A RTS Framework with Self-Adaptive Mechanism and Built-In Interoperability

Emilia M. Colonese, José M. Parente de Oliveira

Instituto Tecnológico de Aeronáutica – ITA
Praça Marechal Eduardo Gomes, 50
São José dos Campos – SP – Brazil 12228-900
{colonese,parente}@ita.br

Abstract. The practicality of achieving interoperability remains a challenge for rapidly process operational requirements allowing system or subsystem components to work together. It must maintain the behavior and data integrity, while improves the answer time. A framework for data interoperability among Real-Time System (RTS) components is proposed by this research to replace gateways. It implements generic behavioral models for monitoring and controlling system composed of self-adaptive dynamic service servers based on received events of the real world, service invokers, a common protocol class, a component class, a message class, and a use case class. The target system might be adjusted to the framework, allowing the software components to have high degrees of cost-effective reuse.

1. Introduction

The construction of complex systems usually uses pre-defined frameworks. A framework is a collection of collaborating classes that provide a set of services for a given domain [2]. According to Booch [1] a system does not need to be written from scratch if it reuses framework elements.

This work shows the conceptualization, implementation, and deployment of a Real-Time System (RTS) Framework, allowing interoperability among Components, Systems or Subsystems, besides providing self-adaptive mechanism for services. The proposed approach was built based on the following insights: monitoring the correct deployment of system requirements; offering an effective interoperability capability that replace gateways and is easily applied to system components; and providing self-adaptive services oriented architecture according to received events at run-time. Gateway is a link between two computer programs. It acts as a portal between two programs allowing them to share information and bypass certain protocols on a host computer.

The authors believe that this approach is unique in the sense of modeling and tracking functional requirements along with system dimensions and components through a control mechanism of use cases actually activated during the system execution. Also, the RTS Framework provides a good solution for interoperability issues, besides allowing run-time adaptation of functional behaviors.
2. Approach

The authors propose three modular viewpoints representing the system based on different concern areas [3] that should work together: system functionalities (requirements), interoperability, and system infrastructure.

Defined requirements might be presented as use cases. A use case describes how users will interact with an application and what information they need from the system to accomplish their task. The infrastructure viewpoint is concerned with non-functional requirements or Quality of Service (QoS) used to achieve as well as possible the desired functional aspects for the target system, which is acquired through non-functional requirements specification. The interoperability viewpoint was created based on authors’ findings: a common pattern used for components and subsystems communications during a RTS design phase. Since a Design Pattern is a generalized solution to a commonly problem occurring, the interoperability viewpoint can be described as a design pattern that implements specific properties, which are not delivered by analysis models. This viewpoint allows reusability of a common communication protocol connects the system functionalities and the system infrastructure concerned to interoperability.

Those three viewpoints are implemented in a framework, which maps requirements to use cases, managed by dynamic Services Servers at run time. The activation of a specific Service Server depends on external events, which are captured by a pre-defined use case. The proposed framework focuses on system functionalities and interoperability viewpoints.

The authors used the IBM-Rational Rose Real-Time (RRRT) [6], a Computer-Aided Software Engineering (CASE) for real-time modeling tool [2], which extends the classic UML diagrams to UML-RT diagrams including the constructor set capable of supporting real-time modeling, design, implementation, and testing.

3. The RTS Framework

Frameworks allow code reuse and fast application developments [2]. The RTS Framework was built with interoperability mechanisms and dynamic services that can be “plug-in/plug-out” at real-time to the SystemController (SC) Capsule. The authors used basic components of RRRT, and passive classes to handle the inclusion/exclusion of system components, a common communication protocol, message signals, and use cases mapped from stakeholders’ requirements; in order to implement the proposed framework. The RTS Framework structure provides tracking of those requirements assuring an effective use case implementation in the system.

The External_Invokers, Internal_Invokers, and SystemStarter Capsules are the system starters. Their statecharts were created on design-pattern bases to implement theirs own behavior. The Interoperability Pattern is applied on their statechart by adjusting the state responsible for the communication among other subsystems or systems, during the design phase.

3.1. System Controller Capsule

In order to implement self-adaptive characteristics, an Event Awareness Service, named ServiceDistribution was created as a Capsule. It also implements the communications
and data interchanges among Systems, Subsystems or Capsules promoting the data interoperability, with no need of gateways.

In this paper, it is considered the transaction of receiving and answering to events, a use case realization. In order to the ServiceDistribution be able of choosing, at run time, the right Service Server, an self-adaptive policy was implemented. This policy is defined by a kind of Event-Condition-Action (ECA) rule. The Event is the received message, which is already related to some use case. The Condition, which is defined by use cases, specifies what service needs to be activated. Finally, the Action is the activation of the correct service. The authors have defined rules to activate ServiceServer Capsules that provides a required service based on events captured by system’s use cases [3, 5].

3.2. Passive Classes

The format design for data interchange was essential for the framework. It has enabled the interoperability among system components by encapsulating it in a structured class, named CommonProtocol Class. This class contains all needed information to identify the event type, the invoker component, the service component, the use case, the message time, the message priority, and the data. The passive class named UseCase Class to register and trace the implementation of all use cases that are defined for the system.

Invokers, Starter, and Service Servers are the actors of the proposed framework. The actors are part of the system’s components. A passive class, named Component Class was created to register all defined system components that might connect to the System Controller. The Common Protocol Class uses a signal to identify the message type transmitted. A passive class, named Message Class was created to store all kind of signals used by the framework.

4. Case Study

A RTS Prototype was implemented applying the RTS Framework with an Internal_Invoker, the SystemController and all the passive classes. The Prototype sends missions, receives collected data and controls an Unmanned Air Vehicle (UAV) from a Control Station (CS). The mission is triggered from an Internal_Invoker Capsule, which invokes a Service provided by the CS. That Service is performed by activating the related Service Server at run-time, to send Remote Sensing missions to the UAV and receive the related mission results. The SystemStarter was not part of this case study.

The scope of the RTS System was reduced to implement a selected set of functionalities (translated to services): Communication; UAV Navigation Control; Target Remote Sensing; Georeferenced Information; and Situational Awareness achieved by graphic scenario visualizations.

4.1. Adopting the framework

In order to successfully reuse the framework 4 steps, shown in Table 1, must be applied.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>Adopting the CommunicationProtocol and Message classes</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>Updating the UseCase and the Component classes by inserting the required data</td>
</tr>
</tbody>
</table>
Step 3 | Adjusting the RTS System to the RTS Framework
--- | ---
Step 4 | Adjusting the External/Internal_Invokers and SystemStarter Subsystems applying the Interoperability Pattern

After the *Internal_Invoker* adaptation to the framework the ECA rules has been implemented in the newer expanded state dimension to activate the required Service Server and to exchange data with the CS Subsystem without gateways. The capsule responsible for the UAV Subsystem connection with the CS Subsystem also had its state diagram adjusted to the framework in order to built-in the Interoperability Pattern into the UAV Subsystem.

The employment of self-adaptation mechanisms could be observed in the RRRT execution view, when the external event coming through the Invoker activates the required Service Server at run-time. The RTS Prototype validated successfully the RTS Framework approach.

### 5. Conclusion

This paper presents the implementation of a self-adaptive framework for real-time system, which allows at run-time the right service activation to perform actions depending on external or internal events of a given scenario. The creation of an efficient design pattern for interoperability by defining a semantic for message exchanging, has allowed an easy and transparent message interchange among subsystems, as shown during the adjustment of the UAV prototype to the framework.

The authors of this paper believe that this approach is unique in the sense of built-in interoperability, and modeling and tracking functional requirements throughout system dimensions. The major findings of this work were a proposed solution for some gaps of tracking the correct deployment of use cases, and the creation of an interoperability design pattern. Future works could extend the use case class creating metrics collected and stored at run-time.

### References


