Chapter 5

A programming model for OSRA

5.1. Introduction

In the previous chapters we have seen model-driven software development approach that was centered on component-based techniques. Dijkstra's principle of separation of concerns was one of the cornerstone principles which was part of the software reference architecture and the component model proposed [PV12; PV14]. According to it, the user design space should be limited to the internals of the components, where only strictly sequential code can be used and the extra non-functional requirements are declaratively specified in the form of annotations on the component provided interfaces. This is already explained in detail in the Step 7 (Specification of non-functional attributes) in Chapter 3.

As discussed in the previous chapters, the reference software architecture is made up of a component model, a computational model, a programming model and a conforming execution platform. It is also clear that the component model should be statically bound to a computational model to formally define the computational entities and the rules which govern their usage.

The realization of extra functional properties or more precisely, the generation of the complete infrastructure code can be done in two steps:

- Automated generation of the non-functional code, code for handling concurrency and interaction requirements for communication between components and the skeletons for the components themselves
- Automated generation of containers for components and the connectors between components

A code generator needs to be developed for this purpose and the next few chapters would be concerned about realizing the above mentioned steps. As a result, the third-party software supplier can solely concentrate on implementing the functional code of the components. This is in line with the principle of separation of concerns, which is of very high interest.

The ASSERT project (Automated proof-based System and Software Engineering for Real-Time systems), was the first, large project which showed the feasibility of a development approach for high-integrity real-time systems centered on separation of concerns, correctness-by-construction and property preservation.

In the ASSERT project, which incorporated Model-driven-engineering approaches [Pan11], a modelling infrastructure named RCM was developed at the University of Padua [BPV08]. This infrastructure included a graphical modeling language and an editor, a model validator and set of model transformations that were necessary to feed model-based analysis and code generation [BPV08]. Ravenscar Computational Model (RP) was chosen as a computational model in this modelling infrastructure and RP directly emanated from the Ada Ravenscar Profile in language-neutral terms [PV12]. RP basically did not allow any language constructs that were exposed to unbounded execution-time and non-determinism [BDV04; PV12; PV13a]. Certain RP-compliant code archetypes were developed to complete the formulation of a programming model in the ASSERT project, which adhered to the vision of principle of separation of concerns and amenable to code generation [PV13a]. The code archetypes used Ada run-time, which is more compact in foot-print and hence could fit the needs of typical embedded systems which were resource-constrained [BDV04]. The archetypes developed in the ASSERT project were based on the previous work on code generation from HRT-HOOD to Ada [PV12; PV13a].

In the following Artemis JU CHESS project, which was an initiative from ESA in parallel to the development of the SCM [Pan11; PV14], these code archetypes from the ASSERT were revised by adding certain features [PV12]. The code archetypes in the CHESS project also targeted the Ravenscar Computational Model for the additional reason that the reduced tasking model used in the Ada Ravenscar Profile matched the semantic assumptions and communication model of real-time theory, the response-time analysis in particular [PV13a]. The code archetypes developed in the CHESS project [PV12] are taken as reference for developing a programming model in this chapter, for further use. The code archetypes discussed in this Master thesis however, target the Tasking framework which is the chosen computational model for this Master thesis. The reasons for choosing Tasking framework as a computational model is already explained in the end of the previous chapter. The code archetypes discussed in this chapter are first steps towards generation of the complete infrastructural code.

5.2. Structure of the code archetypes

The code archetypes discussed in this Master thesis strive to attain as much separation as possible between the functional and extra-functional concerns. At the implementation level, functional/algorithmic code of a component is separated from the code that manages the realization of the extra-functional requirements like tasking, synchronization and different time-related aspects.

The library of sequential code, which may have as many cohesive operations as the software supplier wishes to include in a single executing component, is included in a closed structure. The mapping of this structure to the actual design entity of the infrastructural code is not of concern in this chapter. The sequential code in this structure is executed by a distinct flow of control of the system. The dedicated flow of control can be an active task, together with other tasking primitives from the Tasking framework (if the desired concurrency kind is `asynchronous`) or a simple synchronous method/operation invocation using the flow of control of the component requesting the service from outside (if the desired concurrency kind is `synchronous`). This leads to a combined effect that the component internals are completely hidden from outside, but the provided services invoked by the external clients are executed with the desired interaction semantics.

As multiple clients may independently require the range of services to be executed by one of the two desired flow of controls, it is necessary to safeguard these execution requests. Safeguarding of execution requests in case of synchronous service requests is implicit as these requests would have been raised in the respective flow of control of the component asking for the service, but the safeguarding of execution requests in case of asynchronous requests needs to be handled. The handling of these kind of requests, is explained in the next parts of this section.

Service requests can often lead to valid/invalid data that need to be sent back safely to the components which made the requests. The service requester also need to be informed about any exceptions that might arise due to any unexpected situations during the servicing of the requests. The mechanisms and semantics necessary for realizing these requirements are also explained in the next parts of this section.

5.2.1. Synchronous release patterns

The archetypes for a synchronous release pattern are quite straight forward. When a request for a service is made with the desired concurrency kind specified as `synchronous`, the request is handled straight-away as a normal function/operation call in the flow of control of the service requester. The results (if any) from the service requests, and

exceptions (if any) during the course of handling the service requests are returned back to the service requester using the same flow of control.

Protected

When the non-functional property set on the service in the provided interface side of the component offering the service is `Protected`, it is necessary for the container that wraps around the component to safeguard this non-functional property. As the container is the entity that promotes the provided interface of the component, it intercepts the function/operation call from outside and provides exclusive access to the service implemented by the component. Semaphores provided by the Tasking framework is used for this purpose.

Semantics explained in previous chapters?

`Unprotected` format different the before

When the non-functional property set on the service in the provided interface side of the component offering the service is `Unprotected`, the semantics of handling the service request is essentially the same as the way the protected operations are handled, except for the fact that obtaining and releasing of the semaphore for the operation is not anymore needed.

Maybe provide an example or a sequence chart ??

Did I understand correctly: Protected & Unprotected are the variants of "synchronous"?

5.2.2. Asynchronous release patterns

The archetypes for an asynchronous release pattern are quite complicated when compared to the way the requests with synchronous release pattern. As the requests cannot be anymore handled in the flow of control of the service requester, tasks from Tasking framework along with other tasking primitives, which are independent threads of execution can be activated to cater to these requests on the provided interface side.

are realized

The asynchronous service request is initially intercepted at the required interface port subsumed by the container of the component which makes the request. Here the data (if any) associated with the request is packaged and the packaged data is forwarded to the provided interface port, which is promoted by the container of the component handling the request. Along with the data associated with the service request, it is also important that the required interface port packages information about how to send back the results (if any) and exceptions (if any) to the service requester.

How is this done?

Each thread of control, having its own structure as explained below, is responsible for only one operation in the provided interface side of the component that handles requests.

As the release patterns for requests are already decided statically and as these release patterns are not expected to change at run-time, the number of threads of control that will be necessary to handle the service requests will be known at compile-time.

This is very similar to the way asynchronous release patterns are handled in the code archetype listed in [PV12; PV13a] except for the fact that they do not consider service requests which might result in results or exceptions that need to be sent back to the service requester [PV13a].

Sporadic

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Sporadic`, it is the responsibility of the container of the service provider component to safeguard this property. The sporadic property requires that two subsequent requests for the service needs to always be separated by no less but possibly more than a minimum guaranteed time span, known as the MIAT (Minimum Inter-Arrival Time) [Pan17; PV14]. The container makes use of tasking primitives such as a task channel, task event and a task from the Tasking framework for this purpose.

The general structure of the thread of control on the service provider end, necessary to handle sporadic service requests consists of a task with two synchronized task inputs, attached to it. One of the task inputs is associated with a task event, with absolute timing (fixed task wake-up times) and the other task input is associated with a normal task channel. The task event is configured to wake up the task periodically after every MIAT interval. The task input associated with a normal task channel is configured so that the task input is activated as soon as a push is made against its associated task channel. This task then is instantiated in the container of the service provider component.

When a provided interface port, promoted by the container of the component handling the request, receives a sporadic service release request, it intercepts the request and pushes the packaged data against the channel associated with the task.

Because of synchronized task inputs, the task is activated only after both its task inputs are activated. When activated, the functions of the task will then be to:

Step 1 Unpack the packaged data

Step 2 Acquire the semaphore provided by the Tasking framework associated with the service

Step 3 Execute the desired service

Step 4 Reset the task event attached to the task

Step 5 Release the semaphore acquired

Step 6 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port

In this way, the non-functional properties associated with an asynchronous sporadic release pattern can be preserved at run-time.

Protected

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Protected` it is the responsibility of the container of the service provider component to safeguard this property. The container makes use of tasking primitives such as a task channel and a task from the Tasking framework for this purpose.

The general structure of the thread of control on the service provider end, necessary to handle this kind of service requests, is a task with one task input attached to it. The task input is configured so that the task input is activated as soon as a push is made against its associated task channel.

When a provided interface port, promoted by the container of the component handling the request, receives a service release request of this kind, it intercepts the request and pushes the packaged data against the channel associated with the task. The task is then activated and the functions of the task will then be to:

Step 1 Unpack the packaged data

Step 2 Acquire the semaphore provided by the Tasking framework associated with the service

Step 3 Execute the desired service

Step 4 Release the semaphore acquired

Step 5 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port

In this way, the non-functional properties associated with an asynchronous protected release pattern can be preserved at run-time.

Bursty

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Bursty`, it is the responsibility of the container of the service provider component to safeguard this property. The bursty property requires that a service can be activated at most a given number of times in a given interval called the bound interval [Pan17; PV14].

The general structure of the thread of control on the service provider end, necessary to handle this kind of service requests is a task with two non-synchronized task inputs attached to it. One of the task inputs is associated with a task event, with absolute timing (fixed task wake-up times) and the other task input is associated with a normal task channel. The task event is configured to wake up the task periodically after every bound interval. The task input associated with a normal task channel is configured in a way that the task input is activated as soon as a push is made against its associated task channel. The task also has an internal counting semaphore provided by the Tasking framework in order to keep a count of the number of service requests handled within the bound interval.

When a provided interface port, promoted by the container of the component handling the request, receives a service release request with bursty nature, it intercepts the request and pushes the packaged data against the channel associated with the task.

Because of non-synchronized task inputs, the task is activated if any one of its task inputs are activated. When activated, the functions of the task will then be to:

Step 1 Check the activated input. If the activated input is the one that is attached to a task event, then replenish the counting semaphore, restart the attached task event and go to step 8.

Step 2 Unpack the packaged data

Step 3 Acquire the counting semaphore local to the task which is used to enforce the max. number of activations within a bound interval

Step 4 Acquire the semaphore provided by the Tasking framework associated with the service

Step 5 Execute the desired service

Step 6 Release the semaphore associated with the service

Step 7 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port

In this way, the non-functional properties associated with an asynchronous bursty release pattern can be preserved at run-time. It is important to note that the code archetypes which were developed for the CHESS project do not mention the scheme to handle this kind of release pattern [PV12; PV13a]. It is unclear from the available resources whether the reason to not consider this code archetype was because of the non-possibility to opt this non-functional property for a service request on the provided interface side.

Did you ask some one of the CHESS project by mail?? If not, you may be should

Cyclic

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Cyclic`, it is the responsibility of the container of the service provider component to safeguard this property. Cyclic property requires that the associated request be activated periodically and with a non-zero initial offset [Pan17; PV14].

The general structure of the thread of control on the service provider end, necessary to handle this kind of service requests is a task with a task event provided by the Tasking framework, attached to it. The task event is configured to wake up the task periodically, with absolute timing (fixed task wake-up times). The task event can also be configured to wake up the associated task for the very first time with an initial offset.

When a provided interface port, promoted by the container of the component has a service which needs to be activated periodically, the task is activated and it performs the following functions:

Step 1 Acquire the semaphore provided by the Tasking framework associated with the service

Step 2 Execute the desired service

Step 3 Release the semaphore acquired

The service with a cyclic nature cannot be requested from an external component [Pan17]. The services also need to be parameter less and cannot send out results or throw exceptions [Pan17].

In this way, the non-functional properties associated with an asynchronous cyclic release pattern can be preserved at run-time.