on its static interface definition, but also on a particular property either possessed by the interface statically or by its implementation instance at runtime. For instance, we can specify a runtime property such that an event E can only be accessed by interfaces whose runtime instances are located at node N (or in city C), or must possess role R.

Delta Behavior While extending another interface, the input and output events of an extending interface can be defined incrementally in its interface definition, however, its behavior description is described as a complete SCXML. It would be desirable to design an approach to model the interface behavior changes as statechart delta changes.

Session-aware State Machine Engine The current framework integrates an open source Java SCXML engine (Commons SCXML project from Apache) with customized semantics to implement interface behaviors. The current state machine engine does not have session support. For example, given state machines M_1 and M_2 (both have multiple states), one M_1 state machine instance cannot interact with multiple M_2 state machine instances concurrently. Though we can design in such a way that a M_1 state machine instance is created every time a M_2 state machine is connected, such an approach is prohibitively expensive. It is desirable to enhance

the state machine engine to have built-in session support by using mechanisms such as context switching.

References

[1] Tim Anderson. Components for .NET. `http://www.dnjonline.com/articles/dotnet/apr02_dotnetcomponents.asp`, 2002. 1, 8, 9

[2] A. Buchmann, C. Bornhövd, M. Cilia, L. Fiege, F. Gärtner, C. Liebig, M. Meixner, and G. Mühl. Dream: Distributed reliable event-based application management. In M. Levene and A. Poulovassilis, editors, *Web dynamics: Adapting To Change In Content, Size, Topology And Use*, pages 319–352. Springer-Verlag, Germany, 2004. 4

[3] Cynthia Della Torre Cicalese and Shmuel Rotenstreich. Behavioral Specification of Distributed Software Component Interfaces. *Computer*, 32(7):46–53, 1999. 8

[4] M. Cilia, C. Bornhövd, and A. P. Buchmann. CREAM: An Infrastructure for Distributed, Heterogeneous Event-based Applications. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 482–502. Springer Berlin, 2003. 4

[5] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001 (ESEC/FSE 2001)*, pages 109–120, Vienna, Austria, September 2001. 8

[6] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, Inc., New York, NY, USA, 1999. 3

[7] Pascal Fenkam, Mehdi Jazayeri, and Gerald Reif. On Methodologies for Constructing Correct Event-based Applications. *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, pages 38–43, May 2004. 2, 75

[8] Ludger Fiege. *Visibility in Event-Based Systems*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, April 2005. 11

[9] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering Event-Based Systems with Scopes. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 309–333. Springer-Verlag, London, UK, 2002. 11

[10] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994. 1

[11] Object Management Goup. CORBA 3.0 - OMG IDL Syntax and Semantics chapter. `http://www.omg.org/cgi-bin/doc?formal/02-06-39`, 2002. 1, 3

[12] Object Management Goup. Common Object Request Broker Architecture (CORBA/IIOP). `http://www.omg.org/cgi-bin/doc?formal/04-03-01`, 2004. 1, 2, 3, 7, 9

[13] Object Management Goup. CORBA Component Model, v4.0. `http://www.omg.org/cgi-bin/doc?formal/06-04-01`, 2006. 1, 7

[14] Object Management Goup. Meta Object Facility Core Specification version 2.0. `http://www.omg.org/technology/documents/formal/MOF_Core.htm`, 2006. 10

[15] Object Management Goup. UML 2.1.1 Superstructure Specification. `http://www.omg.org/technology/documents/formal/uml.htm`, 2007. 8, 21

[16] Michael Guppenberger and Burkhard Freitag. Intelligent Creation of Notification Events in Information Systems: Concept, Implementation and Evaluation. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 52–59. ACM, New York, NY, USA, October 2005. 4

[17] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996. 21

[18] JXTA(TM) Community. JXTA Java^{TM} Standard Edition v2.5: Programmers Guide. `https://jxta.dev.java.net/`, September 2007. 11

[19] Chaoying Ma and Jean Bacon. COBEA: A CORBA-based Event Architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 9–9, Santa Fe, New Mexico, April 1998. 2

[20] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. 1

[21] Microsoft Corporation. COM: Component Object Model Technologies. `http://www.microsoft.com/com/default.mspx`, 2007. 3

[22] Sun Microsystems. Java(TM) Message Service Specification Final Release 1.1. `http://java.sun.com/products/jms/docs.html`, 2002. 3, 9

[23] Sun Microsystems. Jini Specification, Version 2.0. `http://java.sun.com/products/jini/`, June 2003. 3, 9

[24] Sun Microsystems. Enterprise JavaBeans 3.0 Final Release. `http://java.sun.com/products/ejb/docs.html`, 2007. 1, 3, 7

[25] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002. 4

[26] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems*. Springer Berlin Heidelberg, 2006. 2, 4

[27] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, New York, USA, February 2001. 10

[28] Peter Pietzuch, Gero Mühl, and Ludger Fiege. Distributed Event-Based Systems: An Emerging Community. *IEEE Distributed Systems Online*, 8(2):2–2, February 2007. 2, 75

[29] Peter R. Pietzuch and Jean Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618. Washington, DC, USA, 2002. 3, 4, 9

[30] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference*, pages 64–84. Springer, Rio de Janeiro, Brazil, June 2003. 4

[31] Simon Roberts and Jon Byous. Distributed Events in Jini Technology. http://java.sun.com/developer/technicalArticles/jini/JiniEvents/, 1999. 3

[32] Szabolcs Rozsnyai, Josef Schiefer, and Alexander 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*, pages 62–70. ACM, New York, NY, USA, 2007. 10

[33] Anthony J.H. Simons. On the Compositional Properties of UML Statechart Diagrams. In *Proceedings of the Third Workshop on Rigorous Object-Oriented Methods (ROOM2002)*. The British Computer Society (BCS), York, UK, January 2000. 21

[34] SpringSource. Spring framework. http://www.springframework.org/, 2007. 3

[35] Sun Microsystems. Java Remote Method Invocation (RMI). http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, 2007. 3

[36] Sun Microsystems Inc. JXTA v2.0 Protocols Specification. https://jxta-spec.dev.java.net/JXTAProtocols.pdf, 2007. 3, 11

[37] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Co., New York, NY, USA, 1998. 1

[38] The Apache Software Foundation. Commons SCXML - An Open-source Java SCXML Engine. http://commons.apache.org/scxml/, 2007. 50

[39] World Wide Web Consortium (W3C). State Chart XML (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft. http://www.w3.org/TR/scxml/, February 2007. 8, 46

[40] Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, 1993. 2

[41] Stefanos Zachariadis. *Adapting Mobile Systems Using Logical Mobility Primitives*. PhD thesis, University of London, May 2005. 10

[42] Stefanos Zachariadis and Cecilia Mascolo. The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems. *IEEE Transactions on Software Engineering*, 32(11):910–927, 2006. 10

Appendix A

Framework Event Schemas for Built-in Interfaces

A.1 ComponentGenericInterface

Listing A.1: ComponentGenericInterface Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <EventSchemaName>gem.interface.commmand.generic.Start</
     EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <EventSchemaName>gem.interface.commmand.generic.Stop</
     EventSchemaName>
</EventSchema>
```

A.2 ReactiveComponentInterface

Listing A.2: ReactiveComponentInterface Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.encapsulate.Assemble">
    <Attr Name="interface"          Type="String" />
    <Attr Name="eventBusID"          Type="String" minOccur="0"
        maxOccur="1" />
  </ComplexType>

  <EventSchemaName>gem.encapsulate.Assemble</EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.encapsulate.AssembleReturn">
    <Attr Name="interface"        Type="String" />
    <Attr Name="address"          Type="GEMAddress" />
  </ComplexType>

  <EventSchemaName>gem.encapsulate.AssembleReturn</
      EventSchemaName>
</EventSchema>
```

A.3 SchemaServiceInterface

Listing A.3: SchemaServiceInterface Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.event.SchemaRegister">
    <Attr Name="schema"    Type="Schema" minOccur="1" maxOccur="*
        " />
  </ComplexType>

  <EventSchemaName>gem.service.event.SchemaRegister</
      EventSchemaName>
</EventSchema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.event.SchemaRemove">
    <Attr Name="schemaName"    Type="String" minOccur="1"
       maxOccur="*" />
  </ComplexType>

  <EventSchemaName>gem.service.event.SchemaRemove</
     EventSchemaName>
</EventSchema>



<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.event.SchemaSearch">
    <Attr Name="schemaName"    Type="String" />
  </ComplexType>

  <EventSchemaName>gem.service.event.SchemaSearch</
     EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.event.SchemaSearchReturn">
    <Attr Name="schemaName"    Type="String" minOccur="1"
       maxOccur="*" />
    <Attr Name="schema"    Type="Schema" minOccur="1" maxOccur="*
       " />
  </ComplexType>

  <EventSchemaName>gem.service.event.SchemaSearchReturn</
     EventSchemaName>
```

```
</EventSchema>
```

A.4 InterfaceServiceInterface

Listing A.4: InterfaceServiceInterface Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceRegister">
    <Attr Name="interface"  Type="Interface" minOccur="0"
       maxOccur="*" />
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceRegister</
     EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceImplRegister
     ">
    <Attr Name="interfaceImplAdv" Type="InterfaceImplAdv"
       minOccur="1" maxOccur="*" />
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceImplRegister</
     EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceSearch">
    <Attr Name="interfaceName"          Type="String" />
  </ComplexType>
```

```
  <EventSchemaName>gem.service.interface.InterfaceSearch</
     EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceImplSearch">
    <Attr Name="interfaceName" Type="String"/>
    <Attr Name="implName" Type="String" minOccur="0" maxOccur="1
       "/>
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceImplSearch</
     EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceSearchReturn
     ">
    <Attr Name="interfaceName"     Type="String"/>
    <Attr Name="interface"         Type="Interface" minOccur="1"
       maxOccur="*"/>
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceSearchReturn</
     EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.
     InterfaceImplSearchReturn">
    <Attr Name="interfaceName" Type="String"/>
```

```
    <Attr Name="implName" Type="String" minOccur="0" maxOccur="1
        "/>
    <Attr Name="interfaceImplAdv" Type="InterfaceImplAdv"
        minOccur="1" maxOccur="*"/>
  </ComplexType>

  <EventSchemaName>gem.service.interface.
      InterfaceImplSearchReturn</EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceRemove">
    <Attr Name="interfaceName" Type="String" minOccur="1"
        maxOccur="*"/>
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceRemove</
      EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gem.service.interface.InterfaceImplRemove">
    <Attr Name="interfaceName" Type="String" minOccur="1"
        maxOccur="*"/>
    <Attr Name="interfaceImplName" Type="String" minOccur="1"
        maxOccur="*"/>
  </ComplexType>

  <EventSchemaName>gem.service.interface.InterfaceImplRemove</
      EventSchemaName>
</EventSchema>
```

Appendix B

Temperature Sensor System

B.1 Event Schemas

Listing B.1: Temperature Sensor System Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="GPSPosition">
    <Attr Name="longitude" Type="FLOAT" />
    <Attr Name="latitude" Type="FLOAT" />
  </ComplexType>
  <ComplexType Name="gemdemo.sensor.SensorData">
    <Attr Name="location" Type="GPSPosition" />
    <Attr Name="temperature" Type="FLOAT" minOccur="1" maxOccur=
        "5" />
  </ComplexType>

  <EventSchemaName>gemdemo.sensor.SensorData</EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gemdemo.sensor.IntervalChg">
    <Attr Name="interval" Type="INTEGER" />
  </ComplexType>
```

```
  <EventSchemaName>gemdemo.sensor.IntervalChg</EventSchemaName>
</EventSchema>
```

B.2 Interfaces

Listing B.2: Sensor Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
  <InterfaceName>
    gemdemo.sensor.Sensor
  </InterfaceName>
  <Extend>
    gem.core.GenericComponentInterface
  </Extend>

  <InputEventSchema>
    <SchemaName>
      gemdemo.sensor.IntervalChg
    </SchemaName>
    <SenderInterfaceName>
      anyone
    </SenderInterfaceName>
  </InputEventSchema>

  <OutputEventSchema>
    <SchemaName>
      gemdemo.sensor.SensorData
    </SchemaName>
    <ImplementationRole>
      anyone
    </ImplementationRole>
    <TTL>
      10000
    </TTL>
  </OutputEventSchema>
  <Behavior>
```

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
    http://gem/CORE" initialstate="loaded" version="1.0">
    <state id="error" final="true">
    <onentry>
      <gem:do action="error"/>
    </onentry>
    <transition event="gem.interface.commmand.generic.Stop">
      <target next="stopped"/>
    </transition>
    </state>

  <state id="loaded">
    <onentry>
      <gem:do action="load"/>
    </onentry>
    <transition event="gem.interface.commmand.generic.Start"
       >
      <target next="started"/>
      <gem:do action="start"/>
    </transition>
  </state>

  <state id="started">
    <transition event="internal::timer">
      <target next="started"/>
      <gem:do action="reportData" outputEventName="gemdemo.
          sensor.SensorData"/>
    </transition>
    <transition event="gemdemo.sensor.IntervalChg">
      <target next="started"/>
      <gem:do action="changeInterval"/>
    </transition>
    <transition event="gem.interface.commmand.generic.Stop">
      <target next="stopped"/>
    </transition>
  </state>

  <state final="true" id="stopped">
    <onentry>
      <gem:do action="stop"/>
```

```
        </onentry>
      </state>

    </scxml>
  </Behavior>
</ComponentInterfaceAdvertisement>
```

Listing B.3: InforCenter Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
  <InterfaceName>
    gemdemo.sensor.InforCenter
  </InterfaceName>
  <Extend>
    gem.core.GenericComponentInterface
  </Extend>

  <InputEventSchema>
    <SchemaName>
      gemdemo.sensor.SensorData
    </SchemaName>
    <SenderInterfaceName>
      anyone
    </SenderInterfaceName>
  </InputEventSchema>

  <OutputEventSchema>
    <SchemaName>
      gemdemo.sensor.IntervalChg
    </SchemaName>
    <ImplementationRole>
      anyone
    </ImplementationRole>
    <TTL>
      10000
    </TTL>
  </OutputEventSchema>
  <Behavior>
```

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
    http://gem/CORE" initialstate="loaded" version="1.0">
    <state id="error" final="true">
    <onentry>
      <gem:do action="error"/>
    </onentry>
    <transition event="gem.interface.commmand.generic.Stop">
      <target next="stopped"/>
    </transition>
    </state>

  <state id="loaded">
    <onentry>
      <gem:do action="load"/>
    </onentry>
    <transition event="gem.interface.commmand.generic.Start"
       >
      <target next="started"/>
      <gem:do action="start"/>
    </transition>
  </state>

  <state id="started">
    <transition event="internal::timer">
      <target next="started"/>
      <gem:do action="toggleInterval" outputEventName="
         gemdemo.sensor.IntervalChg"/>
    </transition>
    <transition event="gemdemo.sensor.SensorData">
      <target next="started"/>
      <gem:do action="collectData"/>
    </transition>
    <transition event="gem.interface.commmand.generic.Stop">
      <target next="stopped"/>
    </transition>
  </state>

  <state final="true" id="stopped">
    <onentry>
      <gem:do action="stop"/>
```

```
        </onentry>
      </state>

    </scxml>
  </Behavior>
</ComponentInterfaceAdvertisement>
```

Appendix C

e-Promotion System

C.1 Event Schemas

Listing C.1: e-Promotion System Event Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gemdemo.promotion.GPSPosition">
    <Attr Name="longitude" Type="FLOAT"/>
    <Attr Name="latitude" Type="FLOAT"/>
  </ComplexType>
  <EventSchemaName>gemdemo.promotion.GPSPosition</
     EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gemdemo.promotion.GPSPosition">
    <Attr Name="longitude" Type="FLOAT"/>
    <Attr Name="latitude" Type="FLOAT"/>
  </ComplexType>

  <ComplexType Name="gemdemo.promotion.MallLocation">
    <Attr Name="mallName" Type="String"/>
    <Attr Name="mallInterfaceAddr" Type="GEMAddress"/>
```

```
    <Attr Name="mallGPSPosition"      Type="gemdemo.promotion.
        GPSPosition"/>
  </ComplexType>

  <EventSchemaName>gemdemo.promotion.MallLocation</
      EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gemdemo.promotion.MallLocationQuery">
    <Attr Name="mallName" Type="String"/>
  </ComplexType>
  <EventSchemaName>gemdemo.promotion.MallLocationQuery</
      EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <ComplexType Name="gemdemo.promotion.Promotion">
    <Attr Name="prodName" Type="String"/>
    <Attr Name="description" Type="String"/>
  </ComplexType>

  <EventSchemaName>gemdemo.promotion.Promotion</EventSchemaName>
</EventSchema>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EventSchema>
<EventSchema>
  <EventSchemaName>gemdemo.promotion.ShopperEntered</
      EventSchemaName>
</EventSchema>


<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE EventSchema>
<EventSchema>
  <EventSchemaName>gemdemo.promotion.ShopperLeft</
     EventSchemaName>
</EventSchema>
```

C.2 Interfaces

Listing C.2: e-Promotion PersonalGPS Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
  <InterfaceName>
    PersonalGPS
  </InterfaceName>
  <Extend>
    gem.core.GenericComponentInterface
  </Extend>
  <OutputEventSchema>
    <SchemaName>
      gemdemo.promotion.GPSPosition
    </SchemaName>
    <ImplementationRole>
      anyone
    </ImplementationRole>
    <TTL>
      10000
    </TTL>
  </OutputEventSchema>
  <Behavior>
    <scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
       http://gem/CORE" initialstate="loaded" version="1.0">
      <state id="error" final="true">
      <onentry>
        <gem:do action="error"/>
      </onentry>
      <transition event="gem.interface.commmand.generic.Stop">
        <target next="stopped"/>
```

```
      </transition>
      </state>

    <state id="loaded">
      <onentry>
        <gem:do action="load"/>
      </onentry>
      <transition event="gem.interface.commmand.generic.Start"
         >
        <target next="started"/>
        <gem:do action="start"/>
      </transition>
    </state>

    <state id="started">
      <transition event="internal::timer">
        <target next="started"/>
        <gem:do action="reportLocation" outputEventName="
           gemdemo.promotion.GPSPosition"/>
      </transition>
      <transition event="gem.interface.commmand.generic.Stop">
        <target next="stopped"/>
      </transition>
    </state>

    <state final="true" id="stopped">
      <onentry>
        <gem:do action="stop"/>
      </onentry>
    </state>

  </scxml>
 </Behavior>
</ComponentInterfaceAdvertisement>
```

Listing C.3: e-Promotion PromotionReceiver Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
 <InterfaceName>
```

```
    PromotionReceiver
</InterfaceName>
<Extend>
    gem.core.GenericComponentInterface
</Extend>
<Encapsulate>
    PersonalGPS
</Encapsulate>

<InputEventSchema>
    <SchemaName>
        gemdemo.promotion.MallLocation
    </SchemaName>
    <SenderInterfaceName>
        anyone
    </SenderInterfaceName>
</InputEventSchema>

<InputEventSchema>
    <SchemaName>
        gemdemo.promotion.Promotion
    </SchemaName>
    <SenderInterfaceName>
        anyone
    </SenderInterfaceName>
</InputEventSchema>

<OutputEventSchema>
    <SchemaName>
        gemdemo.promotion.MallLocationQuery
    </SchemaName>
    <ImplementationRole>
        anyone
    </ImplementationRole>
    <TTL>
        10000
    </TTL>
</OutputEventSchema>

<OutputEventSchema>
```

```
  <SchemaName>
    gemdemo.promotion.ShopperEntered
  </SchemaName>
  <ImplementationRole>
    anyone
  </ImplementationRole>
  <TTL>
    10000
  </TTL>
</OutputEventSchema>

<OutputEventSchema>
  <SchemaName>
    gemdemo.promotion.ShopperLeft
  </SchemaName>
  <ImplementationRole>
    anyone
  </ImplementationRole>
  <TTL>
    10000
  </TTL>
</OutputEventSchema>

<Behavior>
  <scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
      http://gem/CORE" initialstate="loaded" version="1.0">
      <state id="error" final="true">
      <onentry>
        <gem:do action="error"/>
      </onentry>
      <transition event="gem.interface.commmand.generic.Stop">
        <target next="stopped"/>
      </transition>
      </state>

    <state id="loaded">
      <onentry>
        <gem:do action="load"/>
      </onentry>
```

```
      <transition event="gem.interface.commmand.generic.Start"
          >
        <target next="started"/>
        <gem:do action="start"/>
      </transition>
    </state>

    <state id="started">
        <initial>
          <transition>
            <target next="mallLocating"/>
          </transition>
        </initial>

        <state id="mallLocating">
        <onentry>
          <gem:do action="locateMall" outputEventName="gemdemo
              .promotion.MallLocationQuery"/>
        </onentry>
        <transition event="gemdemo.promotion.MallLocation">
          <target next="mallLocated"/>
          <gem:do action="saveMallLocation"/>
        </transition>
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="mallLocating"/>
        </transition>
        <transition event="gemdemo.promotion.Promotion">
          <target next="mallLocating"/>
        </transition>
        </state>

        <state id="mallLocated">
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="inside" cond="gem:inRange"/>
          <gem:do action="enterMall" outputEventName="gemdemo.
              promotion.ShopperEntered"/>
        </transition>
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="outside" cond="gem:!inRange"/>
        </transition>
```

```
        <transition event="gemdemo.promotion.MallLocation">
          <target next="mallLocated"/>
        </transition>
        <transition event="gemdemo.promotion.Promotion">
          <target next="mallLocated"/>
        </transition>
        </state>

        <state id="outside">
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="inside" cond="gem:inRange"/>
          <gem:do action="enterMall" outputEventName="gemdemo.
              promotion.ShopperEntered"/>
        </transition>
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="outside" cond="gem:!inRange"/>
        </transition>
        <transition event="gemdemo.promotion.Promotion">
          <target next="outside"/>
        </transition>
        <transition event="gemdemo.promotion.MallLocation">
          <target next="outside"/>
        </transition>
        </state>

        <state id="inside">
        <transition event="gemdemo.promotion.MallLocation">
          <target next="inside"/>
        </transition>
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="inside" cond="gem:inRange"/>
        </transition>
        <transition event="gemdemo.promotion.Promotion">
          <target next="inside"/>
          <gem:do action="showPromotion"/>
        </transition>
        <transition event="gemdemo.promotion.GPSPosition">
          <target next="outside" cond="gem:!inRange"/>
          <gem:do action="leaveMall" outputEventName="gemdemo.
              promotion.ShopperLeft"/>
```

```
          </transition>
          </state>

        <transition event="gem.interface.commmand.generic.Stop">
          <target next="stopped"/>
        </transition>
      </state>

      <state final="true" id="stopped">
        <onentry>
          <gem:do action="stop"/>
        </onentry>
      </state>

    </scxml>
  </Behavior>
</ComponentInterfaceAdvertisement>
```

Listing C.4: e-Promotion PromotionBroadcaster Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ComponentInterfaceAdvertisement>
<ComponentInterfaceAdvertisement xmlns:jxta="http://jxta.org">
  <InterfaceName>
    PromotionBroadcaster
  </InterfaceName>
  <Extend>
    gem.core.GenericComponentInterface
  </Extend>
  <InputEventSchema>
    <SchemaName>
      gemdemo.promotion.MallLocationQuery
    </SchemaName>
    <SenderInterfaceName>
      anyone
    </SenderInterfaceName>
  </InputEventSchema>

  <InputEventSchema>
    <SchemaName>
      gemdemo.promotion.ShopperEntered
```

```
  </SchemaName>
  <SenderInterfaceName>
    anyone
  </SenderInterfaceName>
</InputEventSchema>

<InputEventSchema>
  <SchemaName>
    gemdemo.promotion.ShopperLeft
  </SchemaName>
  <SenderInterfaceName>
    anyone
  </SenderInterfaceName>
</InputEventSchema>

<OutputEventSchema>
  <SchemaName>
    gemdemo.promotion.MallLocation
  </SchemaName>
  <ImplementationRole>
    anyone
  </ImplementationRole>
  <TTL>
    10000
  </TTL>
</OutputEventSchema>
<OutputEventSchema>
  <SchemaName>
    gemdemo.promotion.Promotion
  </SchemaName>
  <ImplementationRole>
    anyone
  </ImplementationRole>
  <TTL>
    10000
  </TTL>
</OutputEventSchema>
<Behavior>
  <scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:gem="
      http://gem/CORE" initialstate="loaded" version="1.0">
```

```
      <state id="error" final="true">
      <onentry>
        <gem:do action="error"/>
      </onentry>
      <transition event="gem.interface.commmand.generic.Stop">
        <target next="stopped"/>
      </transition>
      </state>

    <state id="loaded">
      <onentry>
        <gem:do action="load"/>
      </onentry>
      <transition event="gem.interface.commmand.generic.Start"
         >
        <target next="started"/>
        <gem:do action="start"/>
      </transition>
    </state>

    <state id="started">
      <transition event="gem.interface.commmand.generic.Stop">
        <target next="stopped"/>
      </transition>

      <transition event="gemdemo.promotion.ShopperEntered">
        <target next="started"/>
        <gem:do action="increaseCount"/>
      </transition>

      <transition event="gemdemo.promotion.ShopperLeft">
        <target next="started"/>
        <gem:do action="decreaseCount"/>
      </transition>

      <transition event="gemdemo.promotion.MallLocationQuery">
        <target next="started"/>
        <gem:do action="answerLocation" outputEventName="
            gemdemo.promotion.MallLocation"/>
      </transition>
```

```
      <transition event="internal::timer">
        <target next="started"/>
        <gem:do action="sendNextPromotion" outputEventName="
            gemdemo.promotion.Promotion"/>
      </transition>

    </state>

    <state final="true" id="stopped">
      <onentry>
        <gem:do action="stop"/>
      </onentry>
    </state>

  </scxml>
 </Behavior>
</ComponentInterfaceAdvertisement>
```