A Framework for Managing the Solution Life Cycle of Event-Driven Pervasive Applications

Johnathan M. Reason[1], Han Chen[1], ChangWoo Jung[2], SunWoo Lee[2], Danny Wong[1], Andrew Kim[2], SooYeon Kim[2], JiHye Rhim[2], Paul B. Chou[1], and KangYoon Lee[2]

[1] IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532
{reason, chenhan, dcwong, pchou}@us.ibm.com
[2] IBM Ubiquitous Computing Laboratory, The MMAA Building, 467-12 Dogok-dong, Gangnam-gu, Seoul 135-700
{jungcw, samlee, akhkim, sooyeon.kim, jhrhim, keylee}@kr.ibm.com

Abstract. Event-driven, embedded applications that embody the composition of many disparate components are emerging as an important class of pervasive applications. For such applications, realizing solutions often requires a breadth of expertise. Consequently, managing the solution life cycle can be a very complex, time-intensive process. In this paper, we present a framework that eases the complexity of managing the life cycle of event-driven, pervasive solutions. We call this framework Rapid Integrated Solution Enablement or RISE. Component composition and software reuse are two central concepts of RISE, where solutions are graphically composed from reusable components using a visual editor. We describe the RISE architecture and discuss an initial prototype implementation that leverages open source technologies, such as Eclipse. Additionally, we illustrate the efficacy of RISE with an example solution for RFID supply chain logistics.

1 Introduction

Pervasive applications are becoming prevalent in our society, especially ones with embedded solutions that are driven by events originating from various sensor modalities. The apparatus comprising an embedded solution is often assembled from disparate components, including hardware devices (e.g., sensors, actuators, programmable logic controllers, and displays) and software components (e.g., device adapters, agents, and event correlators). Thus, realizing an embedded solution can be a very complex process that requires a high-degree of expertise across many specialty domains, such as embedded programming, networking, device adapter programming, wireless communications, and user interface design.

These specialties are often performed across solution partners, including device OEMs, solution integrators, solution developers, and the customers. Solution integrators must integrate the hardware devices and software components into the apparatus, solution developers must write application code for specific customer requirements, solution developers must test and validate the solution, and

IT staff must incorporate the solution into the IT infrastructure. This approach often leads to one-off solutions that are not flexible enough to accommodate new requirements.

RISE is a graphical, actor-oriented software framework for managing the life cycle of event-driven, embedded solutions. Through greater reuse of component-based, customizable software, RISE can lower the total cost of ownership, facilitate rapid development, deployment, and management, and improve flexibility of solutions through dynamic configuration. RISE exploits the concepts of component composition, software reuse, and heterogeneous models of computation to provide the tooling and runtime support.

This paper is organized as follows. Sect. 2 describes the foundational background, Sect. 3 describes the RISE architecture, Sect. 4 describes our initial prototype implementation, Sect. 5 discusses a use case example, and we conclude with some comments about ongoing work in Sect. 6.

2 Background

In this section, we discuss the core concepts that form the basis of RISE and introduce the terminology used throughout the remainder of this paper.

2.1 Related Work

RISE gets its motivation from other tools that provide a graphical block diagram methodology for actor-oriented modeling. To name a few, Simulink from The Mathworks®, LabVIEW® from National Instruments, and Ptolemy II from the The Ptolemy Project of the University of California at Berkeley are examples of actor-oriented design environments. While all of these tools (and others) have their strengths and weaknesses, we found all of them lacking a unified framework for deployment and management of embedded environments. Nevertheless, we find instruction in their theoretical underpinnings and leverage some specific results from Ptolemy II.

Actor-Oriented Design. In actor-oriented modeling, components are called actors and can communicate and execute with other actors in a model, where a model is the composition of one or more actors. Hewitt first introduced the term actor to describe the concept of autonomous reasoning agents [1], and later Agha refined the term to describe a formalized model of concurrency [2,3,4]. The Ptolemy Project has further refined the term to embody more models of concurrency and support actors that do not necessarily have their own thread of control [5,6].

All actors have an external component interface that abstracts its internal state and behavior. An actors interface is defined by its port/parameter specification, where ports represent points of communication for an actor and parameters affect the behavior of an actor. Parameter values can be static or dynamic during execution of a model.

Connections between ports are called channels, and actors communicate over channels via some method of messaging. Thus, actors do not interact directly with other actors, only through channels. This differs from object-oriented design, where components communicate through method calls.

A model can also have an external interface, which represents a hierarchical notion of abstraction. A model's interface also consists of ports and parameters, which can be connected by channels to other ports of the model or to the ports of the model's internal actors. Similarly, a model's parameters can be used to determine the parameter values of its internal actors.

Model of Computation. The concepts above describe the abstract syntax of actor-oriented modeling. However, the semantics are governed by its model of computation (MoC).

An MoC defines the semantics of inter-component communications and the runtime execution semantics of a model. One might think of a model of computation as the rules governing component interaction and execution. These rules govern when and how a component invokes its internal computation, updates its state, and communicates through its ports.

There are many well know models of computation, too many to enumerate here. One key result coming from The Ptolemy Project is its comprehensive study of concurrent MoCs, and Ptolemy II provides open source, Java implementations for a full list of MoCs [7]. The initial RISE prototype leverages MoC implementations from Ptolemy II (see Sect. 4.3).

2.2 RISE Terminology

We mostly adopt the actor-oriented terminology described in Sect. 2.1, with a few extensions and differences. We define two general types of actors: atomic and composite. Atomic actors represent the most primitive of actors and are typically implemented in a high-level programming language, such as Java. In contrast, composite actors are hierarchical actors that are constructed by graphically connecting atomic actors and other composite actors. We reserve the term model to represent a composite actor that is a deployable application, and we use connection instead of channel.

Additionally, an atomic actor can be behavior-polymorphic, which we define as an atomic actor that has a generic external interface (i.e., ports and parameters), but defines an abstract interface for its internal implementation. Thus, behavior-polymorphic actors can have multiple concrete internal implementations. We use behavior-polymorphic actors to model actors that have similar attributes, such as a device adapter for a particular family of devices. Behavior-polymorphism complements the notions of data- and domain-polymorphism found in Ptolemy II [8,9,10].

3 Architecture

The RISE software architecture is comprised of three platforms: RISE Development Platform (RDP), RISE Runtime Platform (RRP), and RISE Library Server Platform (RLP). RDP provides the tooling for developers to build, deploy, and manage RISE-based solutions. RRP provides the software that supports execution of RISE-based solutions. While, RLP provides the means by which RISE runtimes can dynamically discover the actors that comprise a solution.

3.1 Development Platform

Fig. 1 illustrates the software architecture for RDP. The underlying computing framework for RDP is a Java virtual machine (JVM) running on a hardware device (e.g., Java 2 Standard Edition on a personal computer). The middleware layer provides the software that serves as the building blocks for the tooling and the artifacts created by the tooling. The tools layer provides all the functions that comprise the integrated development environment (IDE).

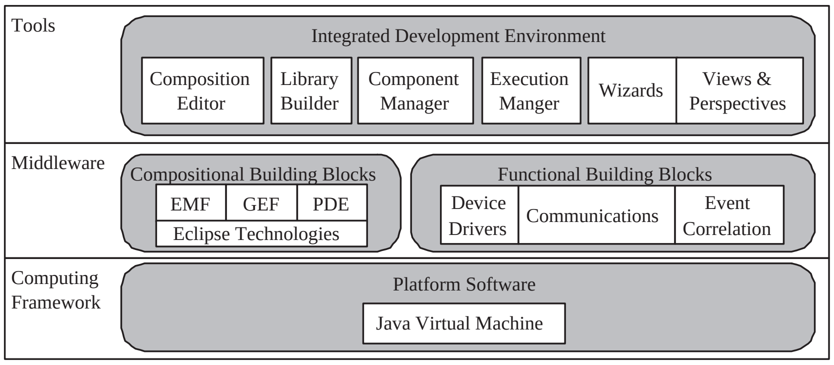

Fig. 1. RDP contains three layers: computing framework, middleware, and tooling

Integrated Development Environment. RDP's IDE provides a graphical user interface for a user to construct actors, models, and libraries. The tools that comprise the IDE are a composition editor, a component manager, a library builder, and a model execution manager. The composition editor presents a user with a canvas to edit an actors structural elements and to drag-n-drop actors from the component manager to the model diagram.

The component manager maintains the persistent storage of reusable components available from the local environment. The library builder is a utility that packages one or more actors or models into a deployable library and installs the library to an appropriate library server. The execution manager is a utility that provides the user interface and protocol to load models in the runtime, retrieve a model from the runtime, and display execution events. The IDE also supports a number of wizards, views, and perspectives to provide an intuitive design flow. One important view of the IDE is the library view, which provides four different means to access reusable actors (see Sect. 4.4).

Compositional Building Blocks. Eclipse technologies provide the compositional building blocks of RDP's IDE (see Sect. 4.2).

Functional Building Blocks. The devices that produce the event and data streams for a solution and comprise a solutions apparatus are central to most event-driven applications. Thus, the device adapters that provide the application-level software interface to devices are an important category. RDP supports building device adapters on top of manufacturer-specific device drivers or using generic device drivers (e.g., serial port). In addition, RDP provides the framework for porting other device adapters to the RISE device adapter abstraction via Eclipse plug-in technology.

Distributed communications where one instance of a solution might need to communicate to other remote entities, such as an enterprise server, is another important category. Using polymorphic components, RDP provides the framework for developing reusable communications components that can support different implementations of well know communications paradigms, such as HTTP client/servlet, publish/subscribe, and UDP/TCP.

Event correlation is the general terminology given to middleware technology that can identify patterns in one or more data sources, define events as the occurrence of one or more patterns, and then use the detected events to trigger some action. A user usually configures the patterns and events through a set of rules, which are specified by a rule language and executed by a correlation engine. RDP provides the framework for developing solutions with event correlation components.

3.2 Runtime Platform

Fig. 2 illustrates the software architecture for RRP, which contains two layers: computing framework and runtime. The underlying computing framework for RRP is the Open Services Gateway Initiative (OSGi) Service Platform Release 3 specification [11]. The runtime layer provides the runtime libraries and the runtime execution services.

OSGi Service Platform Release 3. OSGi defines a framework on which multiple applications can run on a single JVM. Using the OSGi framework, application developers partition applications into services, and then package these services into application bundles. Bundles can register services with the framework that other bundles can use; thereby, facilitating the sharing of services at

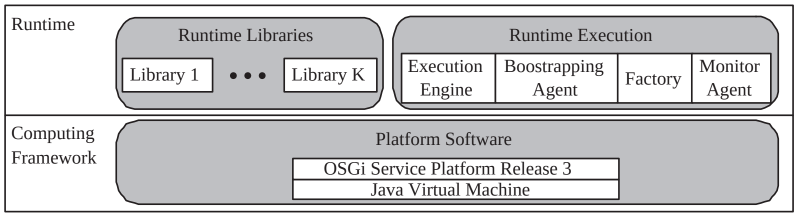

Fig. 2. The primary services of RRP are the bootstrapping agent, the factory, the execution engine, and the monitor agent. These services provide loading and unloading of a model, instantiation of the actors, execution of a model, and execution monitoring

the package level. In RISE, we package all RISE-specific technologies into OSGi bundles, including the RISE runtime libraries and runtime execution services.

Runtime Libraries. A RISE runtime library is an OSGi bundle that contains the class definitions for one or more RISE actors and/or models. Because RISE libraries are OSGi bundles, they derive all the benefits provided by the framework, including dynamic installation, automatic resolution of dependencies, and import/export of services and packages. These benefits provide a convenient means for supporting a dynamic, distributed method for resolving the composition of model during runtime. In particular, the child actors of a model need not be loaded from the same library, nor do all the libraries have to reside on the same library server.

Runtime Execution. The execution engine specializes a model to a particular MoC. The engine also supports execution of models with a heterogeneous mixture of MoCs, which is a concept we borrow from the Ptolemy Project. For our application domain, we exploit this concept primarily by augmented discrete event or data flow semantics with control flow.

The bootstrapping agent loads a model into the execution engine, passing along any necessary parameters. The bootstrapping agent provides a server interface for clients to connect to it. Clients can load fully executable models or load by reference. In the latter case, the composition of a model must be resolved and its child actors instantiated. For this purpose, the agent defers to the factory.

The factory services the bootstrapping agents requests to instantiate a RISE model, and it maintains inventory of all RISE actors available to the runtime. A RISE library contributes its actors to the factory through a factory contributor interface. The RISE factory exports its factory contributor interface via normal OSGi protocol. This function allows the runtime to add new libraries dynamically, thereby facilitating remote deployment and service discovery.

The monitor agent is the runtime service that allows clients, such as an RDP, to monitor execution flow of an RRP. This agent is primarily used for debugging during development.

3.3 Library Server Platform

The software architecture for RLP contains two layers: computing framework and library server. The underlying computing framework for RLP is also OSGi. The library server layer provides a reusable library repository, a library manager, and a deployment protocol, which interacts with the bootstrap agent in RRP for deploying libraries on demand.

The library manager supervises the storage and deployment of libraries in a heterogeneous network of devices. Through a deployment protocol, the library server delivers a library to an RRP in response to a request coming from the factory in an RRP. The library manager also provides an interface that allows an RDP to present a library browser view. Additionally, RLP provides the means of binding a generic polymorphic actor with its concrete implementation.

4 Implementation

4.1 OSGi Implementation

For our particular implementation of OSGi, we use Service Management Framework (SMF) 3.7, which comes with WebSphere Studio Device Developer (WSDD) 5.7.1. WSDD is built on Eclipse 2.11 technology. SMF facilitates deployment and it provides a standardized framework for RRP and RLP.

4.2 RDP Implementation

RDP is implemented on top of WSDD in the form of Eclipse plug-ins. The persistent state of all actors and models constructed using RDP is captured by the data model, which was designed using Eclipse Modeling Framework (EMF). Solutions created with the composition editor are persisted in XMI format, which is converted to Java code using a code generator. The generated Java code utilizes base classes defined in the runtime package.

The GUI for the IDE is built using Graphical Editor Framework (GEF), which provides the visual layout of all the graphical objects displayed in a model diagram.

A library view plug-in provides four different means to access reusable actors. First, there is a workspace folder containing all the RISE libraries under development in the current workspace. Second, there is a list of all available library servers, each of which contains a list of libraries of reusable actors. Third, there is a base library folder, which contains libraries of some pre-packaged actors available to the local environment. Lastly, there is an anonymous folder that lists the contained component classes of the composite actor currently being edited by the developer. This plug-in interacts with the component manager and any number of library servers.

A set of resource management plug-ins provide the wizards for creating RISE projects and libraries. A set of runtime management plug-ins provide the execution manager functionality. Additionally, RDP provides a local RRP and RLP, which enable rapid testing of solutions.

4.3 RRP Implementation

We implement the RISE runtime execution services as SMF bundles, and we deploy them on an SMF runtime. To date, we have tested RRP on a windows platform and on an embedded Linux platform (Arcom Viper®). For the embedded RRP, SMF runs on J9.

Our execution engine leverages the Discrete Event, Finite State Machine, and Synchronous Data Flow implementations from Ptolemy II. Since RISE execution services are SMF bundles, they also benefit from dynamic updates. As new services become available, for example a new MoC, RRP can discover the service and update the execution engine upon loading a model that uses the new MoC.

4.4 RLP Implementation

We implement the RISE deployable libraries as SMF bundles and install them in an SMF bundle server, using the tools provided by RDP. Currently, the RISE actor libraries include implementations of components relevant to the RFID application domain. We have implementations of various device adapters (e.g., motion sensors, RFID readers, and LED actuators), communications protocols (e.g., HTTP and publish/subscribe messaging), custom controllers and agents, and basic logic actors. For event correlation, we have an implementation for Application Level Event (ALE).

5 RFID Dock Door Receiving Use Case

The dock door receiving use case is an RFID application of supply chain management. In this section, we describe our experience in building a RISE solution to support this use case.

Referring to Fig. 3, while goods are being physically moved through the dock door, the RFID reader will read the goods' pallet and case RFID tags. The tag data is sent to the store's backend system, which will then check the data against the database to determine if the tags just read should be accepted or rejected. The accept/reject status is reported back to a controller at the dock door, then the controller's logic triggers a status indication on the light stack.

Fig. 4 illustrates what might be the initial step of building the solution, testing the RFID reader. The model was constructed via drag-n-drop from the library server view. The reader as a polling device whose poling frequency is optionally driven by an external clock. During the duty cycle of the polling period, the reader activates its antennas and starts reading tags in its field-of-view. Since an RFID reader can detect multiple tags in one read, the reader device packages multiple reads in the form of a map. Thus, a transform actor is needed to split the map into its individual RFID tag data objects. The output of the map splitter is simply sent to a tag logger, which writes the RFID tag ID to a stream.

Fig. 5 illustrates the ease with which a developer can modify an existing solution. Since the RFID reader adapter is designed to report any tag that it

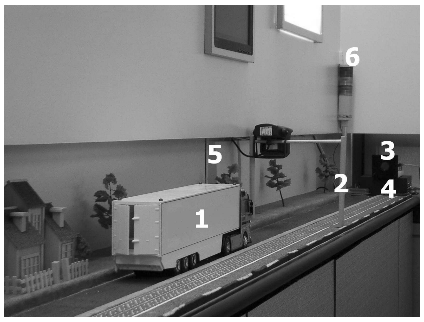

Fig. 3. The demo apparatus contains a delivery vehicle (1), a warehouse dock door (2), a ranging sensor (3), an embedded controller (4), an RFID reader (5), and a light stack (6)

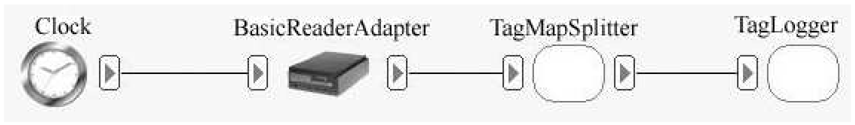

Fig. 4. Snapshot of the editing canvas illustrating an example model for testing an RFID reader

detects during a poll, redundant RFID tag reads are often observed. Thus, we can improve our model with a temporal filter to eliminate duplicate tag reads.

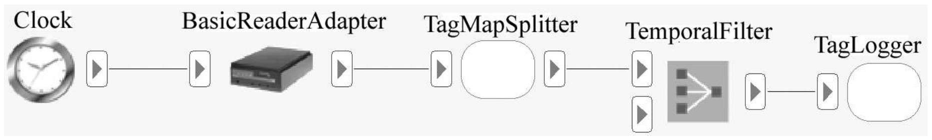

Fig. 5. RISE model for eliminating duplicate reads from RFID reader

Fig. 6 shows the complete RISE model that implements the dock door receiving use case, which is an extension of the previous examples. Major additions include a sonar range sensor, a tracking agent, a light adapter, and tag validation. Like the reader, the sonar device is a polling device whose period is driven by a clock. The sonar device's output is processed by the tracking agent, which determines whether an object is in proximity to the dock door. The tag validation component communicates to the backend system and controls the light stack.

While some base knowledge of RFID systems is required for constructing useful scenarios, this example illustrates how changing the behavior of an existing solution is quite simple.

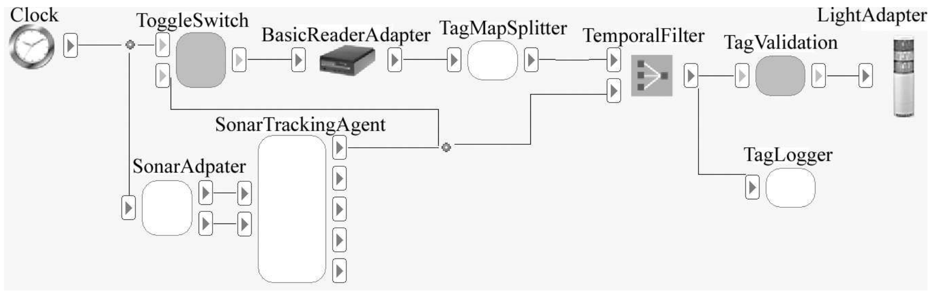

Fig. 6. Complete RISE model for the example dock door receiving use case

6 Future Work

This paper represents results from the first phase of the RISE project. The prototype developed provides a platform for us to evaluate the methodology. We plan to conduct user studies involving naïve users and practitioners. We are also addressing scale and management issues by adding capability to support solutions involving multiple, distributed computing nodes.

References

1. C. Hewitt, Viewing Control Structures as Patterns of Passing Messages, Jour. of Art. Intel., 8(3):323363, June 1977.
2. G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, 1986.
3. G. Agha, Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems, in Formal Methods for Open Object-based Distributed Systems, IFIP Trans., E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.
4. G. Agha, Concurrent Object-Oriented Programming, Comm. of the ACM, 33(9):125140, Sept. 1990.
5. E. A. Lee and S. Neuendorffer, Concurrent Models of Computation for Embedded Software, IEE Proc. Comp. and Dig. Tech., 2005.
6. Edward A. Lee, Computing for Embedded Systems, IEEE Instr. and Meas. Tech. Conf., Budapest, Hungary, May 21-23, 2001.
7. E. A. Lee, et al, Volume 3: Ptolemy II Domains, http://ptolemy.eecs.berkeley.edu, UC Berkeley, 2005.
8. L. de Alfaro and T. A. Henzinger, Interface Theories for Component-Based Design, Proc. of EMSOFT 2001, Tahoe City, CA, LNCS 2211, Springer-Verlag, Oct 2001.
9. E. A. Lee and Y. Xiong, A Behavioral Type System and Its Application in Ptolemy II, Formal Aspects of Computing Journal, special issue on Semantic Foundations of Engineering Design Languages, Volume 16, Number 3, August 2004.
10. R. Milner, A Theory of Type Polymorphism in Programming, Jour. of Comp. and Sys. Sci., 17, pp. 375-384, 1978.
11. OSGi Alliance, OSGi Service Platform, Release 3 Specification, http://www.osgi.org, March 27, 2003.