Document Title	Applying Simulink to AUTOSAR
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	185
Document Classification	Auxiliary

Document Version	1.0.6
Document Status	Final
Part of Release	3.2
Revision	1

Document Change History

Date	Version	Changed by	Change Description
23.03.2011	1.0.6	AUTOSAR Administration	Legal disclaimer revised
23.06.2008	1.0.5	AUTOSAR Administration	Legal disclaimer revised
31.10.2007	1.0.4	AUTOSAR Administration	• Document meta information extended • Small layout adaptations made
24.01.2007	1.0.3	AUTOSAR Administration	• “Advice for users” revised • “Revision Information” added
04.12.2006	1.0.2	AUTOSAR Administration	Legal disclaimer revised
28.06.2006	1.0.1	AUTOSAR Administration	Layout Adaptations
28.04.2006	1.0.0	AUTOSAR Administration	Initial release

Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Introduction 5
1.1 Related Documents 5
1.2 Terminology 5
1.3 Scope 6
1.3.1 In Scope 6
1.3.2 Out of Scope 7
1.4 Document Overview 7

2 Mapping of AUTOSAR software components to Simulink 8
2.1 AUTOSAR Concepts / Simulink Concepts 8
2.2 AtomicSoftwareComponentType 10
2.2.1 Canonical Pattern 11
2.2.1.1 Current Limitations 11
2.2.2 AUTOSAR Ports 12
2.2.3 Sender/Receiver Interface (SenderReceiverInterface) 13
2.2.4 Client/Server Interface (ClientServerInterface) 15
2.2.5 Sensor Actuator Software Component 16
2.2.5.1 Simulation 16
2.2.5.2 Code Generation 18
2.2.6 Services 18
2.2.6.1 Simulation 18
2.2.6.2 Code Generation 19
2.3 Modeling Runnables Explicitly 19
2.3.1 Canonical Pattern 19
2.3.2 Code Generation Layout 20
2.3.3 Communication Modes of Sender-Receiver Communication 22
2.3.3.1 Code Generation 22
2.3.3.2 Simulation 23
2.3.4 Inter Runnable Communication 23
2.3.4.1 Code Generation 23
2.3.4.2 Simulation 23
2.3.5 Exclusive Areas 24
2.3.5.1 Code Generation 25
2.3.6 Sensor Actuator Component Runnables 25
2.3.6.1 Code Generation 26
2.3.6.2 Simulation 26
2.3.7 Runnables That Access Services 27
2.3.8 Complex Runnables 28
2.3.8.1 Annotations 28
2.3.9 Communication Specification – ComSpec 30
2.3.9.1 ComSpec – initValue (Receive and Send) 31
2.3.9.2 ComSpec – Acknowledgement Request and timeout (Send) 31
2.3.9.3 Code generation 31
2.3.9.4 Simulation 31
2.3.9.5 ComSpec – canInvalidate (Send) 31
2.3.10 DataElement Filter (Receive) 32
2.3.10.1 Code generation 32
2.3.10.2 Simulation 32

2.4 Compositions 33
2.4.1 Canonical Pattern 33
2.5 The Use of Types and Prototypes in AUTOSAR 34
2.5.1 Modeling one Software component 34
2.5.2 Modeling several Software components 35
2.5.2.1 Simplification 35
2.6 Communication between Atomic Software-Components 36
2.7 Datatypes, Constants and Parameters 37
2.7.1 Datatypes 37
2.7.1.1 Primitive Datatypes in AUTOSAR 37
2.7.1.2 Composite Datatypes in AUTOSAR 37
2.7.2 Characteristics 38
3 RTE Simulation 39
3.1 Basic Mechanism 39
3.2 Modeling Runnables for Sender Receiver Communication 40
3.2.1 Sender Receiver Communication 40
3.3 Modeling Runnables for Client Server Communication 43
3.3.1 Client Server Communication – Client Model 43
3.3.1.1 Synchronous Client Server Communication 43
3.3.1.2 Asynchronous Client Server Communication 44
3.3.2 Client Server Communication – Server Model 45
3.3.2.1 Simulation 45
3.3.2.2 Code generation 45
3.3.3 Client Server Communication – Software Component View 46
3.4 Modeling distributed functional networks – Simulation of Communication Delays 47
3.5 Simulation of Communication Specification – ComSpec 48
3.5.1 ComSpec – initValue 48
3.5.2 ComSpec – Acknowledgement Request 48
3.5.2.1 ComSpec – canInvalidate 49
3.5.3 DataElement Filter 49
4 References 50
4.1 Normative References to AUTOSAR documents 50
4.2 Normative References to Non-AUTOSAR documents 52

1 Introduction

Model-Based Design with Simulink [15] has become a very common approach to designing and implementing embedded software systems, particularly embedded control. This document explores how AUTOSAR concepts are mapped to equivalent Simulink concepts in order to make use of Model-Based Design techniques for AUTOSAR.

1.1 Related Documents

The following documents are related to this Styleguide:

- This Simulink Styleguide is based on the "**Specification of Interaction with Behavioral Models**" [3], which is independent of particular behavior modeling tools. In the specification general use cases and requirements for mapping AUTOSAR modeling elements to behavior models and vice versa are defined. In the center this specification identifies the parts of the overall AUTOSAR meta-model [4], which are relevant to Behavior Modeling.
- The "**Specification of Feature Definition of Authoring Tools"** [12] gives a recommendation for a stepwise implementation of the overall AUTOSAR concept with respect to the interchange descriptions, namely the Software-Component Template [5], the ECU Resource Template [11] and the System Template [10]. As the basis for a first implementation, a subset (corresponding to the definition of features) of the AUTOSAR templates mentioned above for a first implementation of AUTOSAR authoring tools is defined.
- The "**Specification of Interoperability of Authoring Tools"** [8] emphasizes on issues that might come up when exchanging AUTOSAR models between different tools. After describing some basic concepts of data exchange the document sketches strategies on how these issues can be resolved. Requirements on AUTOSAR Authoring Tools for ensuring interoperability are defined.
- The "**Specification of graphical Notation"** [12] defines the graphical AUTOSAR notation for AUTOSAR Authoring Tools. For example, the document provides a comprehensive schema for graphically modeling Software components. The graphical notation should be used as a guideline for implementing AUTOSAR Authoring Tools.

1.2 Terminology

In this section the terminology as used throughout this document is defined. The definitions are to some extend specific for the scope of AUTOSAR and especially for this deliverable. Common use of these terms, however, is taken into account as far as possible:

- An **Authoring Tool** is an AUTOSAR tool operating on any form of AUTOSAR models describing systems (software component, ECU hardware, network topology and system constraint descriptions). It is regarded to be a design entry tool for AUTOSAR descriptions according to the respective templates. Typical functions may include creating, retrieving, modifying, validating and storing such descriptions. An authoring tool may provide a tool specific language or notations for the design entry, typically used as the expressive language at the tool's user interface.

These languages might differ from those used for AUTOSAR standard description formats. E.g. a graphical behavior modeling tool could be used to edit a software component description, stored as an XML file according to the software component template. Being an authoring tool is thus more a dedicated role of a tool than a classification of a tool itself.

- **AUTOSAR Model** is a generic expression for any kind of representation of instances of the AUTOSAR meta model. It might be a set of files in a file system, an XML stream, a database or memory used by some running software, etc.
- **AUTOSAR Tools** are software tools that may occur within the AUTOSAR methodology and support interpreting, processing and/or creating of AUTOSAR models.
- **Behavior** is used in two ways in AUTOSAR. On the one hand, behavior is used as an abbreviation for `InternalBehavior` – to describe the scheduling relevant aspects of a software component. On the other hand, behavior is a common control engineering term used to identify the functional input/output relation of a control design over time. Throughout this document the term behavior and combinations of this term like behavior models should be understood in this control engineering interpretation.
- A **Behavior Modeling Tool** (BMT) is used to edit a functional behavior model in a functional behavior modeling language.

1.3 Scope

This document is concerned with describing the elements that constitute an AUTOSAR software component in Simulink. Given this “mapping” of the component structure it is possible to use Simulink to describe the behavior of individual control modules, for application with a code generator to create implementations that are harmonized with the AUTOSAR RTE, and to create system wide simulators of the entire ECU network and physical environment.
In this document, Simulink will be considered in the role of a behavior modeling tool rather than an AUTOSAR authoring tool.

1.3.1 In Scope

The following items are considered in scope for this document:

- Mapping of the AUTOSAR meta-model relevant for behavior modeling, as defined in “Interaction with Behavior Models” document [3], to Simulink.
- Use of features that are present in MATLAB/Simulink/Stateflow R2006a [15]
- A description of how to generate application code that is harmonized with the AUTOSAR RTE layer.
- A description of how to create a simulation of the AUTOSAR software environment.
- The use of client/server communication for software component to AUTOSAR services communication and sensor actuator software component to ECU abstraction communication.

1.3.2 Out of Scope

The following items are considered out of scope for this document:

- A description of how AUTOSAR xml files should be imported/modified/exported from Simulink.
- The use of client/server communication mechanism for software component to software component communication (excluded from the meta-model relevant for behavior modeling, as defined in “Interaction with Behavior Models” document [3]).
- A description of how to model category 2 runnables.

1.4 Document Overview

The document is structured as follows: Chapter 2 contains the mapping of AUTOSAR software component meta-classes to their Simulink constructs and details the strategy for generating implementations that are harmonized with the AUTOSAR RTE. Chapter 3 highlights simulation aspects that are relevant for system-wide modeling not least the modeling of distributed networks of software components.

2 Mapping of AUTOSAR software components to Simulink

2.1 AUTOSAR Concepts / Simulink Concepts

As this document may be read by Simulink users who are not so familiar with AUTOSAR concepts, Table 1 summarizes the key mappings between AUTOSAR concepts and their proposed Simulink concepts.

AUTOSAR Concept	***AUTOSAR Description [1] [4]***	***Simulink Concept***	***Document Section***
Atomic Software Component	Smallest non-dividable software entity, connected to the AUTOSAR Virtual Functional Bus, relocatable [1].	Can be represented as any type of subsystem (virtual & non-virtual) and also by a model. NB. AUTOSAR's notion of atomic is not to be confused with a Simulink atomic subsystem.	2.2
P-Port Provide-Port	Specific Port providing data or providing a service of a server [1].	Outport for sender/receiver communication	2.2.2
R-Port Require-Port	Specific Port requiring data or requiring a service of a server [1].	Inport for sender/receiver communication	2.2.2
PortInterface	A PortInterface characterizes the information provided or required by a port. Can be either sender/receiver interface or client/server interface.	Abstract class – no realization in Simulink.	--
ComSpec	ComSpec defines specific communication attributes [4].	Various concepts see sections	2.3.9/3.5
Sender/Receiver Interface	A sender-receiver interface is a special kind of port-interface used for the case of sender-receiver communication. The sender-receiver interface defines the data-elements which are sent by a sending component (which has a p-port providing the sender-receiver interface) or received by a receiving component (which has an r-port requiring the sender-receiver interface) [1].	BusObject and bus selector/creators	2.2.2
Client/Server Interface	The client-server interface is a special kind of port-interface used for the case of client-server communication. The client-server interface defines the operations that are provided by the server and that can be used by the client [1].	Specific blocks, realizing RTE-API	2.2.4
Sender Receiver Annotation	Annotation of the data elements in a port that realizes a sender/receiver interface [4].	description field assigned to a specific DataElementPrototype inside the Runnable subsystem	2.3.8.1

Sensor Actuator Software Component	AUTOSAR SW-Component dedicated to the control of a sensor or actuator [1].	Virtual subsystem	2.2.5
Services	An AUTOSAR Service is a logical entity of the basic software offering general functionality to be used by various AUTOSAR software components [1].	Virtual subsystem	2.2.6
Runnable	A Runnable Entity is a part of an Atomic Software-Component which can be executed and scheduled independently from the other Runnable Entities [1].	Function call subsystem	2.3
RTEEvents	An RTEEvent encompasses all possible situations that can trigger execution of a runnable entity by the RTE [1].	Function calls	2.3
Exclusive Areas	Exclusive Areas prevent runnables from being preempted by other runnables [4].	Atomic subsystem marked as ExclusiveArea.	2.3.5
Composition	Composition encapsulates a collaboration of Components thereby hiding detail and allowing the creation of higher abstraction levels [1].	Virtual subsystems	2.4
Datatypes	AUTOSAR datatypes are either primitive or complex they are used to type data-elements, arguments of the operations in a client-server interface and constants.	--	2.7.1
Primitive Datatype	All primitive datatypes allow an efficient mapping to programming languages like C	Simulink built-in types	2.7.1.1
Complex data types	Composite or complex datatypes are either arrays or records. An array consists of numberOfElements elements that each have the same type, arrays have zero based indexing. A record describes a non-empty set of objects, each of which has a unique identifier.	Simulink wide signal for arrays. Simulink bus signal for records.	2.7.1.2
Characteristics	Values of characteristics can be changed on an ECU via calibration data management tool or an offline calibration tool	Overloaded Simulink.Parameter class suitable for online calibration.	2.7.2

Table 1: Mapping of AUTOSAR Concepts to Simulink Concepts

2.2 AtomicSoftwareComponentType

In AUTOSAR, application software is organized in independent units, called software components. Such components hide the implementation of the functionality and behavior they provide and simply expose well defined connection points called ports.

In this section, we examine how these software component fundamental meta-classes can be expressed in Simulink. This section forms the framework for the next section where the lower level Runnables (`RunnableEntity`), which are (at least indirectly) a subject for scheduling by the underlying AUTOSAR operating system, are expressed in Simulink. This section also forms the framework for the subsequent section for expressing the higher level AUTOSAR compositions, which allows for encapsulation of functionality by aggregating existing software components, in Simulink.

On the level of abstraction of software components, `ComponentType`, `AtomicSoftwareComponentType` and `ComponentPrototype` are relevant. The meta-class `ComponentType` is an abstract class and as such is not instantiable. Conversely the meta-class `ComponentPrototype` assigns a certain role to a component type (eg. a windshield wiper component is assigned the left wiper role and another one the right wiper role). What is salient in behavioral modeling terms is the `AtomicSoftwareComponentType` which is the only meta-class to support an implementation and hence runnables.

`AtomicSoftwareComponentType` is interesting from a Simulink point of view, perhaps largely due to the prefix atomic, not referring to atomicity in terms of the atomic execution of Simulink subsystems, but to the fact that this software component cannot be decomposed to run on more than one ECU and is to be mapped to one AUTOSAR ECU only.

2.2.1 Canonical Pattern

The canonical pattern for modeling atomic software components in Simulink is shown in Figure 1.

- An atomic software component type may be modeled as a virtual/non-virtual subsystem, library block or model. Figure 1 shows atomic software components as virtual subsystems.
- The first inport should be used to route RTEEvents to the software component's runnables. Although there is no fundamental reason why all RTEEvents cannot be routed through one RTEEvent bus and selected within the software component, generally it is preferred to separate the function calls of the software components, by bus selectors and creators.
- The rest of the inports (R-Ports) and all of the outports (P-Ports) are responsible to fulfill the R-Port and P-Port sender/receiver communication of the software component.
-

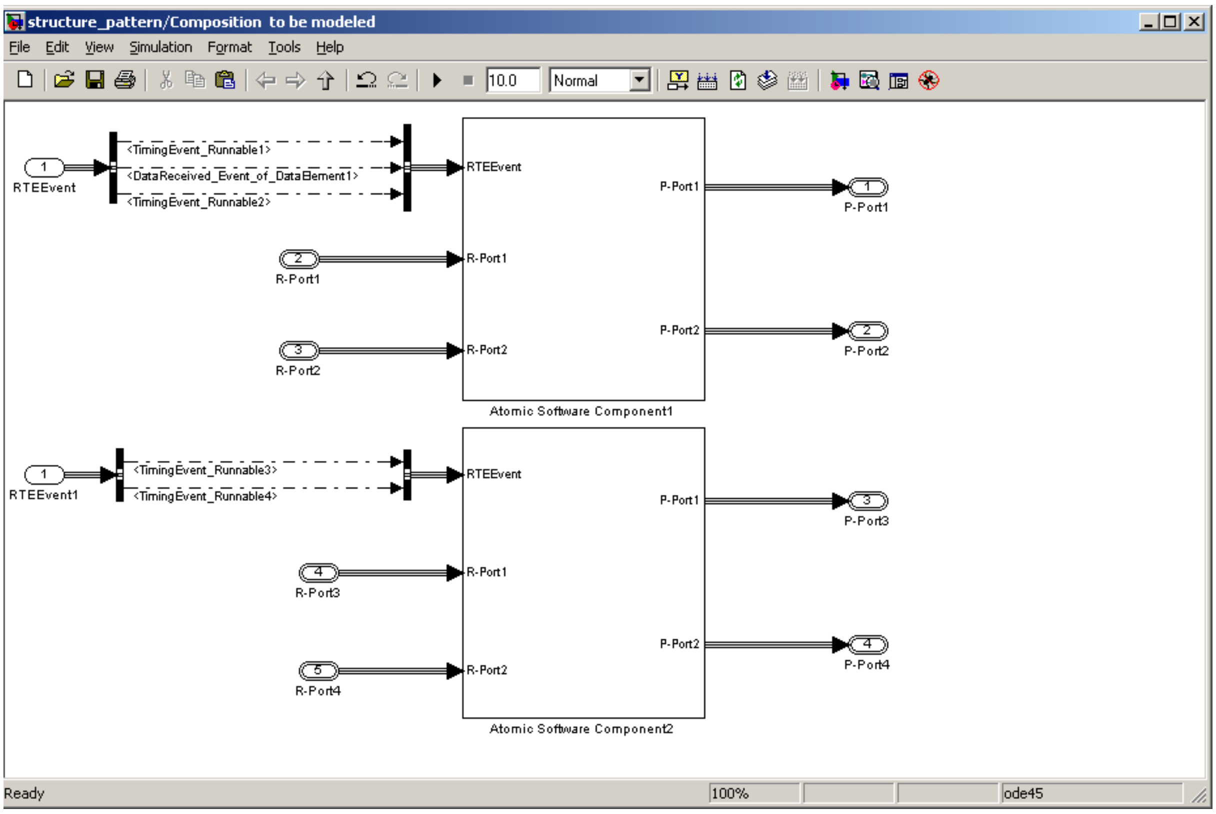


Figure 1: Canonical Pattern for Software Components

2.2.1.1 Current Limitations

Simulink version (R2006a) [15]only supports the propagation of a single function to a model reference block, thus if an atomic software component requires more than one RTEEvent (represented as a function call, see Figure 8) a model should not be used to represent an atomic software component.

2.2.2 AUTOSAR Ports

As mentioned in the introduction to this section, AUTOSAR software components have well defined interaction points, called ports, to describe the possible kinds of communication with other software components. This concept should be very familiar to users of Simulink:

- An AUTOSAR receive port (R-Port), or R-PortPrototype for sender/receiver communication corresponds to a Simulink inport.
- An AUTOSAR provide port (P-Port), or P-PortPrototype for sender/receiver communication corresponds to a Simulink outport.
- The data elements required or provided by these ports are defined by a `Sender-ReceiverInterface`. This type of interface however only describes structure and does not provide information about whether communication needs to be done reliably, or whether an initial value exists. This is covered by the communication specification or ComSpec classes.

2.2.3 Sender/Receiver Interface (SenderReceiverInterface)

A port interface defines which information can be exchanged between the ports of the components by formally describing the names and signatures of data elements exchanged between components.
A `SenderReceiverInterface` behaves sufficiently similar to a Simulink BusObject for this to be the recommended way to express this mechanism in Simulink.
The individual elements that are described by a `SenderReceiverInterface` are called data elements (`DataElementPrototype`) which has a name and a datatype, which could be a predefined AUTOSAR datatype or a user-defined datatype.
For arguments of extensibility AUTOSAR ports do not necessarily have to refer to the same `SenderReceiverInterface` in order to be connected. Rather the port interfaces must be "compatible". These compatibility rules are discussed in depth in [5] and the salient features are restated here: Two `PortInterfaces` are compatible if:

- The data elements datatypes are the same or aliases.
- The data elements have the same name.
- For each data element that is required, there exists one compatible data element that is provided.

Interfaces describe the static structure of this exchanged information, while the communication relevant dynamic attributes are attached to ports by communication attributes. The SenderReceiverInterface aggregates DataElementPrototypes.
Figure 2 shows the recommended use of BusObject and bus selector/creators in Simulink to represent port interfaces. The p-port1 of the sender subsystem is validated against busObject Interface_a. The r-port1 of the receiver subsystem is validated against busObject Interface_c. A bus selector has been used to show which data has been provided but in practice this is a redundant block, however the bus creator is needed to "support" AUTOSAR's port interface compatibility rules and allow the receiving subsystem to receive a subset of the data provided by the sender subsystem. R-Port2 requires the same data elements (not necessarily referencing the same port interface) as those being provided by P-Port2 hence no bus selector is needed.

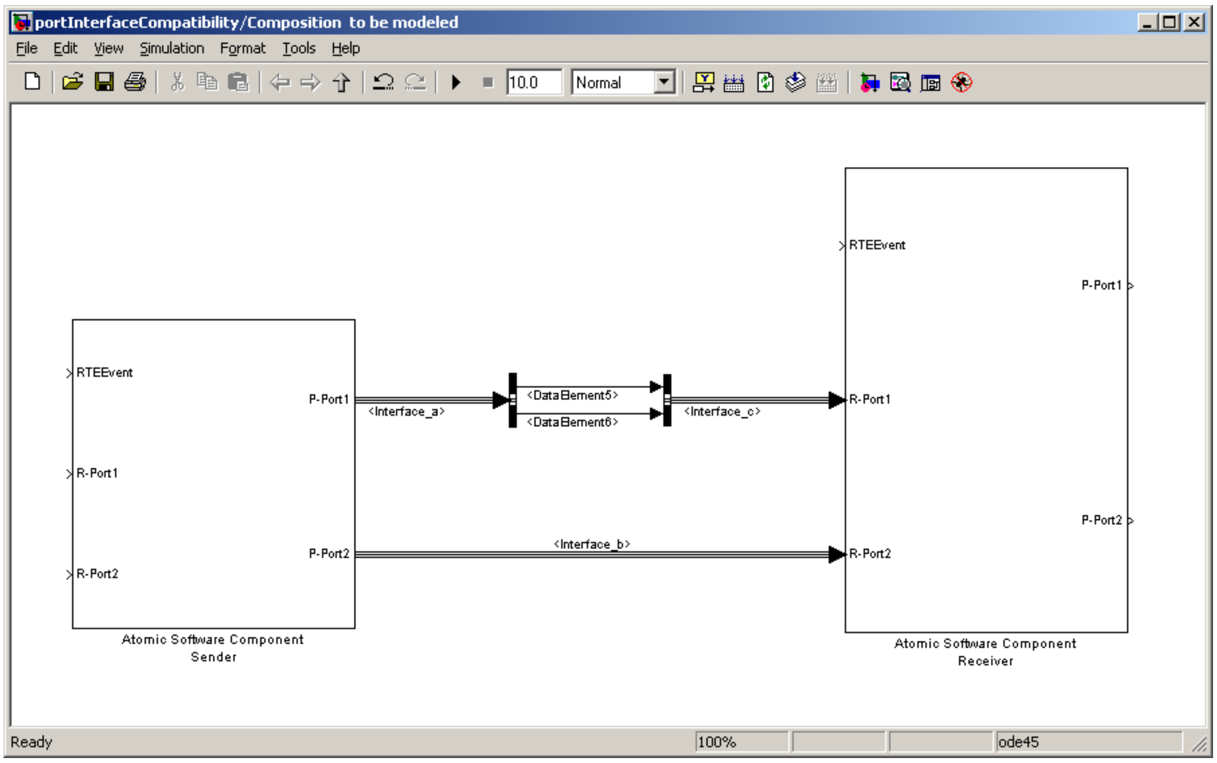


Figure 2: Respecting AUTOSAR's interface compatibility rules using a bus creator (not strictly needed) and a bus selector block.

2.2.4 Client/Server Interface (ClientServerInterface)

For Client/Server communication, the client initiates the communication, requesting that the server perform a service, transferring a parameter set if necessary. The client may be blocked (synchronous communication) or non-blocked (asynchronous communication) until the response from the server is received.

Client/Server among atomic software components is considered out of scope for this document; however, we do consider client/server communication between:

- atomic software components (client) and AUTOSAR services and;
- sensor actuator software components (client) and the ECU abstraction.

The interfaces of AUTOSAR services and the ECU abstraction are slightly different than software component interfaces in that they have been standardized. This standardization obviates the need to explicitly connect the software component to the standardized entity.

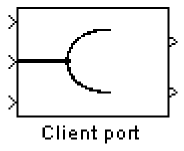


Figure 3: A client port block within Simulink which will reside in the runnables.

Figure 3 shows an example of how the client port would look for a software component communicating with a service or the ECU abstraction. This port will reside in the runnable and will be either a masked s-function or a Stateflow chart (See Section 3.3). The operation signatures for both AUTOSAR services and ECU abstractions is standardized so the user may wish to create a unique client port block for each operation offered to the software component runnable.

2.2.5 Sensor Actuator Software Component

The sensor actuator software components offer a physical (real world) unit interface (eg. engine speed in kph) of sensor/actuator values to the application software components. Sensor/actuator components will have an implementation, say providing simple filtering, however an appropriate implementation might be delivered together with the sensor/actuator or more easily built manually. Hence the focus in this section is on simulation rather than code generation or the unification of these two activities.

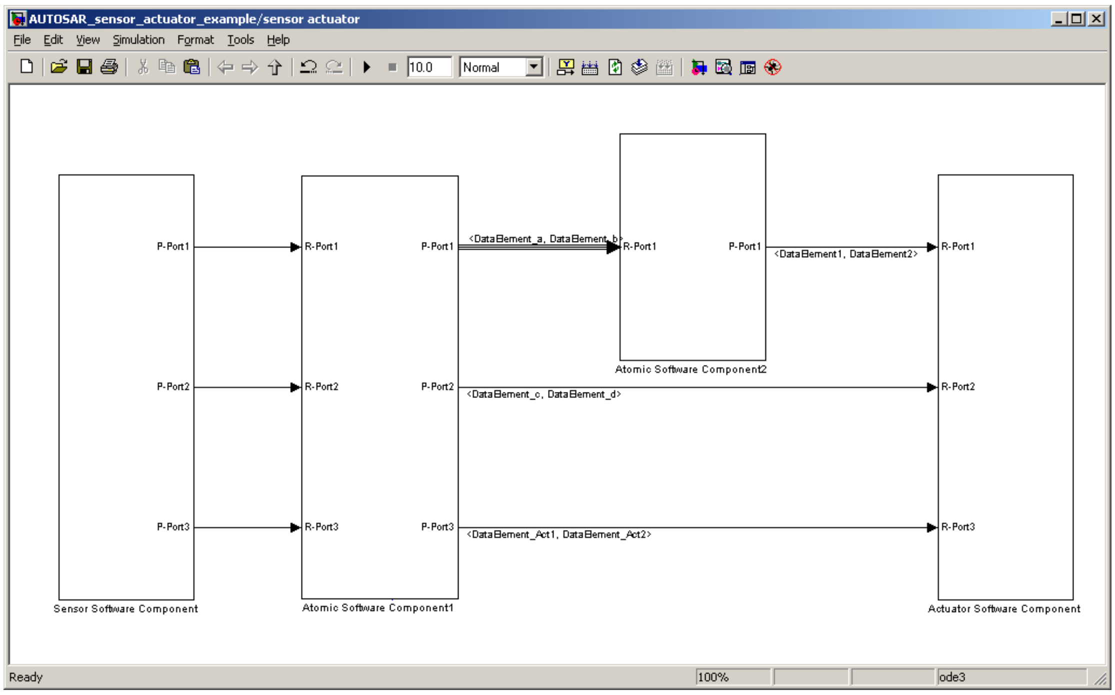

AUTOSAR_sensor_actuator_example/sensor actuator
File Edit View Simulation Format Tools Help
10.0
Normal
P-Port1
P-Port2
P-Port3
Sensor Software Component
R-Port1
R-Port2
R-Port3
P-Port1
P-Port2
P-Port3
Atomic Software Component1
<DataElement_a, DataElement_b>
R-Port1
P-Port1
Atomic Software Component2
<DataElement1, DataElement2>
<DataElement_c, DataElement_d>
<DataElement_Act1, DataElement_Act2>
R-Port1
R-Port2
R-Port3
Actuator Software Component
Ready
100%
ode3

Figure 4: Sensor and Actuator Software Components communicating with application software components

2.2.5.1 Simulation

The sensor/actuator components are convenient placeholders for connecting various plant models to the overall system for simulation. The signals offered by these components are in physical (real-world) units and as such can generally be connected to various plant models, for system wide simulation, with little/no unit conversion. Figure 5 shows an example of using the sensor/actuator software components as simulation placeholders to model sensor/actuator dynamics and to offer a connection interface to a plant model. What is interesting to note is that the sensor/actuator dynamics would normally consist of two components: 1. electrical dynamics between the ECU abstraction layer and the electrical signal offered by the real µC peripheral layer and 2. the electro/mechanical dynamics of the sensor/actuator. The system's engineer will have to use his/her judgment as to the level of fidelity needed. Figure 6 shows an example of some simple sensor/actuator dynamics that may be used. The sample rate of the blocks would generally reflect the cyclic activation or rate of the sensor/actuator runnables.

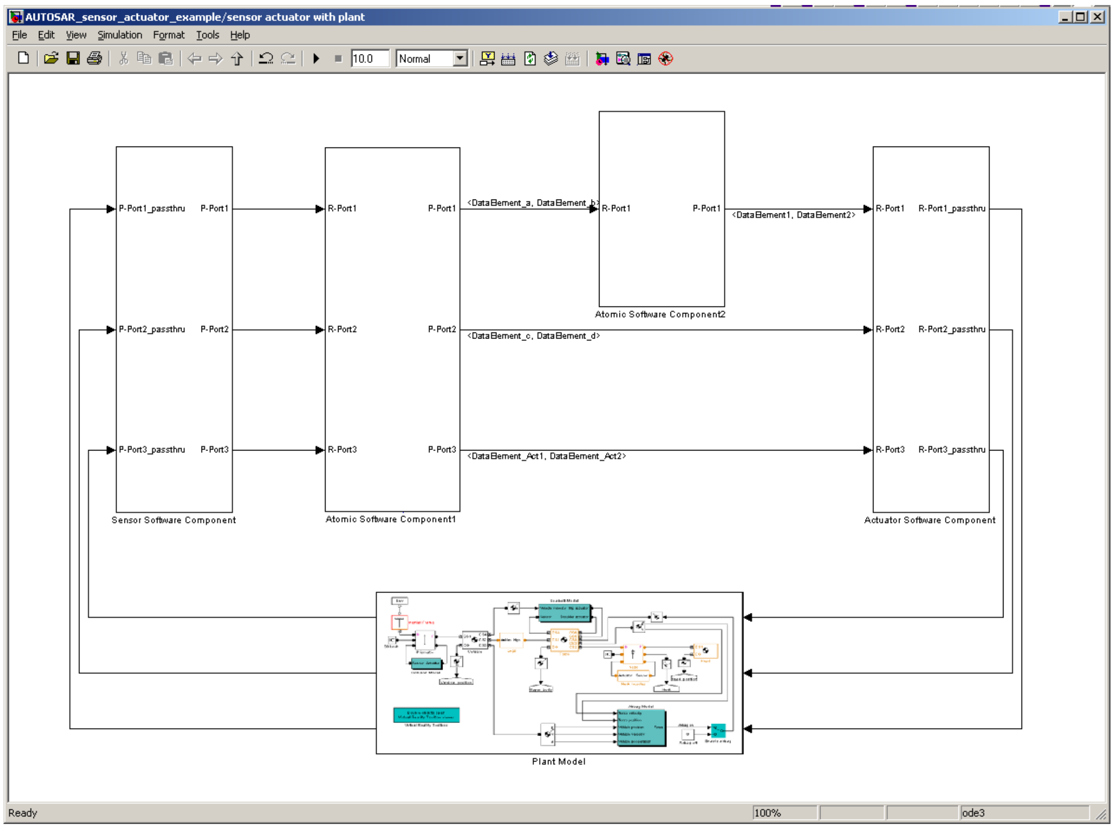

AUTOSAR_sensor_actuator_example/sensor actuator with plant
File Edit View Simulation Format Tools Help
10.0
Normal
P-Port1_passthru
P-Port1
P-Port2_passthru
P-Port2
P-Port3_passthru
P-Port3
Sensor Software Component
R-Port1
R-Port2
R-Port3
Atomic Software Component1
<DataElement_a, DataElement_b>
<DataElement_c, DataElement_d>
<DataElement_Act1, DataElement_Act2>
Atomic Software Component2
<DataElement1, DataElement2>
R-Port1_passthru
R-Port2_passthru
R-Port3_passthru
Actuator Software Component
Plant Model
Ready
100%
ode3

Figure 5: Application components connected to a plant model

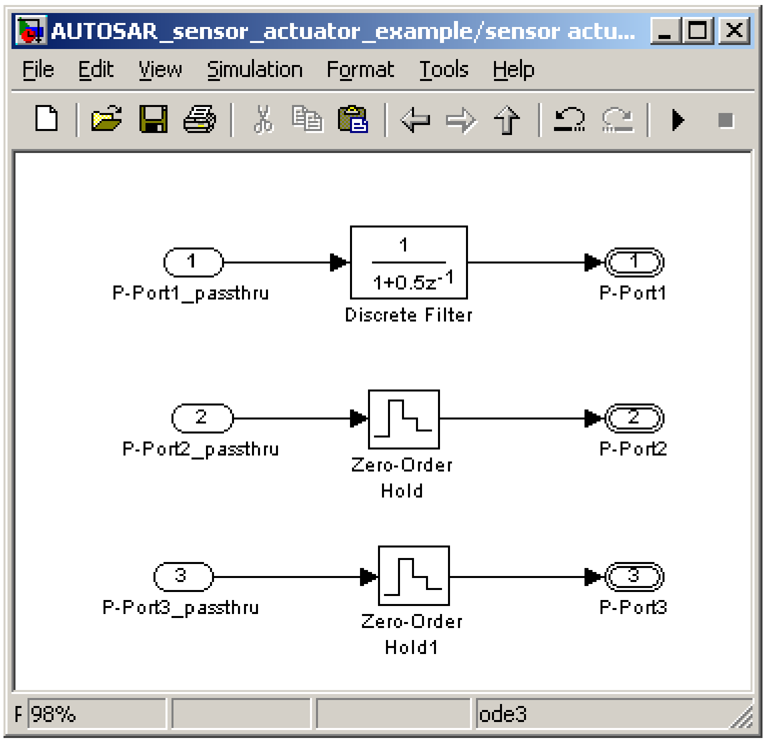


Figure 6: Simple Sensor dynamics

2.2.5.2 Code Generation

A user may wish to create an implementation for a sensor/actuator software component this is addressed in Section 2.3.6.1.

2.2.6 Services

Some functions that are utilized by a software component are not implemented by other application level components but are provided by the runtime environment and are referred to as AUTOSAR services [7]. Typical examples include state management, memory management, synchronization services.
The flag `isService` provided by the class `PortInterface` specifies that the service is provided by the RTE and the connection does not need to be made explicit.
Internally to the software component, the `PortInterface` *still* defines the communication mechanism irrespective of whether the port is externally connected to a service or another software component and the RTE API call signature in the software component's runnables remains the same. The `PortInterface` description for the services has been standardized.
In general, services implement client/server interfaces as well as sender/receiver interfaces. Client/server interfaces for software components is considered out of scope for this document, however client/server interface for services will be discussed.

2.2.6.1 Simulation

Figure 7 shows a place holder for the simulation of an AUTOSAR service.

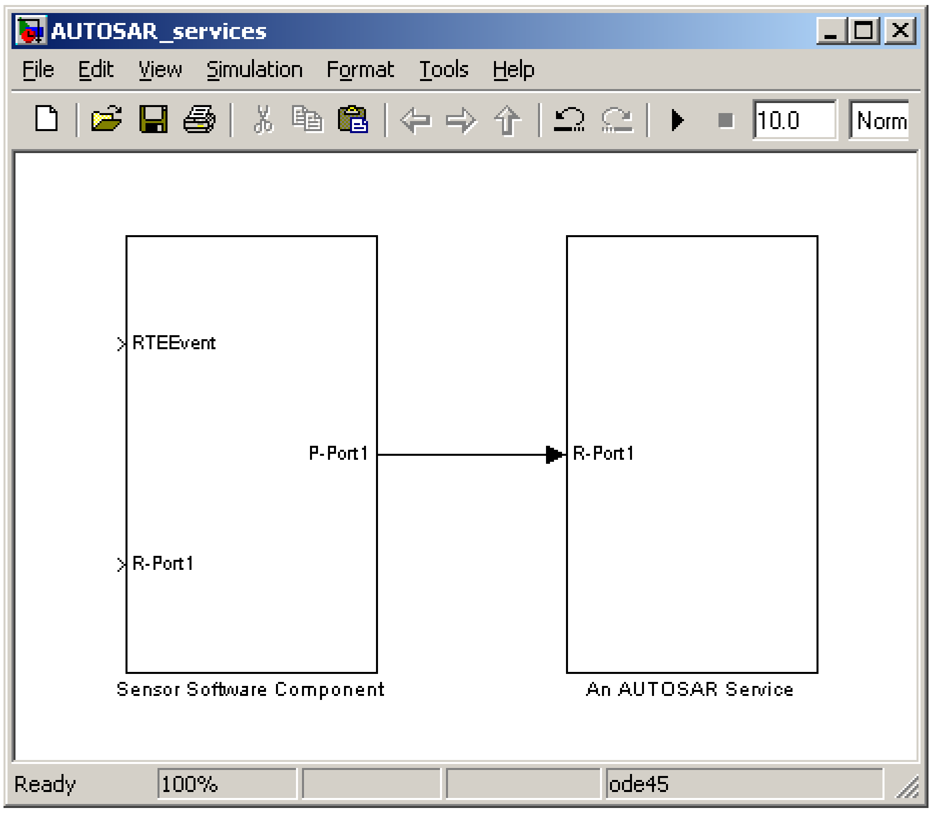


Figure 7: AUTOSAR Service

2.2.6.2 Code Generation

Generating code for software components that access services is addressed in Section 2.3.6.1.

2.3 Modeling Runnables Explicitly

A runnable (`RunnableEntity`) is part of an Atomic Software-Component which can be executed and scheduled independently from the other Runnable Entities. The categorization is summarized in **Table 2**. Runnables can be a target for code generation.

Table 2: Runnable Categorization

Runnable Categorization	***Summary***	***In scope***
Category 1A	Activated by the RTE using RTEEvents. Do not have WaitPoints and have to terminate in finite time. Only implicit communication.	✓
Category 1B	Activated by the RTE using RTEEvents. Do not have WaitPoints and have to terminate in finite time. Explicit and implicit communication.	✓
Category 2	Same as Category 1B plus can have at least one WaitPoint.	x

2.3.1 Canonical Pattern

The canonical pattern for modeling a runnable within Simulink is largely determined by the need for i) independent execution and ii) regular (cyclical) and irregular event activation predicating i) atomicity and ii) triggering of functionality. Figure 8 shows the Simulink representation.
A runnable may be cyclically triggered by a `TimingEvent` or triggered by other events that also subclass the meta-class `RTEEvent` (e.g. `DataReceivedEvent`) which could be posted asynchronously. For generality the runnable should be represented as a function-call subsystem where the type of function-call corresponds to the type of the event that is triggering it; e.g. cyclical or asynchronous.
Runnables for sender-receiver communication can access the individual data elements in the components r and p-ports.
These general observations lead to the following design decisions:

- A Runnable is represented by a function-call subsystem.
- Individual data elements should be routed to/from the function-call subsystem via signals.
- For clarity, the function calls for the different Runnables are combined by using a function-call bus, which is routed via inport 1.

- Function calls should be selected using the bus selector block and routed to the different Runnables. More than one function call may be passed to the function-call subsystem by a bus of function calls (see Runnable_simple in Figure 8).
  - Events can be generated using function-call generator blocks or Stateflow.

Please Note: that the top-level subsystem of an AUTOSAR software component (see Figure 8) contains only runnables and virtual blocks.

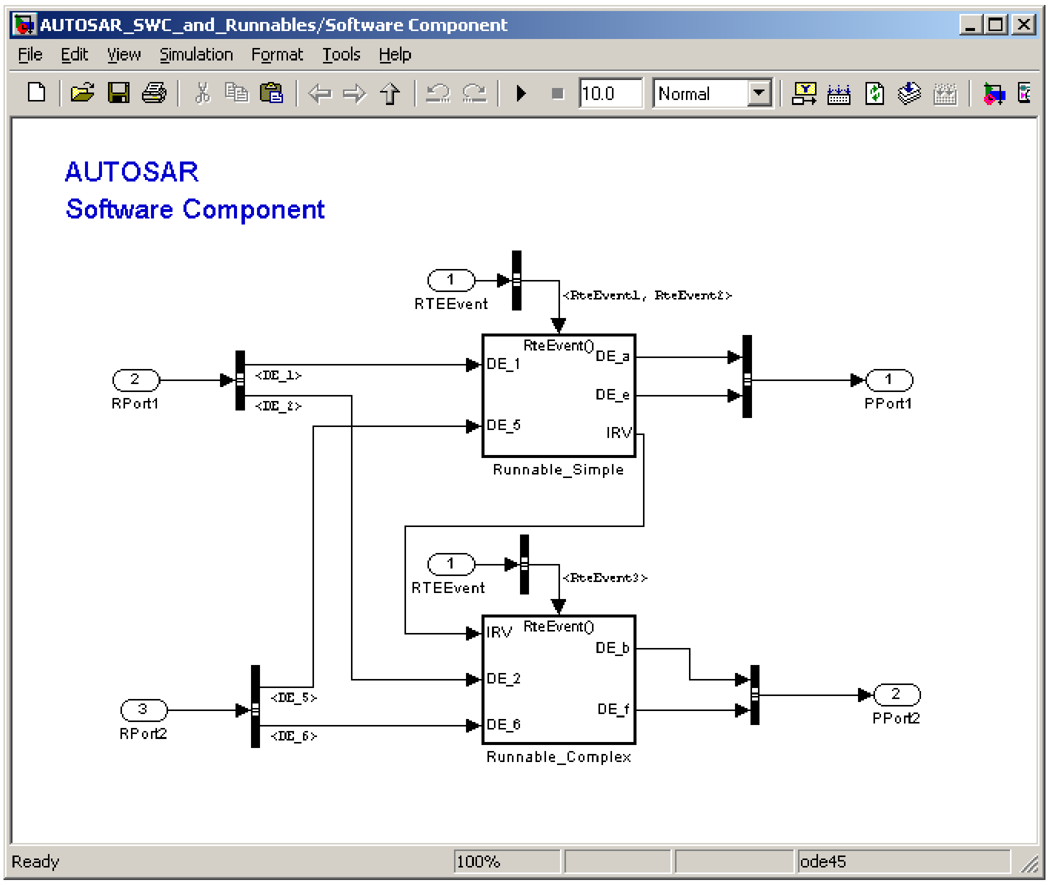


Figure 8: Canonical Pattern for Runnables. Runnable_Simple is activated by multiple RTEEvents.

2.3.2 Code Generation Layout

Code generated from Simulink and a code generation tool must conform to the APIs set out in the RTE software specification [2]. An RTE generation tool will generate the header file and the Simulink code generator will be responsible for generating the corresponding source file. Here we show an example of some generated code to out-line the calling sequence and the function signatures that will be discussed through-out this section:

```
void RSimple( Rte_Instance self )
{
      uint16 indata1, indata2, indata3;
      uint16 outdata1, outdata2, outdata3;
      uint8 invalid;
// copy data from RTE to Simulink
/*     rte_status = Rte_IRead_<re>_<p>_<d>( self, <data> );
      <re> = RSimple, Short Name of RunnableEntity instance
      <p>  = RP1, Short Name of PortPrototype instance
      <d>  = DE1, Short Name of Data Element instance, referenced by port
instance
*/

      Rte_IRead_RSimple_RP1_DE1( self, &indata1 );
      Rte_IRead_RSimple_RP1_DE2( self, &indata2 );
      Rte_IRead_RSimple_RP1_DE3( self, &indata3 );


      // Simulink implementation e.g.
      outdata1 = indata1 + indata2+ indata3 + Rte_CData_PARAMA;
      outdata2 = indata1 + indata2;
      outdata3 = indata3;


/*     return = Rte_CData_<name>( self );
      <name> = param1, Short Name of characteristic
*/

// copy data from Simulink to RTE
/*     rte_status = Rte_IWrite_<re>_<p>_<d>( self, <data> );
      <re> = RSimple, Short Name of RunnableEntity instance
      <p>  = PPa, Short Name of PortPrototype instance
      <d>  = DEa, Short Name of Data Element instance, referenced by port
instance
*/

      Rte_IWrite_RSimple_PPa_DEa( self, outdata1 );
      Rte_IWrite_RSimple_PPa_DEb( self, outdata2 );
      Rte_Write_PPa_DEc( self, outdata3 );


// could invalid explicit data transmission if we want
invalid = 1;
if(invalid){
/*     Rte_Invalidate_<p>_<o>( self);
      <p>  = PP1, Short Name of PortPrototype instance
      <d>  = EB1, Short Name of Data Element instance, referenced by port
instance
*/


      Rte_Invalidate_PPa_DEc( self );
      }
}
```

2.3.3 Communication Modes of Sender-Receiver Communication

Figure 9 shows an example of a runnable with some functional behavior (highlighted in yellow). The data for the function is passed into and out of the function by sender-receiver communication over the RTE (inports 1/2 and outports 1/2) and also via an inter-runnable variable (outport 3). For sender-receiver communication there are two modes of operation:

- **Implicit:** The RTE automatically reads a specified set of data elements before a runnable is invoked and automatically writes (a different) set of data elements after the runnable entity has terminated. This mode is termed “implicit” since the runnable takes no explicit action to inform the RTE to read or write data.
- **Explicit:** A component uses explicit RTE API calls to send and receive data elements before the runnable has terminated.

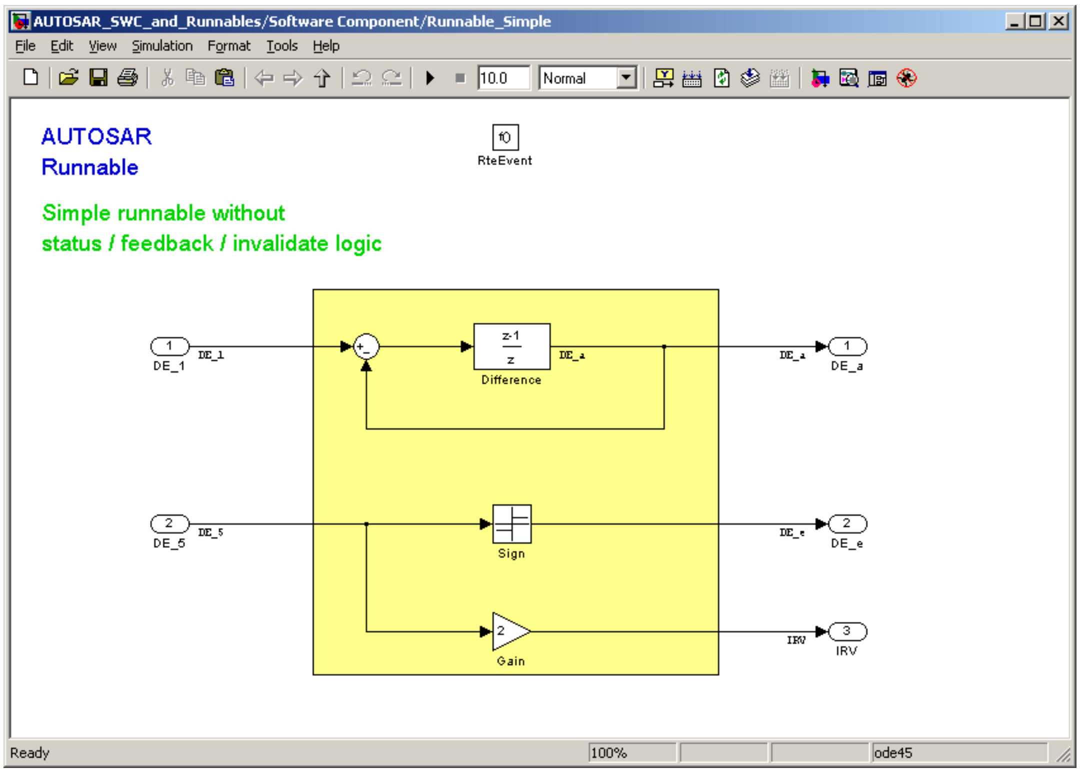


Figure 9: The Inside of a runnable which implements some functional behavior (highlighted in yellow). Data is passed into and out of the runnable by sender-receiver communication (inports/outports, 1 and 2) and via an inter-runnable variable (outport 3).

2.3.3.1 Code Generation

The following code signatures need to be generated for each data element of the runnable depending on whether it is a receive port or provide port and also whether communication is achieved implicitly or explicitly:

Concept	*RTE API*
implicit data reception	`rte_status = Rte_IRead_<re>_<p>_<d>( self, <data> );`
explicit data reception	`rte_status = Rte_Read_<p>_<d>( self, <data> );`
implicit data transmission	`rte_status = Rte_IWrite_<re>_<p>_<d>( self, <data> );`
explicit data transmission	`rte_status = Rte_Write_<p>_<d>( self, <data> );`

The RTE API access can be realized either using an overall subsystem configuration block, custom storage classes (CSC) attached to the input and output signals or custom blocks attached to the inports and outports.

2.3.3.2 Simulation

The code generation artifacts should not affect simulation.

2.3.4 Inter Runnable Communication

Inter runnable communication is handled by inter runnable variables (See port 3 in Figure 9). These variables can be read and written by all runnables of an atomic software component. Akin to sender-receiver communication access can either be implicit or explicit.

2.3.4.1 Code Generation

Concept	*RTE API*
implicit IRV reception	`rte_status = Rte_IrvIRead_<re>_<p>_<d>( self, <data> );`
explicit IRV reception	`rte_status = Rte_IrvRead_<p>_<d>( self, <data> );`
implicit IRV transmission	`rte_status = Rte_IrvIWrite_<re>_<p>_<d>( self, <data> );`
explicit IRV transmission	`rte_status = Rte_IrvWrite_<p>_<d>( self, <data> );`

Again the RTE API access can be realized either using an overall subsystem configuration block, custom storage classes (CSC) attached to the input and output signals or custom blocks attached to the inports and outports.

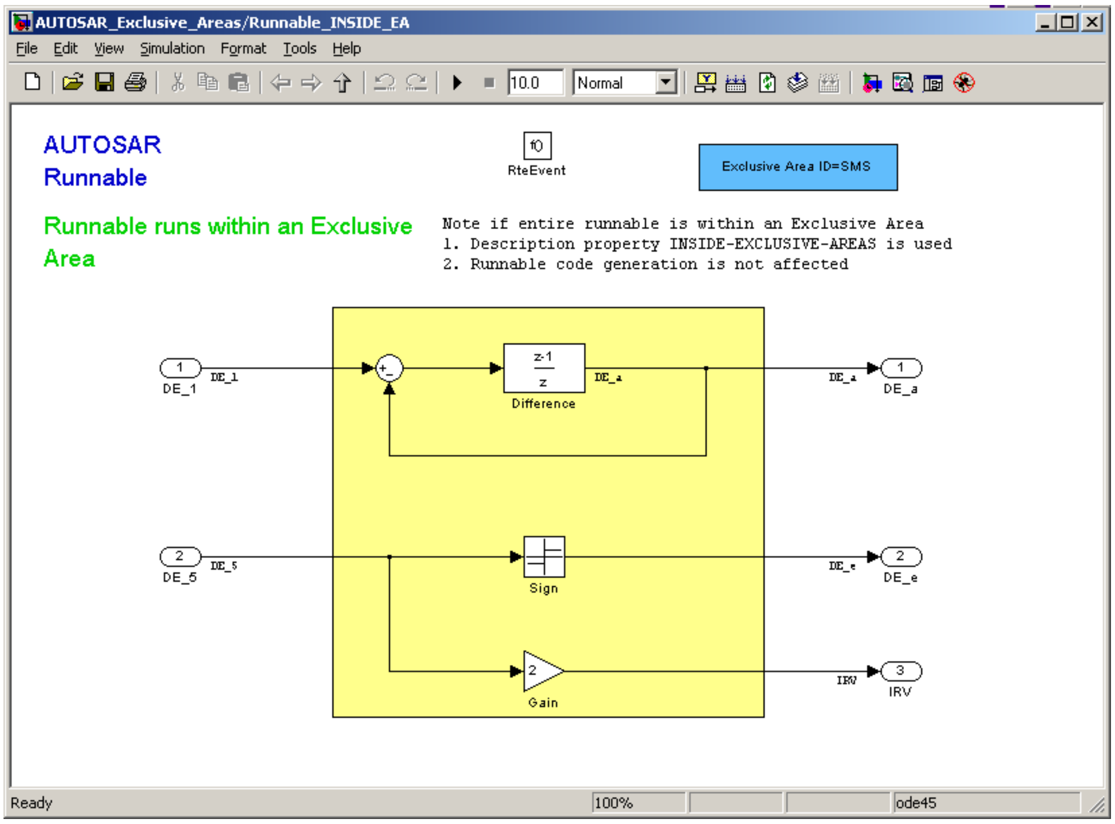
2.3.4.2 Simulation

Again the code generation artifacts should not affect simulation. Also it should be noted that since the simulation of a runnable is atomic and can not be interrupted by an additional parallel operation e.g. of an additional runnable the consequences of direct explicit communication between two runnables cannot be observed during simulation.

2.3.5 Exclusive Areas

An exclusive area defines a region of functionality within which concurrent data access is blocked by the RTE.

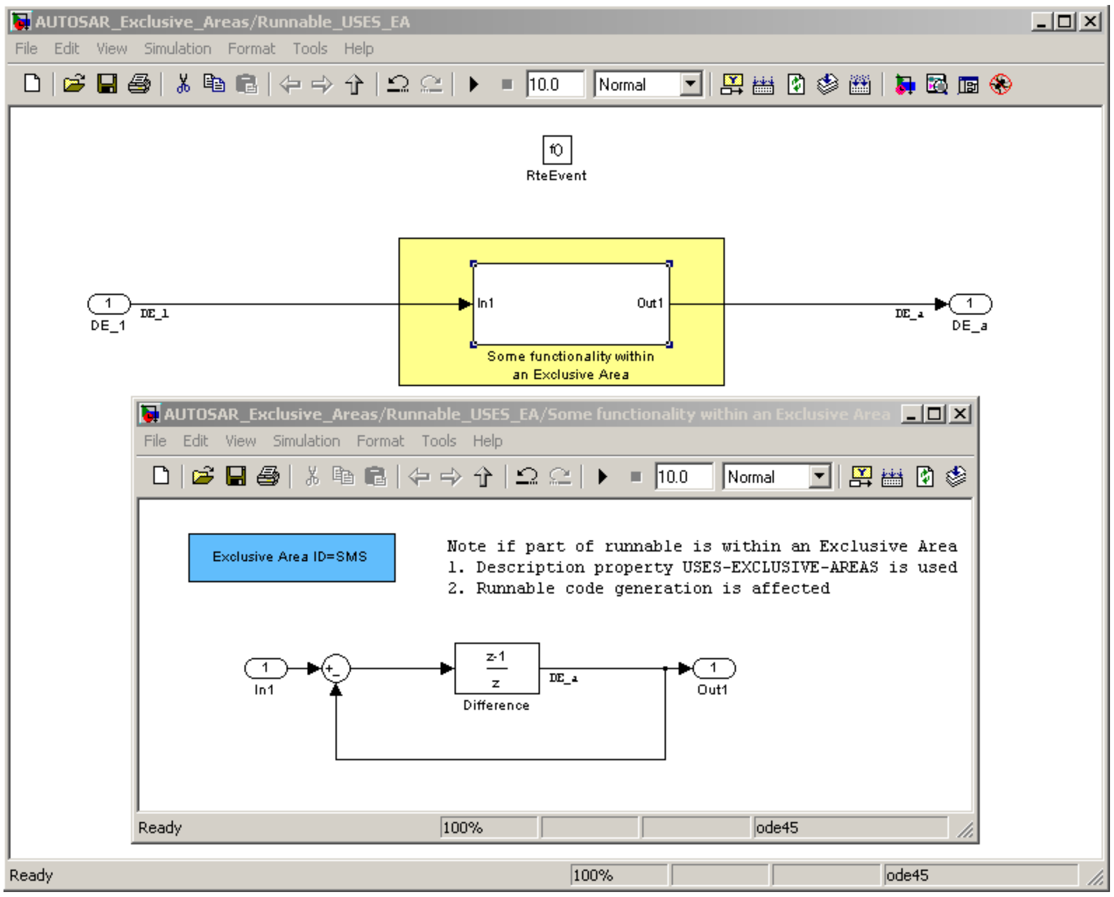
For a runnable that runs entirely within an exclusive area the top-level runnable-subsystem should be tagged as an exclusive area. This tagging only affects the description of the software component (meta model property INSIDE-EXCLUSIVE-AREAS) and not code generation (Figure 10).



```
Note if entire runnable is within an Exclusive Area
1. Description property INSIDE-EXCLUSIVE-AREAS is used
2. Runnable code generation is not affected
```

Figure 10: Entire runnable is within an Exclusive Area

If the runnable internally contains parts which should be protected by an exclusive area (meta model property USES-EXCLUSIVE-AREAS), this corresponds to nested Simulink subsystems within the runnable which should be tagged as exclusive areas (Figure 11).


AUTOSAR_Exclusive_Areas/Runnable_USES_EA
File Edit View Simulation Format Tools Help
10.0
Normal
f()
RteEvent
1
DE_1
DE_1
In1
Out1
Some functionality within an Exclusive Area
DE_a
1
DE_a
AUTOSAR_Exclusive_Areas/Runnable_USES_EA/Some functionality within an Exclusive Area
Exclusive Area ID=SMS
1
In1
+
-
z-1
z
Difference
DE_a
1
Out1
Ready
100%
ode45

Note if part of runnable is within an Exclusive Area
1. Description property USES-EXCLUSIVE-AREAS is used
2. Runnable code generation is affected

Figure 11: Only part of the runnable is within an Exclusive Area.

2.3.5.1 Code Generation

Concept	*RTE API*
Entering an exclusive area	`Rte_Enter_<name>( self );`
Exiting an exclusive area	`Rte_Exit_<name>( self );`

2.3.6 Sensor Actuator Component Runnables

Sensor/actuator components are the only components that can access ports from the ECU abstraction below the RTE layer, these ports are automatically connected by the RTE generator rather than at the component authoring stage.

A sensor/actuator component will contain normal ports (e.g. any application software component can access) and ECU abstraction ports (The ECU Abstraction provides an interface to physical values of an ECU). As with application components that can access service ports the code generation mechanism for the runnable does not need to explicitly distinguish these two, as the RTE API calls port name and data element will cause the connection. An example of a sensor actuator component runnable is shown in Figure 12.

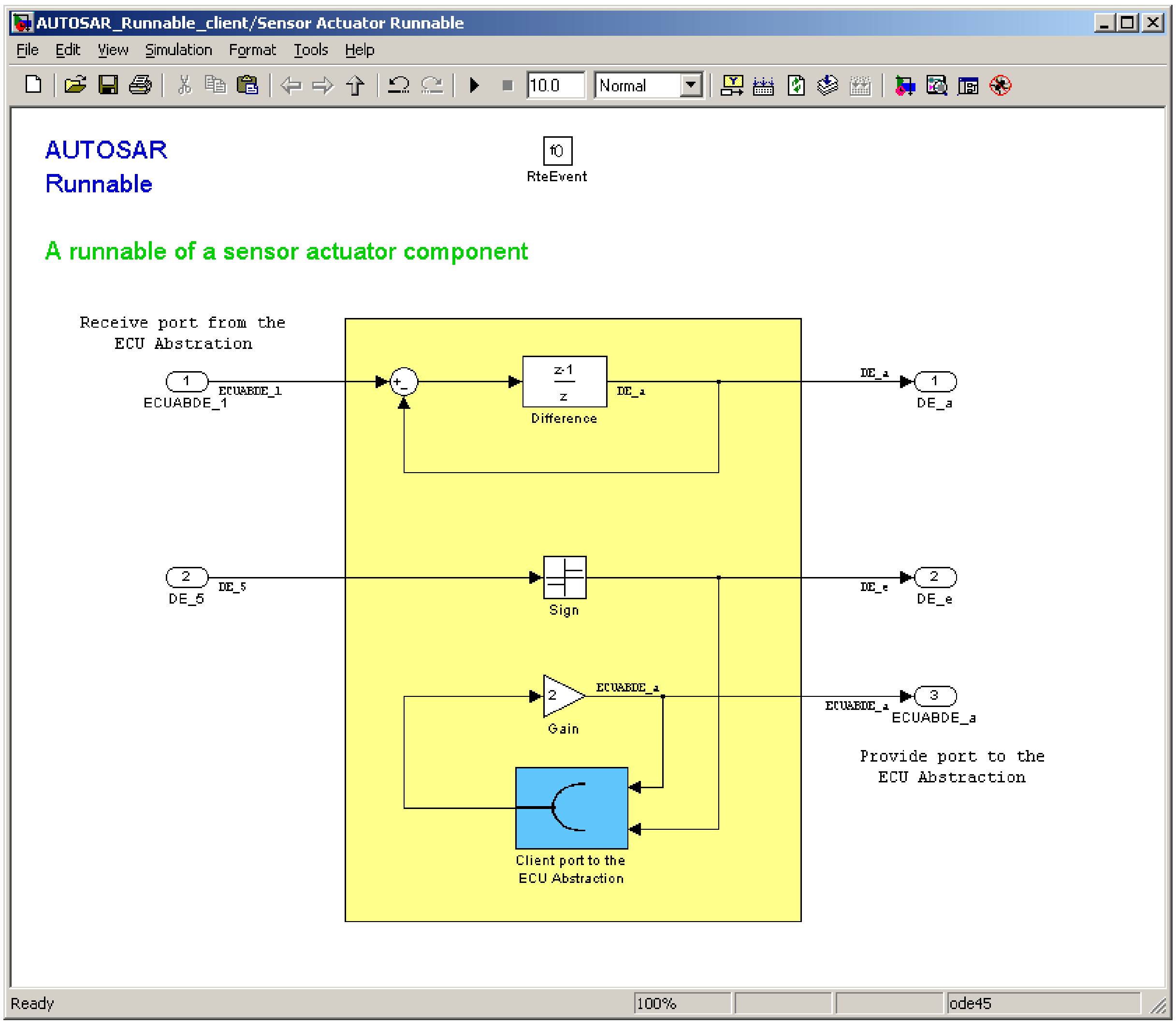


Figure 12: Inside the sensor/actuator runnable for code generation

2.3.6.1 Code Generation

Concept	***RTE API***
Invoke server operation	`rte_status = Rte_Call_<p>_<o>( self, <data_1>,...);`
Collect the result of an asynchronous client-server communication (client side)	`rte_status = Rte_Result_<p>_<o>( self, <param_1>,...);`

2.3.6.2 Simulation

Simulation aspects are covered in Section 2.2.5.1.

2.3.7 Runnables That Access Services

Figure 13 shows the layout of the runnable of the software component that utilizes an AUTOSAR service.

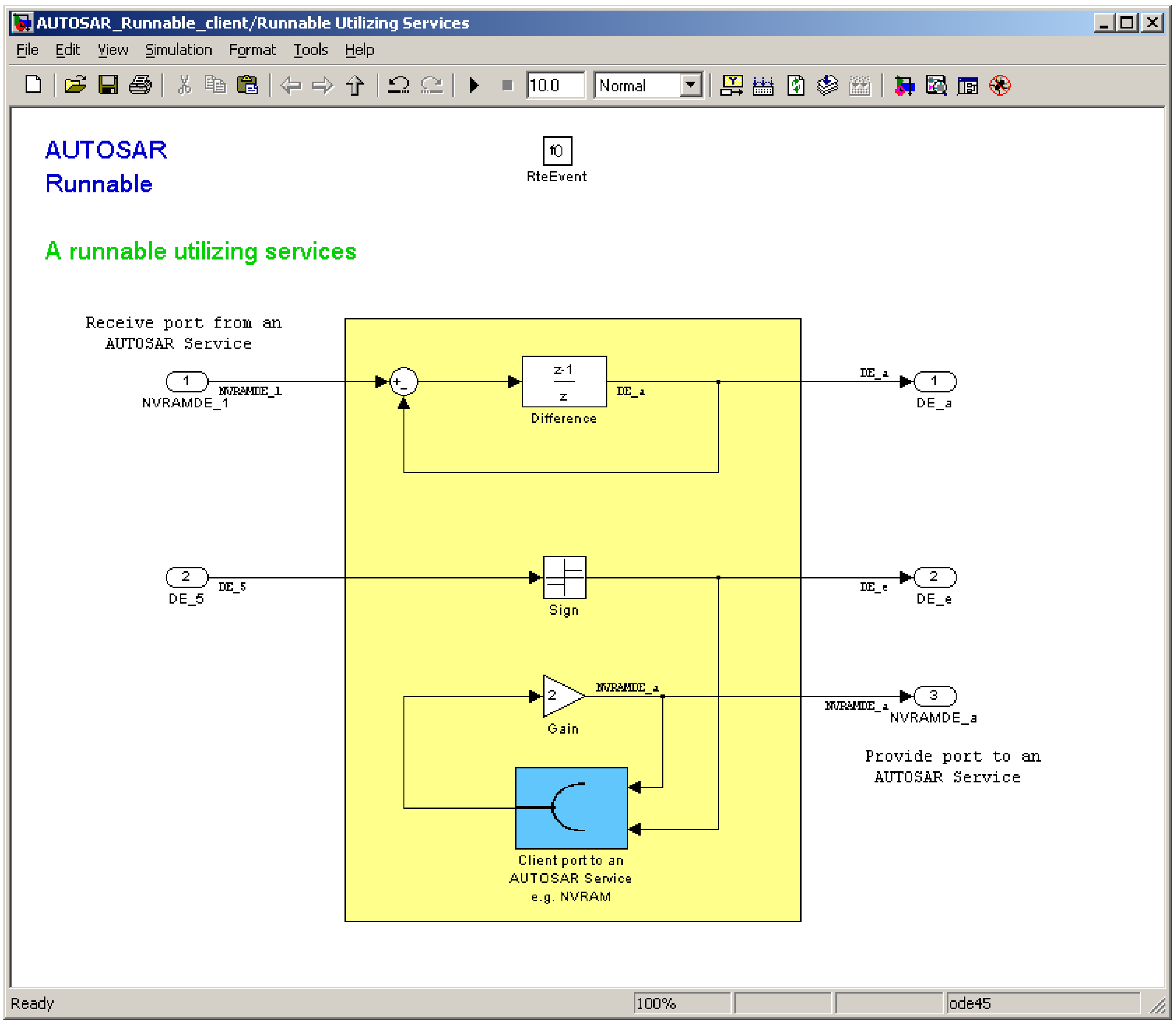

AUTOSAR_Runnable_client/Runnable Utilizing Services
File Edit View Simulation Format Tools Help
10.0
Normal
AUTOSAR
Runnable
f()
RteEvent
A runnable utilizing services
Receive port from an AUTOSAR Service
1
NVRAMDE_1
NVRAMDE_1
z-1
z
Difference
DE_a
DE_a
1
DE_a
2
DE_5
DE_5
Sign
DE_e
2
DE_e
2
Gain
NVRAMDE_a
NVRAMDE_a
3
NVRAMDE_a
Provide port to an AUTOSAR Service
Client port to an AUTOSAR Service e.g. NVRAM
Ready
100%
ode45

Figure 13: The runnable of a software component, utilizing AUTOSAR services.

2.3.8 Complex Runnables

Figure 9 shows an example of a runnable whose functional behavior does not depend on the error status of the communication between software components, Figure 14 in contrast, shows how this information might be offered to the user.

All data reception/transmission functions return a status value (of type Rte_Status) indicating whether the communication was successful or not. In addition when sending data the runnable can request that the RTE reports whether the data was received by the corresponding reception units successfully.

There are generally two approaches to offer this information to the author of the runnable, 1) augment the data signal with the additional information packed in a bus or 2) use a block approach (chosen for pedagogical reasons, see Figure 14).
As we have chosen to use the block approach we also cover annotations and the ComSpec issues in this section.

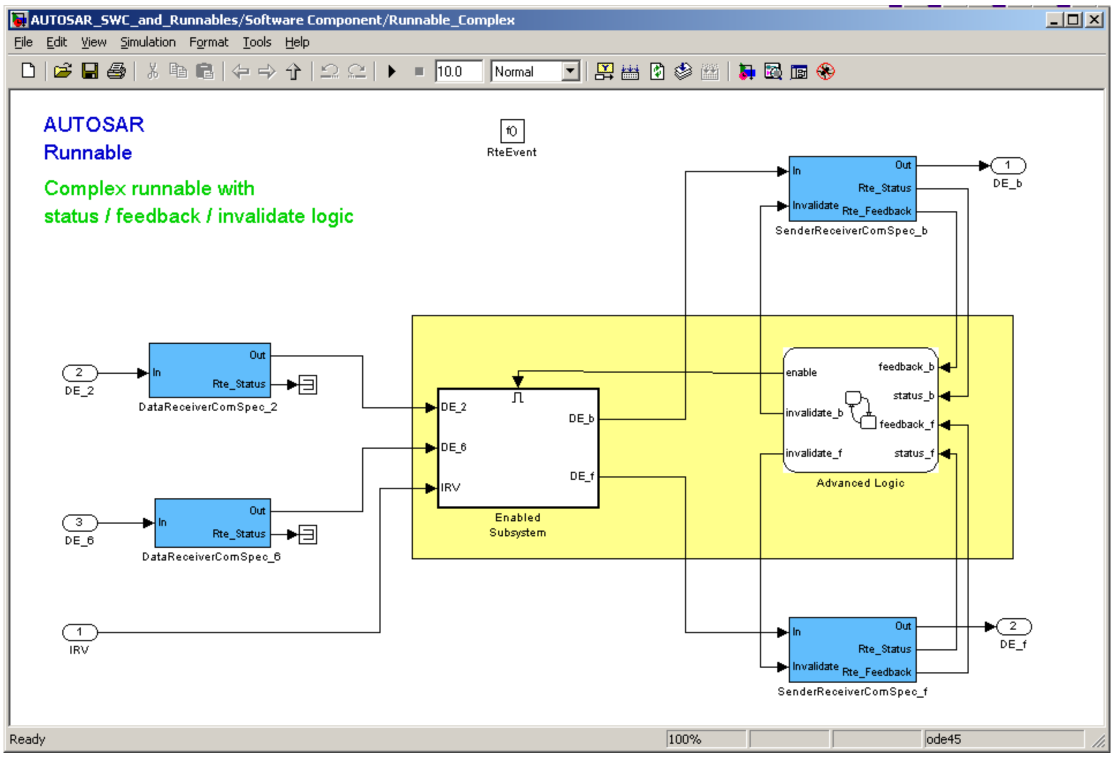
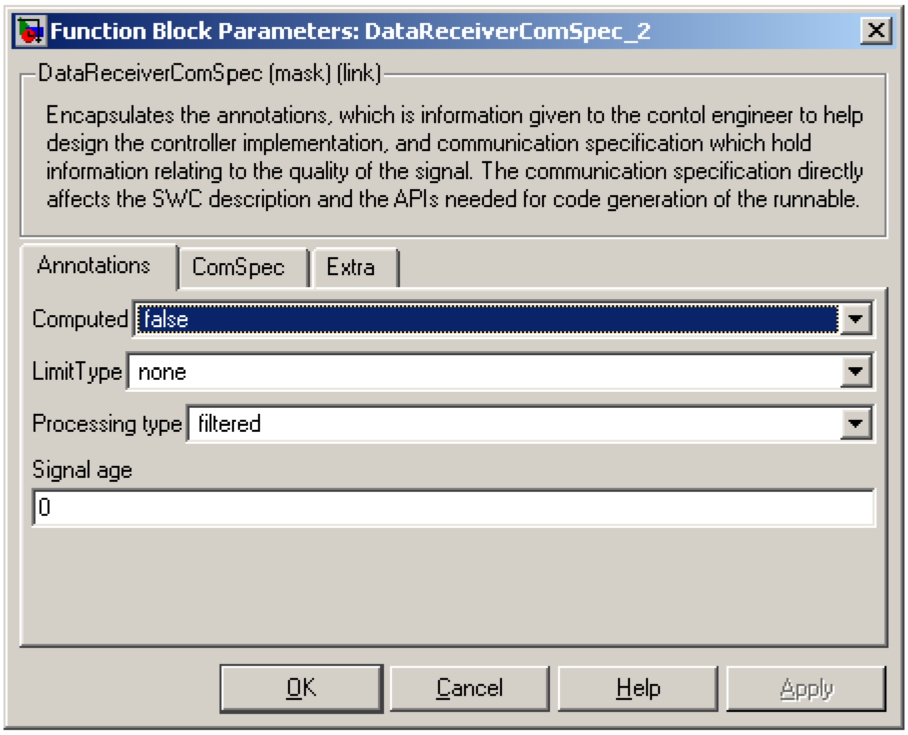


Figure 14: A complex runnable utilizing rte_status, rte_feedback and data invalidation information.

2.3.8.1 Annotations

The `SenderReceiverAnnotation` annotates data elements in a port that realizes a sender receiver interface. An annotation is information given to the control engineer to aid with the design of the controller implementation. For example it could be used to inform the control designer about the maximum allowed age of the signal since it was originally detected by a sensor 0[3]. SenderReceiverAnnotations are only commenting annotations and they do not affect either simulation or code generation. This information could be attached to a block holding both the annotations and communication specification information (Figure 15).



Function Block Parameters: DataReceiverComSpec_2

DataReceiverComSpec (mask) (link)

Encapsulates the annotations, which is information given to the contol engineer to help design the controller implementation, and communication specification which hold information relating to the quality of the signal. The communication specification directly affects the SWC description and the APIs needed for code generation of the runnable.

Computed: false

LimitType: none

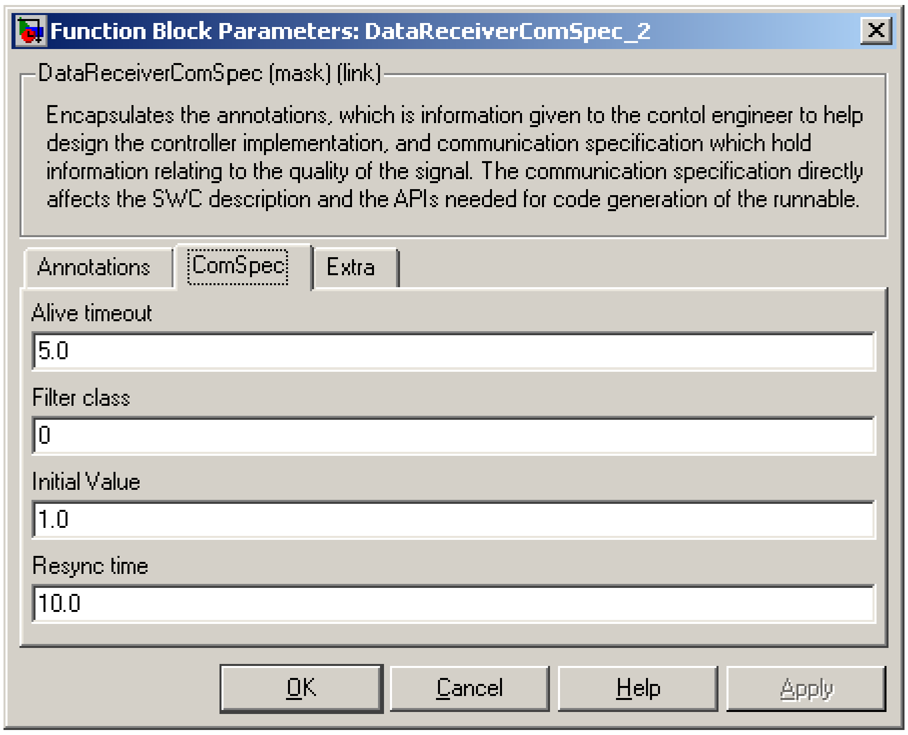
Processing type: filtered

Signal age: 0

Figure 15: The ComSpec block being used to store annotations.

2.3.9 Communication Specification – ComSpec

The communication specification (ComSpec) classes provide information relating to the quality of data communication. This information is applied to individual data elements and is different for sent and received data. A summary of the information is shown for receiver is show in Figure 16.and for sender in Figure 17.



Function Block Parameters: DataReceiverComSpec_2

DataReceiverComSpec (mask) (link)

Encapsulates the annotations, which is information given to the contol engineer to help design the controller implementation, and communication specification which hold information relating to the quality of the signal. The communication specification directly affects the SWC description and the APIs needed for code generation of the runnable.

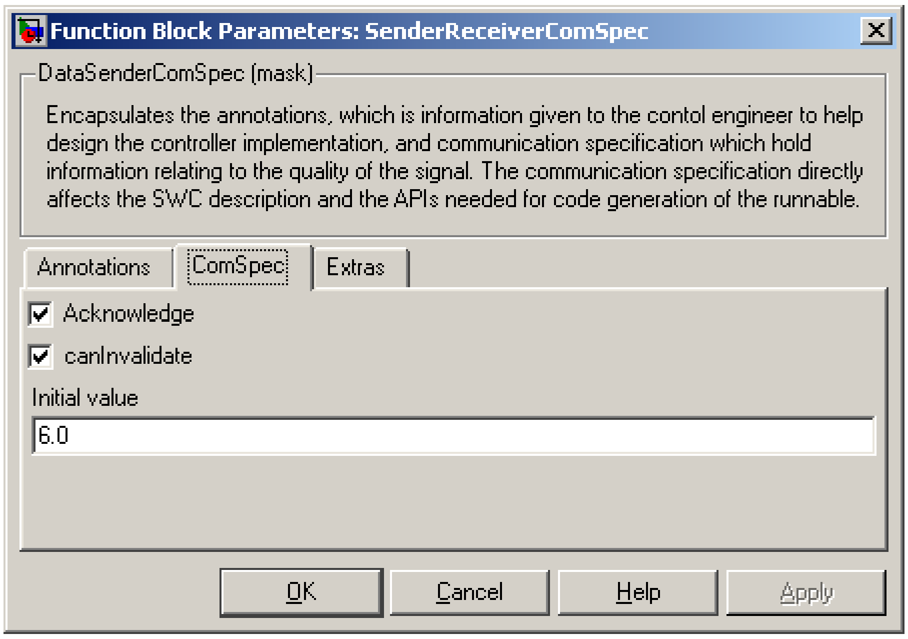
Alive timeout: 5.0

Filter class: 0

Initial Value: 1.0

Resync time: 10.0

Figure 16: DataReceiverComSpec parameters.



Function Block Parameters: SenderReceiverComSpec

DataSenderComSpec (mask)

Encapsulates the annotations, which is information given to the contol engineer to help design the controller implementation, and communication specification which hold information relating to the quality of the signal. The communication specification directly affects the SWC description and the APIs needed for code generation of the runnable.

✓ Acknowledge

✓ canInvalidate

Initial value: 6.0

Figure 17: DataSenderComSpec parameters.

2.3.9.1 ComSpec – initValue (Receive and Send)

The initValue defines the initial value for the data.

2.3.9.2 ComSpec – Acknowledgement Request and timeout (Send)

The AcknowledgementRequest provides access to an acknowledgement notification for a provided DataElementPrototype. The attribute timeout specifies the number of seconds before an error is reported.

Request Type	***Summary***
transmission	data reached the receiving port
reception	not mentioned in RTE spec

2.3.9.3 Code generation

For notification is accessed using the RTE API.

Concept	***RTE API***
acknowledgement notification for explicit data transmission	`rte_status = Rte_Feedback_<p>_<o>( self );`

2.3.9.4 Simulation

Simulation aspects are shown in the Section 3.5.2.

2.3.9.5 ComSpec – canInvalidate (Send)

The SenderComSpec defines communication attributes for a sender port (P-Port and sender-receiver interface). The canInvalidate flag indicates whether the component can actively invalidate data. The component could set this flag perhaps because the data is out of date or other application specific reasons. The canInvalidate Flag is assigned to individual DataElementPrototypes. The RTE provides an Invalidate API for any DataSendPoint that references a provided DataElementPrototype that is marked as invalidatable [2].

2.3.9.5.1 Code Generation

Concept	***RTE API***
invalidate a data element for explicit data transmission	`rte_status = Rte_Invalidate_<p>_<o>( self );`

2.3.9.5.2 Simulation

Simulation aspects are shown in the Section 3.5.2.1.

2.3.10 DataElement Filter (Receive)

Data-Filtering is handled by the RTE respectively COM. Data Filters are assigned to individual DataElementPrototypes [4].
By means of the `FILTER` attribute an additional filter layer can be added on the receiver side [2]. Value-based filters can be defined, i.e. only signal values fulfilling certain conditions are made available for the receiving component. The possible conditions are the same as listed in OSEK COM version 3.0.2. While receiving messages, only the message values allowed by the filter algorithms pass to the application [14]. If a value has been filtered out the last message value that passed through the filter is provided.

2.3.10.1 Code generation

Not needed: filtering is provided by COM layer.

2.3.10.2 Simulation

Simulation aspects are shown in the Section 3.5.3.

2.4 Compositions

The purpose of an AUTOSAR composition is to allow encapsulation of functionality by aggregating existing software components. Since a composition is also a kind of component, it again may be aggregated in even further compositions.
It is important to note that while compositions allow for system abstraction, they are solely an architectural element supporting model scalability and compositions have no effect on how components interact with the Virtual Functional Bus.
As noted in the Software Component Template [5] it has to be understood that by defining a composition a new component type is defined, which is not instantiated by itself. However, the components that are part of a composition are assigned roles with the component's context.

2.4.1 Canonical Pattern

The canonical pattern for the top level view of a composition is shown in Figure 18.

- The function calls for all Runnables are created within a subsystem "Function Call Creation".
  - In this central place it is possible to control the execution order of runnables. E.g. In order to execute simulation experiments the execution order of the runnables can be changed in this central place. (Main reason to have a central place, where all function calls are generated).
- The subsystem "Function Call Creation" belongs to the "Software Component Environment" model, as defined in the deliverable "Interaction with Behavioral Models" [3].The inside behavior of this subsystem is not specified within this document, it is up to the user to implement this in an appropriate way.
-

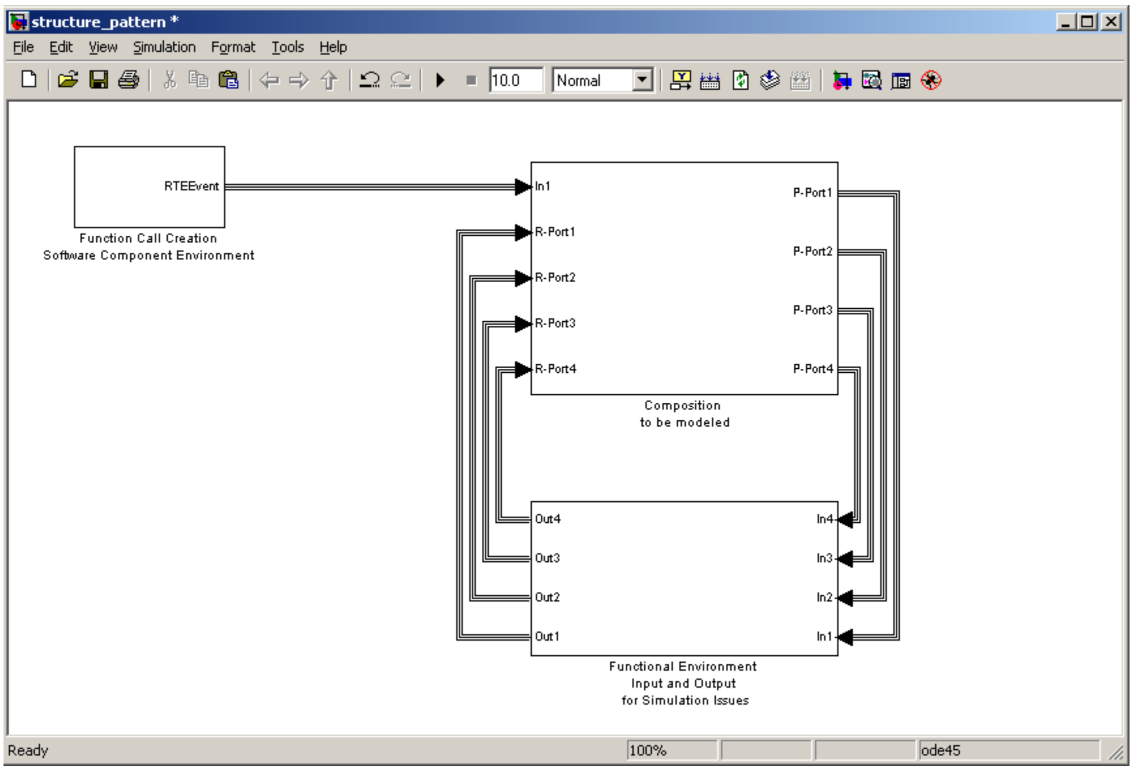


Figure 18: Canonical Pattern for the Top Level Composition View

2.5 The Use of Types and Prototypes in AUTOSAR

When examining components and compositions in AUTOSAR, one is quickly aware of the need to distinguish between meta-classes that are types and meta-classes that are prototypes. `ComponentType`, `AtomicSoftwareComponentType`, `ComponentPrototype` and `CompositionType` are all relevant.
The meta-class `ComponentType` is an abstract class and as such is not instantiable. Conversely the meta-class `ComponentPrototype` assigns a certain role to a component type (e.g. a windshield wiper component is assigned the left wiper role and another one the right wiper role). The only meta-class to support an implementation is the `AtomicSoftwareComponentType` and the need to aggregate compositions is addressed by `CompositionType`.
Examining the meta-model one sees that a `ComponentPrototype` has a <<isOfType>> reference to the `ComponentType`, which is the parent abstract class for both `AtomicSoftwareComponentType` and `CompositionType`.
A natural mechanism to model this <<isOfType>> reference relationship in Simulink is by the use of model reference or library links.
Simulink allows you to include models in other models as blocks, a feature called model referencing and allows you to include subsystems that have been defined in a library in other models as blocks, a feature called library linking. Table 3 summarizes the mapping of AUTOSAR Component Types to Simulink concepts. It is recommended that the user tags the subsystem with the AUTOSAR meta-class name to ensure that these blocks could be easily searched for in a Simulink model.

AUTOSAR Concept	***Simulink Concept***
ComponentType	Abstract class: no Simulink representation needed.
Atomic Software Component	A Simulink model or virtual/non-virtual subsystem stored in a library representing a SWC, this model must not contain any further ComponentPrototypes (Model or library reference blocks expressing ComponentPrototypes).
ComponentPrototype	A model or library reference block which references a AtomicSoftwareComponentType or a CompositionType contained in a CompositionType.
CompositionType	A Simulink model with ComponentPrototypes (blocks which reference a AtomicSoftwareComponentType or a CompositionType)

Table 3: Simulink Representation of AUTOSAR Component Types

2.5.1 Modeling one Software component

- In case of modeling only one software component, the `AtomicSoftwareComponentType` can be modelled in a normal Simulink model file.

2.5.2 Modeling several Software components

- In case of modeling more than one software component, the `AtomicSoftware-ComponentType` should be stored in a Simulink library (or in separate model files in case of model reference concept).
- In the actual Simulink model only `ComponentPrototypes` are used, which link to the Library (or reference to another model).

2.5.2.1 Simplification

- If an `AtomicSoftwareComponentType` is only used once inside a model (only one prototype of a specific type), the model can directly contain the software component type. No library link or model reference is needed. (See Atomic Software Component Type1 and Atomic Software Component Type2 in Figure 19.)
- If the same type is referenced by more than one prototype, then the type is defined within a library or referenced model (grayed subsystems in Figure 19).

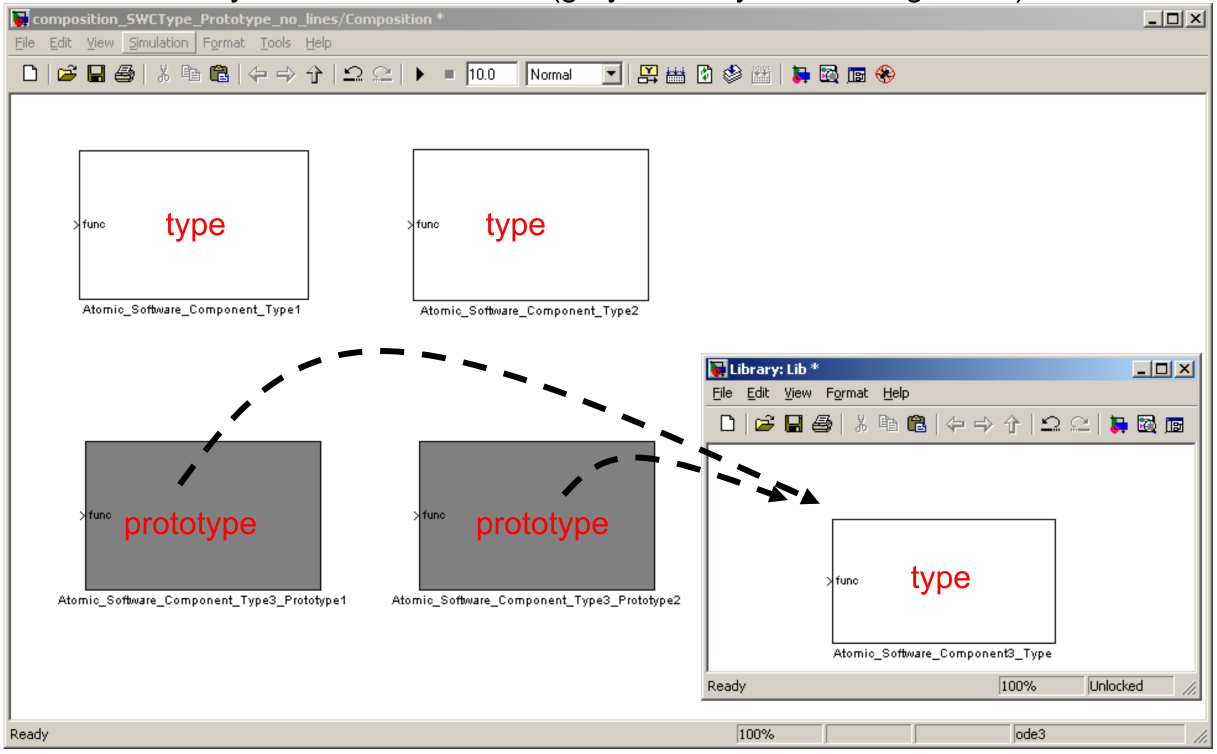

composition_SWCType_Prototype_no_lines/Composition *
File Edit View Simulation Format Tools Help
10.0
Normal
func
type
Atomic_Software_Component_Type1
func
type
Atomic_Software_Component_Type2
func
prototype
Atomic_Software_Component_Type3_Prototype1
func
prototype
Atomic_Software_Component_Type3_Prototype2
Library: Lib *
File Edit View Format Help
func
type
Atomic_Software_Component3_Type
Ready
100%
Unlocked
Ready
100%
ode3

Figure 19: Mapping of AUTOSAR Type Prototype concept to Simulink Libraries (alternatively Model Reference)

2.6 Communication between Atomic Software-Components

Figure 20 shows the communication between Atomic Software-Components within Compositions in case of sender receiver communication.
The communication is modeled without modeling of communication delays between software component. For communication delay modeling see chapter 3.

- Simulink lines represent the sender receiver communication:
  - Bus Objects for the definition of Interfaces are taken
  - Signal lines between software components represent the "assembly connectors".
  - Signal lines to the In- and Out-Ports to the upper subsystem represent the "delegation connectors".

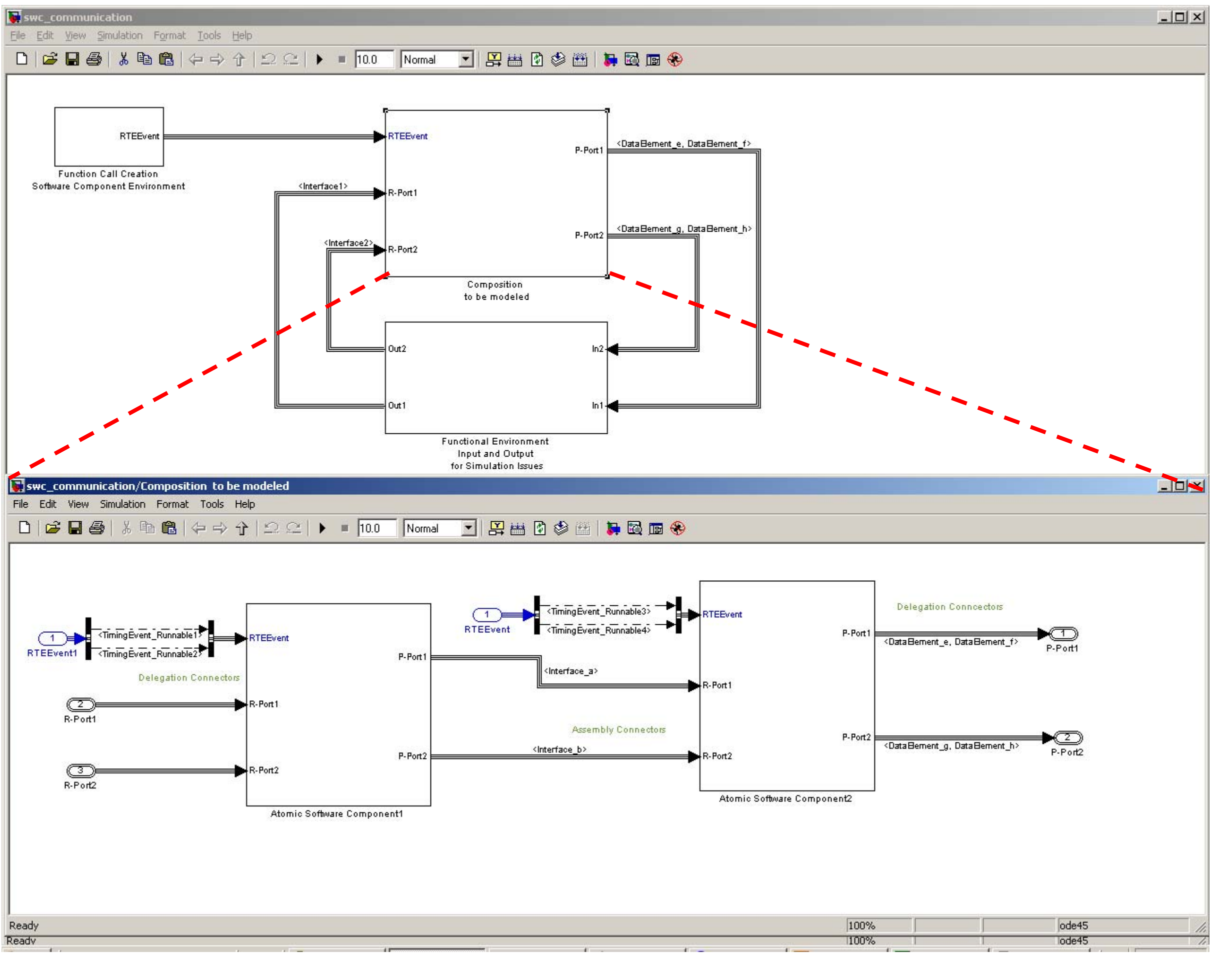


Figure 20: Communication between Atomic Software-Components

2.7 Datatypes, Constants and Parameters

2.7.1 Datatypes

The software component template defines the meta model classes for defining integers, floats as well as “complex” data types such as records. These datatypes are used to type:

- the data-elements inside a sender-receiver interface
- the arguments of the operations in a client-server interface
- constants
- characteristics
- InterRunnableVariables

2.7.1.1 Primitive Datatypes in AUTOSAR

All primitive datatypes in AUTOSAR allow an efficient mapping to programming languages like C. The user-defined datatypes, are always mapped to a set of base types. The mapping of Simulink built-ins to standard AUTOSAR primitive types is shown in Table 4.

AUTOSAR Type	***BSW Type***	***Simulink Datatype***
UInt4	uint8	uint8
SInt4	sint8	int8
UInt8	uint8	uint8
SInt8	sint8	int8
UInt16	uint16	uint16
SInt16	sint16	int16
UInt32	uint32	uint32
SInt32	sint32	int32
Float_with_NaN	float32	float
Float	float32	float
Double_with_NaN	float64	double
Double	float64	double
Boolean	boolean	boolean
Char8	uint8	uint8
Char16	uint16	uint16

Table 4: AUTOSAR and Simulink’s built-in datatypes.

2.7.1.2 Composite Datatypes in AUTOSAR

AUTOSAR provides the composite datatypes array and record. An array consists of numberOfElements elements that each have the same type, arrays have zero based

indexing. A record describes a nonempty set of objects, each of which has a unique identifier with respect to the record-type and a datatype.
Both arrays and records can consist of objects that are types by any AUTOSAR data type including arrays and records.
In Simulink it is common for a wide signal to represent an array, with individual elements being concatenated together using mux blocks and separated using demux or the selector block, records could either be represented by bus objects or alternately as a Simulink.Structtype object.
One thing to note is that it is currently not possible to have a wide signal of bus objects. The workaround would be to have a wide signal of Simulink.Struct objects, however the support for Simulink.Struct objects is not as extensive as that of bus objects.

2.7.2 Characteristics

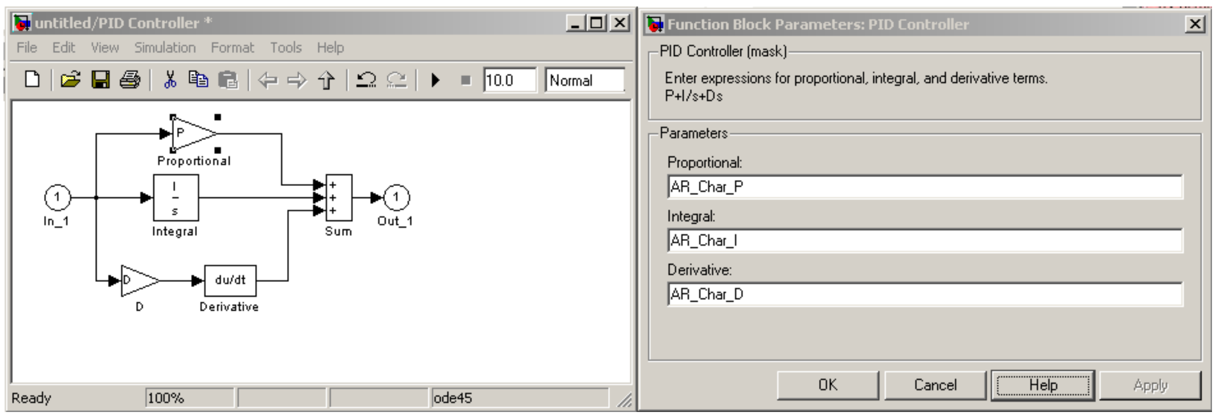
In AUTOSAR, a Characteristic defines a value that can be modified on an ECU via a calibration data management tool or an offline calibration tool. In Simulink this concept maps to a parameter. For code generation the respective RTE-API call needs to generated:

```
<return>
Rte_CData_<name>(IN Rte_Instance <instance>)
```

Where `<name>` is the configuration data name.

One way to achieve this functionality is to extend the Simulink.Parameter class to a new class, AUTOSAR.Characteristic. Using this class the user may then attach AUTOSAR.Characteristic objects in place of parameters (Figure 21).

Concept	***RTE API***
provide access to characteristics	`rtn = Rte_CData_<name>( self );`



Function Block Parameters: PID Controller

PID Controller (mask)

Enter expressions for proportional, integral, and derivative terms.
P+I/s+Ds

Parameters

Proportional: AR_Char_P

Integral: AR_Char_I

Derivative: AR_Char_D

Figure 21: Using AUTOSAR Characteristics. AR_Char_P, AR_Char_I, AR_CHAR_D are AUTOSAR.Characteristic objects.

3 RTE Simulation

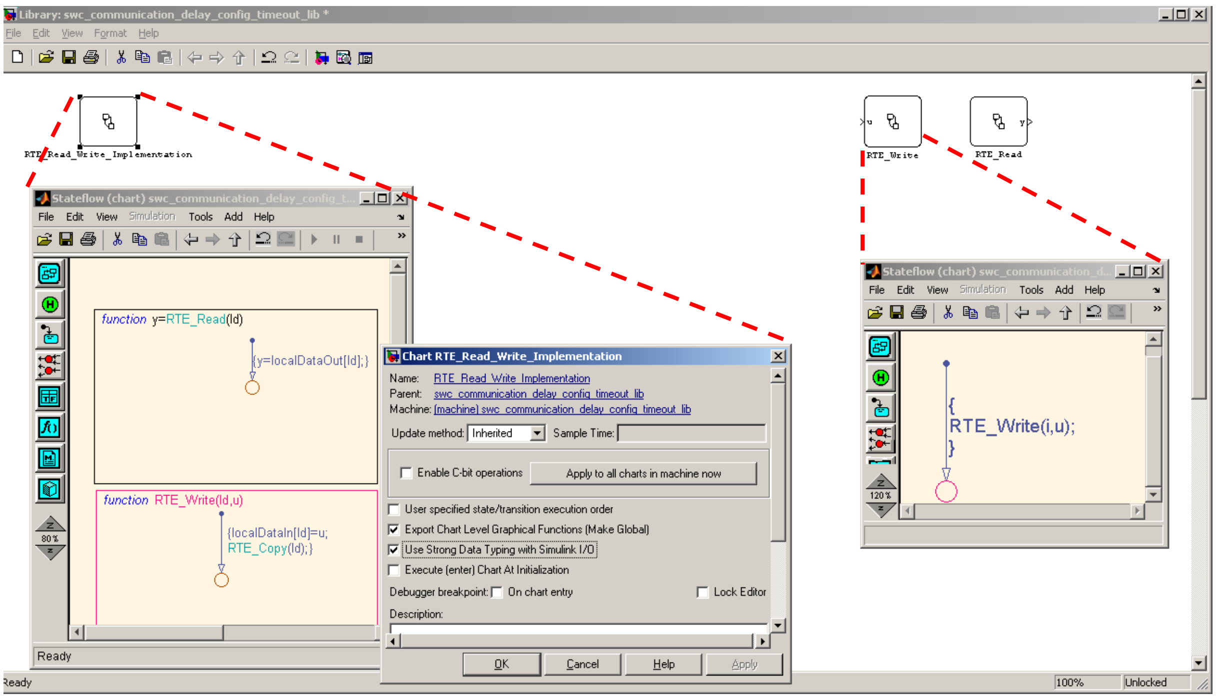
The following section shows how one could model the RTE environment in Simulink/Stateflow.

3.1 Basic Mechanism

The basic idea is to use a central Stateflow chart, which implements simple RTE functionality for simulation issues. This chart is located at a central place of the environmental model. Various simple Stateflow charts in the software component model are able to call the exported chart level graphical functions of the central chart.
By use of state charts the communication of DataElements between software components can be realized. Additional internal buffers can be added, which allow to store DataElement values etc. Concepts like communication delay modeling can be easily added in a seamless way, as well as service implementation e.g. of the NVRAM Manager.
Please note: The Stateflow charts are only used for simulation aspects, for code generation these blocks have to be ignored. For code generation custom storage classes can be used.
Please note: This Styleguide will not describe the implementation of the RTE simulation within Stateflow, it only outlines the basic mechanism.

An example of this basic mechanism is shown in Figure 22:

- As an example the two graphical functions *RTE_Read* and *RTE_Write* are provided as exported chart level graphical functions. These graphical functions can be access by other state charts in the model.
- All state charts are provided by a model library (the central chart, and the individual charts for accessing the graphical functions).


Library: swc_communication_delay_config_timeout_lib *
File Edit View Format Help
RTE_Read_Write_Implementation
RTE_Write
RTE_Read
Stateflow (chart) swc_communication_delay_config_t...
Stateflow (chart) swc_communication_d...
File Edit View Simulation Tools Add Help
Ready
100%
Unlocked
OK Cancel Help Apply

```
function y=RTE_Read(Id)
{y=localDataOut[Id];}

function RTE_Write(Id,u)
{localDataIn[Id]=u;
RTE_Copy(Id);}

{
RTE_Write(i,u);
}
```

Chart RTE_Read_Write_Implementation
Name: RTE_Read_Write_Implementation
Parent: swc_communication_delay_config_timeout_lib
Machine: (machine) swc_communication_delay_config_timeout_lib
Update method: Inherited Sample Time:
☐ Enable C-bit operations Apply to all charts in machine now
☐ User specified state/transition execution order
☑ Export Chart Level Graphical Functions (Make Global)
☑ Use Strong Data Typing with Simulink I/O
☐ Execute (enter) Chart At Initialization
Debugger breakpoint: ☐ On chart entry ☐ Lock Editor
Description:

Figure 22: Exported chart Level Graphical Functions and chart library

3.2 Modeling Runnables for Sender Receiver Communication

In the following it is outlined how the Stateflow concept for RTE simulation can be applied to communication mechanisms of the Runnables. The following basic rules have been followed:

- same model for simulation and code generation
- try to do it without simulation-specific blocks within runnable
  - for some advanced features it might still be needed
- self contained subsystem representing the runnable: code generation only of the runnable subsystem
- additional subsystems provide stubs to the simulated RTE
- signals connecting software components shall have valid values, e.g. when taking a scope for visualization.

3.2.1 Sender Receiver Communication

Figure 23 and Figure 24 illustrate the Stateflow concept for RTE simulation in the case of Sender Receiver Communication:

- In the center the runnable is located, which is target for code generation.
- Additionally the subsystems are executed before and after the runnable for simulation issues, controlled by the function call sequencer.
- The left subsystem provides the Runnable with the current DataElement values from the simulated RTE (see Figure 23). Thus the actual value of the data element is read from the simulated RTE via the RTE_Read Stateflow chart. The value present on the signal line is ignored. Thus the signal line is mainly used for illustrating the connections between software components.
- The right subsystem writes the Data Elements to the Simulated RTE buffers, similar to the left subsystem (see Figure 24). Additionally the DataElement is provided on the bus connecting the Software Component ports. Thus the signals connecting software components have valid values, e.g. when taking a scope for visualization.

Please Note: For code generation the left and right subsystems do not have any effect, since code is only generated from the Runnable subsystem. The access to the data elements is done in the same way as described in section 2.3.

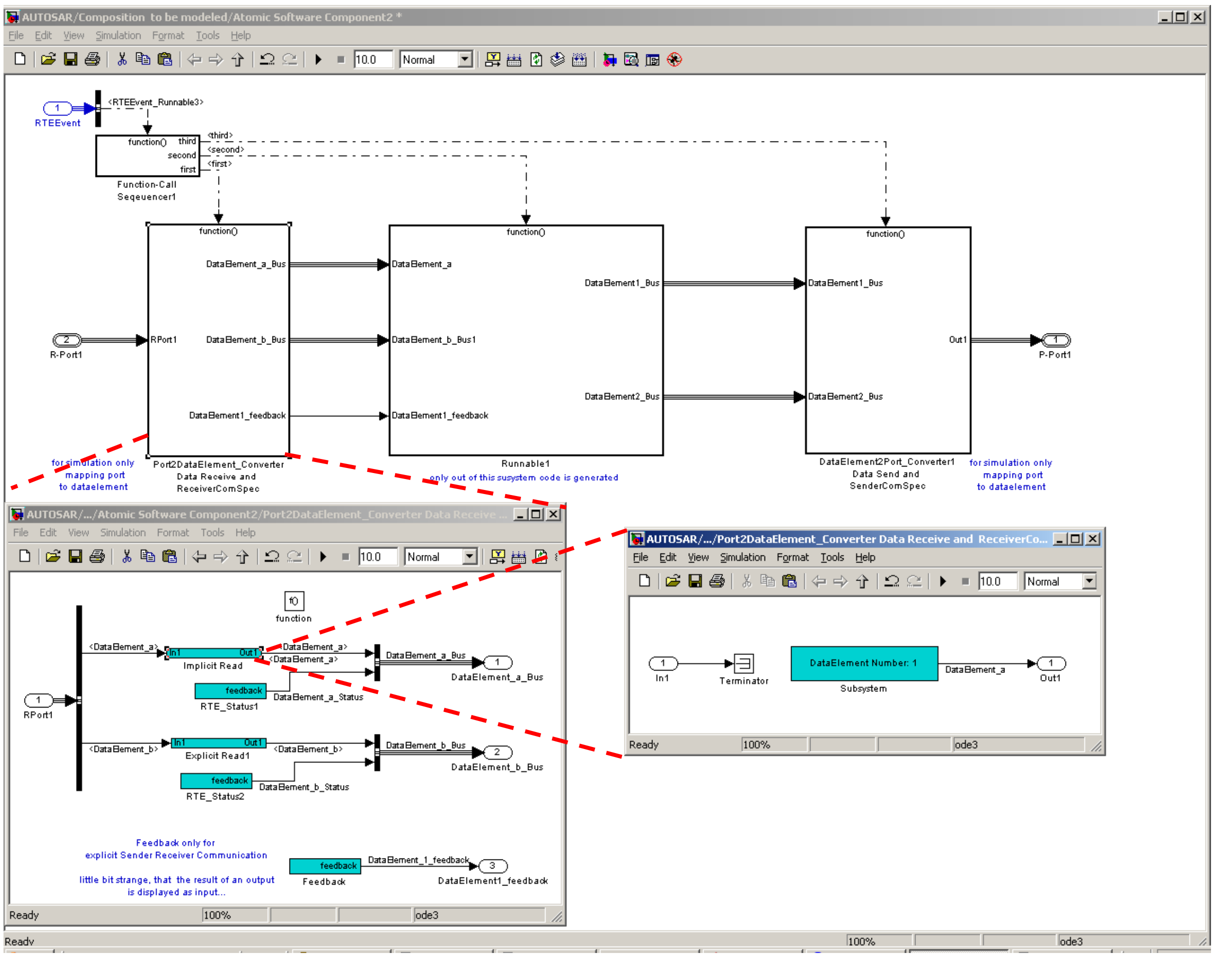

AUTOSAR/Composition to be modeled/Atomic Software Component2 *
RTEEvent
<RTEEvent_Runnable3>
Function-Call Seqeuencer1
Port2DataElement_Converter Data Receive and ReceiverComSpec
Runnable1
DataElement2Port_Converter1 Data Send and SenderComSpec
R-Port1
P-Port1
for simulation only mapping port to dataelement
only out of this susystem code is generated
AUTOSAR/.../Atomic Software Component2/Port2DataElement_Converter Data Receive ...
Implicit Read
Explicit Read1
RTE_Status1
RTE_Status2
Feedback
Feedback only for explicit Sender Receiver Communication
little bit strange, that the result of an output is displayed as input...
AUTOSAR/.../Port2DataElement_Converter Data Receive and ReceiverCo...
Terminator
DataElement Number: 1
Subsystem
Ready
100%
ode3

Figure 23: Left Subsystem

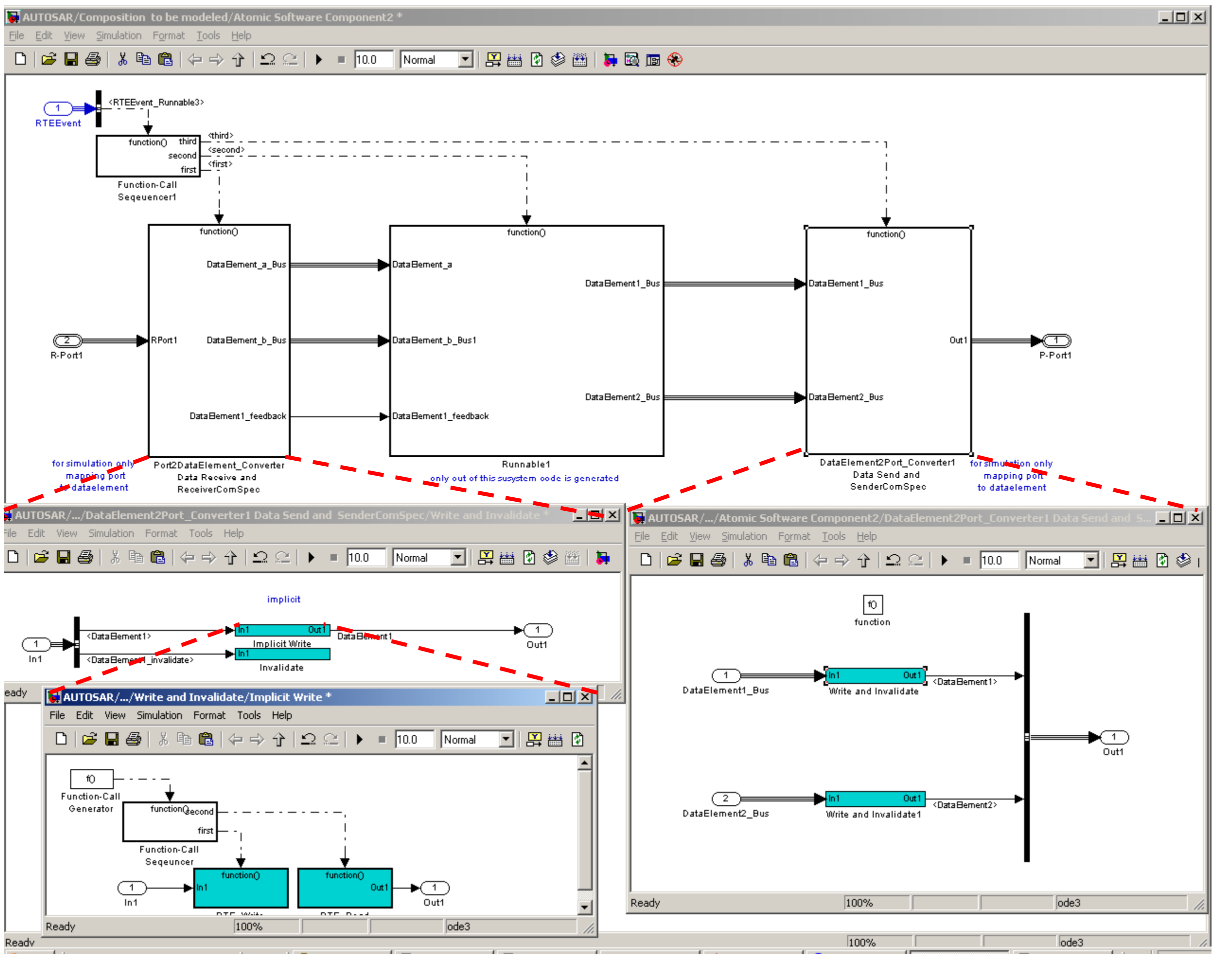

AUTOSAR/Composition to be modeled/Atomic Software Component2 *
RTEEvent
<RTEEvent_Runnable3>
Function-Call Seqeuencer1
Port2DataElement_Converter Data Receive and ReceiverComSpec
Runnable1
DataElement2Port_Converter1 Data Send and SenderComSpec
R-Port1
P-Port1
for simulation only mapping port to dataelement
only out of this susystem code is generated
for simulation only mapping port to dataelement
AUTOSAR/.../DataElement2Port_Converter1 Data Send and SenderComSpec/Write and Invalidate *
implicit
Implicit Write
Invalidate
AUTOSAR/.../Write and Invalidate/Implicit Write *
Function-Call Generator
Function-Call Sequencer
AUTOSAR/.../Atomic Software Component2/DataElement2Port_Converter1 Data Send and S...
DataElement1_Bus
DataElement2_Bus
Write and Invalidate
Write and Invalidate1
Out1
Ready
100%
ode3

Figure 24: Right Subsystem

3.3 Modeling Runnables for Client Server Communication

Simulink in general does not support a client server communication paradigm. In the following it is tried to give an outline, how client server communication could be realized. It is not claimed to be complete. In future specific block sets might be available, which support client sever communication.
Client server communication can be synchronous and asynchronous. For invoking a server operation the RTE provides the RTE_Call API. The RTE_Result API is used by a client to collect the result of an asynchronous client-server communication [2]:

```
Rte_StatusType
Rte_Call_<p>_<o>(IN Rte_Instance <instance>,
[IN|IN/OUT|OUT] <data_1>...
[IN|IN/OUT|OUT] <data_n>)
```

Where `<p>` is the port name and `<o>` the operation within the client-server interface categorizing the port.

```
Rte_StatusType
Rte_Result_<p>_<o>(IN Rte_Instance <instance>,
[OUT <param 1>]...
[OUT <param n>])
```

Where `<p>` is the port name and `<o>` the operation within the client-server interface categorizing the port.

3.3.1 Client Server Communication – Client Model

Figure 25 shows the client model. Inside the client runnable, a server operating is called. This is realized by a specific block, which realizes the RTE_Call API.

3.3.1.1 Synchronous Client Server Communication

3.3.1.1.1 Simulation

The Stateflow approach outlines how client server communication can be realized within Simulink:

- A Stateflow chart models the RTE_Call API for simulation issues. The chart is provided with a number of input data.
- The RTE_Call-chart calls a central state chart (via virtual exported chart level graphical function), which simulates the RTE behavior. This central state chart immediately triggers the server runnable via function call via event broadcast. The server runnable is triggered via a function call, which realizes the OperationInvokedEvent. By the event broadcast construct the client runnable interrupted, until the server has completed (synchronous) (see Figure 27).
- The RTE_Call-chart provides the result of the server call, via the “out-data” signals (could also be a signal bus).
- The R-Port is realized by a Simulink outport. It does not have any functionality, it is only used in order to display the port on software component level. Therefore the outport is connected with a ground block.

3.3.1.1.2 Code generation

In case of Code generation instead of the state flow chart, a block has to be provided, which implements the RTE_Call API according to the RTE specification.

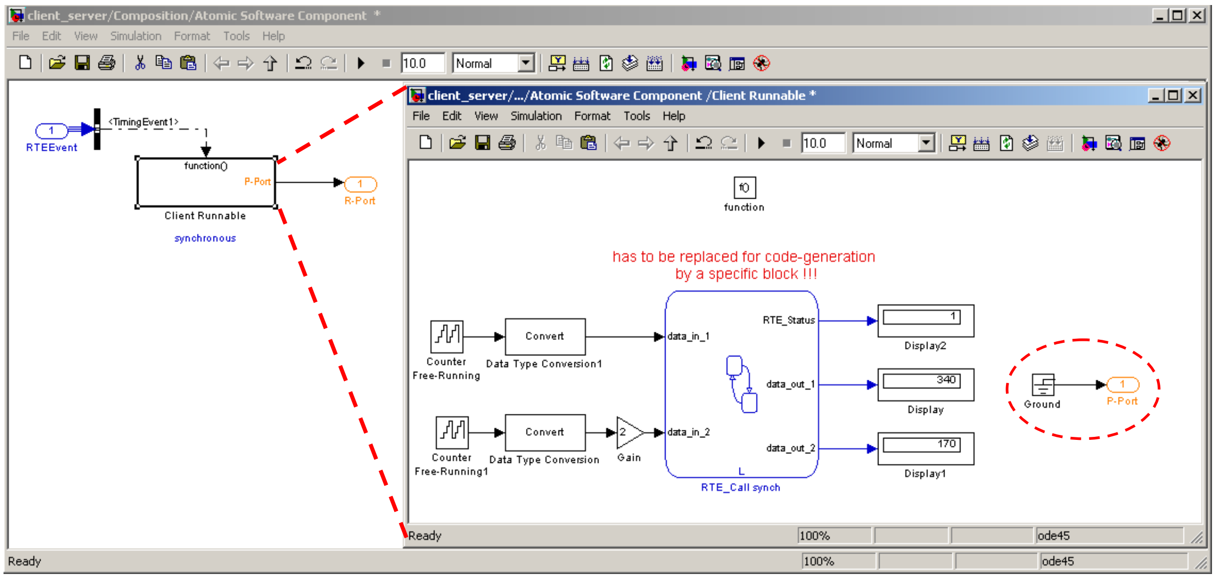


Figure 25: Model of Client Runnable (Synchronous)

3.3.1.2 Asynchronous Client Server Communication

3.3.1.2.1 Simulation

Figure 26 shows the Simulink model for asynchronous client server communication:

- Similar to the synchronous call, input data can be provided.
- The RTE_Call-chart also calls a central state chart, which simulates the RTE behavior. This central state chart returns without providing output data. Somewhere later in time the simulated RTE triggers the asynchronous server call returns runnable.
- Inside the "*Asynchronous ServerCallReturns runnable*", the result of the server call can be accessed via an additional chart.

3.3.1.2.2 Code generation

In case of Code generation instead of the Stateflow chart, blocks have to be provided, which implement the RTE_Call and RTE_Result API according to the RTE specification.

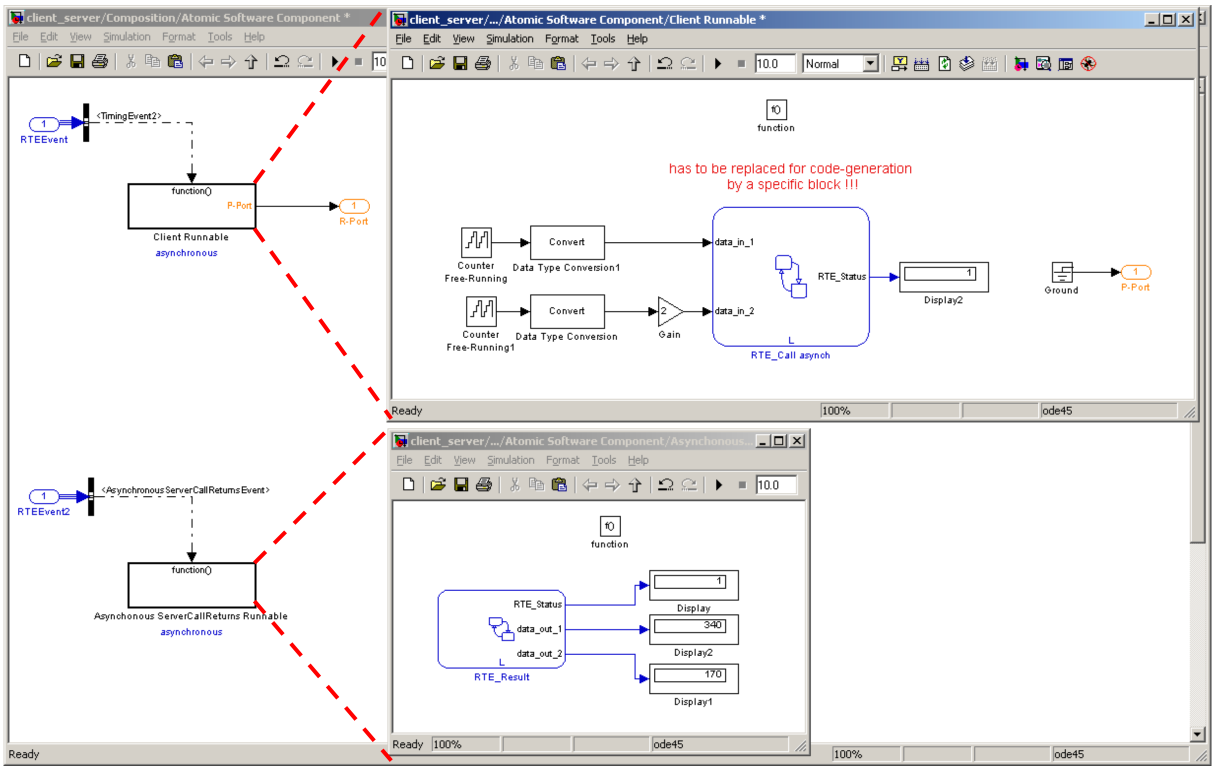


Figure 26: Model of Client Runnable (Asynchronous)

3.3.2 Client Server Communication – Server Model

3.3.2.1 Simulation

Figure 27 shows the server model for client server communication.

- The left and the right subsystems are only for simulation issues. They provide the server runnable with input data from the simulated RTE and provide with RTE the resulting output data of the server call for the RTE.
- Inside the of subsystem in the middle, the server functionality is modelled.
- The P-Port is realized by a Simulink inport. It does not have any functionality, it is only used in order to display the port on software component level. Therefore the inport is connected with a ground block.

3.3.2.2 Code generation

In Figure 27 a similar approach to section 3.2 is used. Only the middle subsystem representing the server runnable is target of code generation. Out of this subsystem the code-generator has to provide a RTE conformant server implementation.

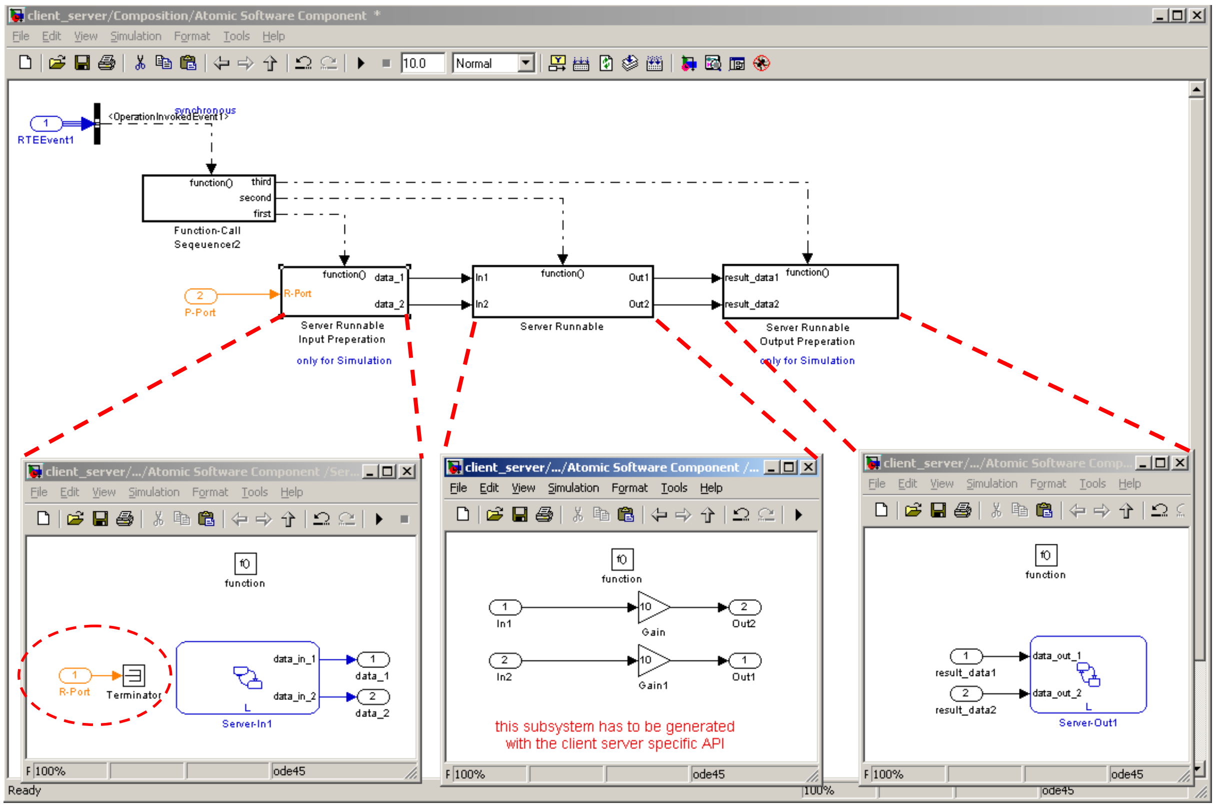

client_server/Composition/Atomic Software Component *
RTEEvent1
<OperationInvokedEvent1>
synchronous
function() third second first
Function-Call Seqeuencer2
R-Port
P-Port
Server Runnable Input Preparation
only for Simulation
Server Runnable
Server Runnable Output Preperation
only for Simulation
function
Terminator
Server-In1
data_1
data_2
In1
In2
Gain
Gain1
Out1
Out2
this subsystem has to be generated with the client server specific API
result_data1
result_data2
Server-Out1
Ready
ode45

Figure 27: Model of Server Runnable

3.3.3 Client Server Communication – Software Component View

Figure 28 shows the software component view of client server communication. The R-Ports and P-Ports can be connected, in order to display the relationship between clients and servers. However this connection does not have any functionality.

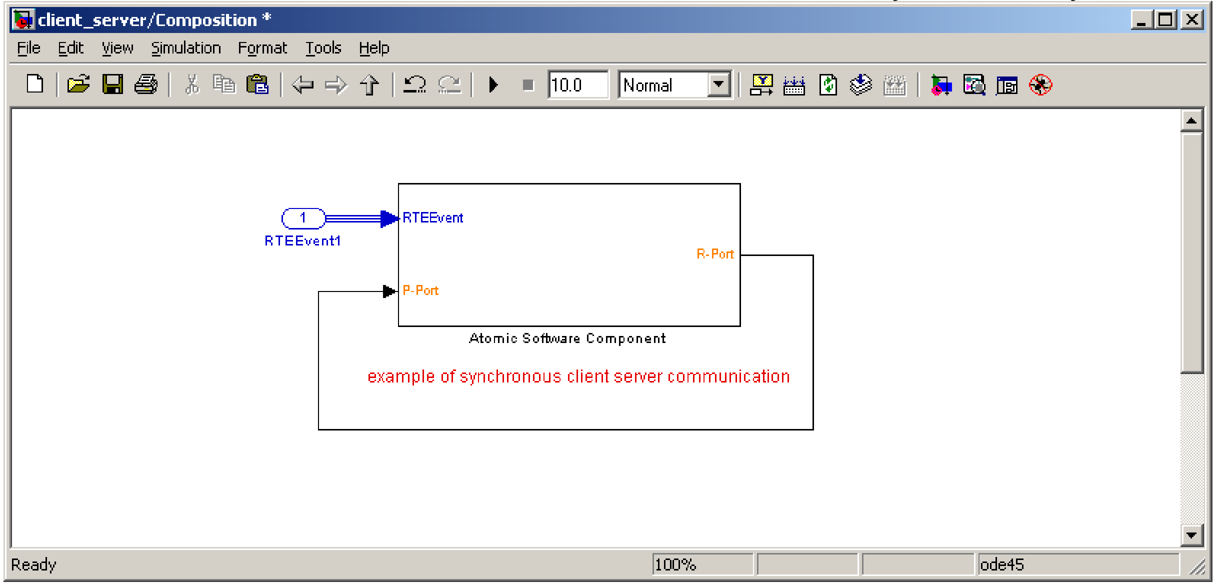

client_server/Composition *
File Edit View Simulation Format Tools Help
10.0
Normal
RTEEvent1
RTEEvent
R-Port
P-Port
Atomic Software Component
example of synchronous client server communication
Ready
100%
ode45

Figure 28: Representation of R-Ports and P-Ports on Software Component Level

3.4 Modeling distributed functional networks – Simulation of Communication Delays

For modeling distributed functional networks within Simulink, the modeling of communication delays is interesting, in order to evaluate functional behavior of software components with respect to communication delays, mapping aspects etc.
Figure 29 shows the top level view for the modeling of communication delays:

- Additionally to the subsystems representing atomic software components communication delay subsystems are added, which allow to simulate the communication delays. The communication delay blocks are assigned to the R-Port side of software components. Thus in the example of Figure 29 the communicated DataElements of Port1 can have different communication delays for the communication to *Atomic Software Component 2* and *Atomic Software Component 3*.
- These communication delay subsystems are triggered via an additional function call bus. The generation of these function calls are handled by a central state chart in the outside simulation environment.
- Inside the communication delay subsystems the communication delay is assigned to individual data elements, since data elements can follow independent communication paths, or are located in different bus messages.

Please note: These communication delays do not have effect to the code generation, since code is only generated for individual Runnables of the atomic software components.

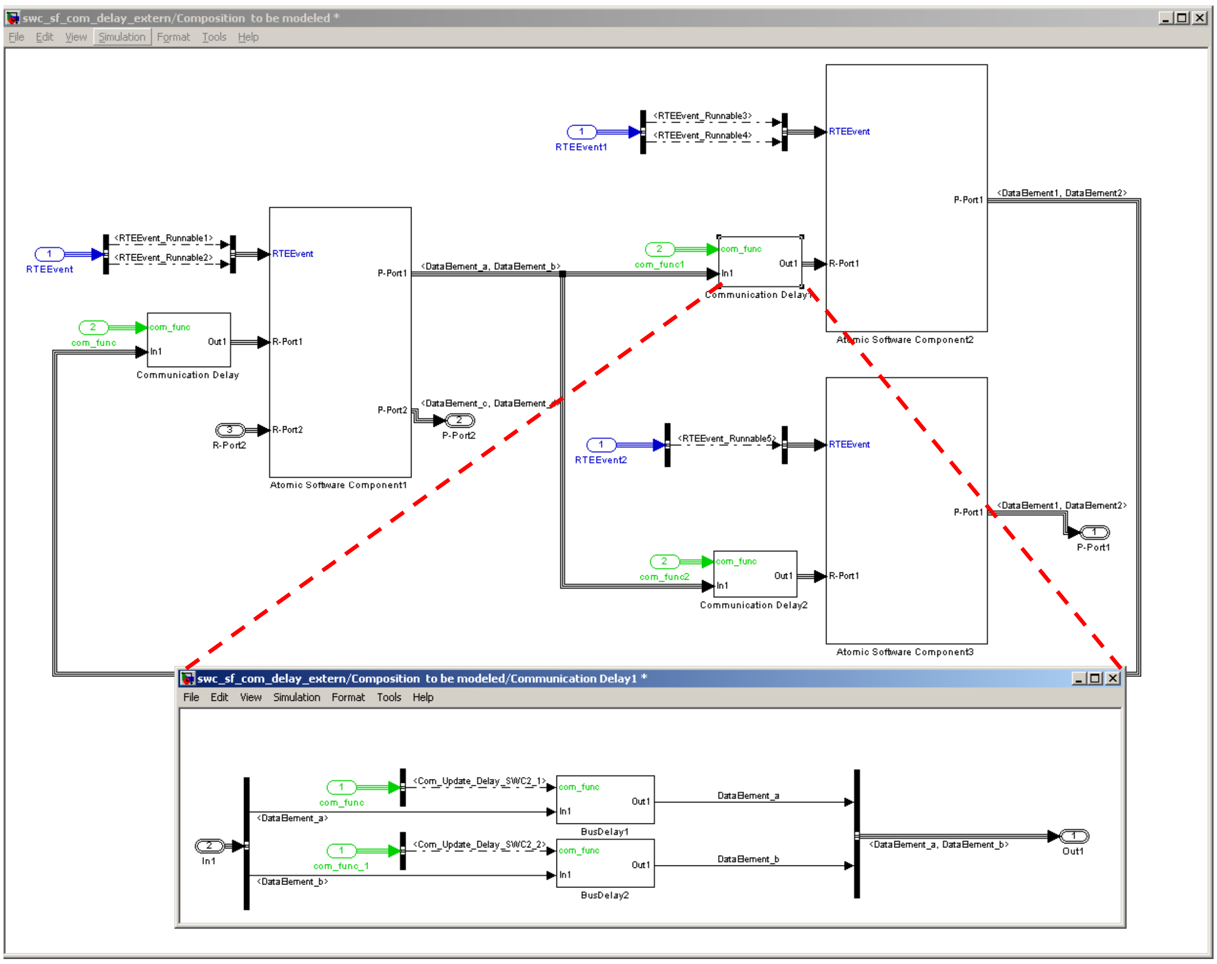

swc_sf_com_delay_extern/Composition to be modeled *
File Edit View Simulation Format Tools Help
RTEEvent
RTEEvent1
RTEEvent2
<RTEEvent_Runnable1>
<RTEEvent_Runnable2>
<RTEEvent_Runnable3>
<RTEEvent_Runnable4>
<RTEEvent_Runnable5>
com_func
com_func1
com_func2
In1
Out1
Communication Delay
Communication Delay1
Communication Delay2
P-Port1
P-Port2
R-Port1
R-Port2
<DataElement_a, DataElement_b>
<DataElement_c, DataElement_d>
<DataElement1, DataElement2>
Atomic Software Component1
Atomic Software Component2
Atomic Software Component3
swc_sf_com_delay_extern/Composition to be modeled/Communication Delay1 *
File Edit View Simulation Format Tools Help
<Com_Update_Delay_SWC2_1>
<Com_Update_Delay_SWC2_2>
com_func_1
<DataElement_a>
<DataElement_b>
BusDelay1
BusDelay2
DataElement_a
DataElement_b
<DataElement_a, DataElement_b>

Figure 29: Modeling communication delays of distributed functional networks

The communication delay block is realized by using the Stateflow approach (see Figure 30): The upper function call subsystem is called by the function call Creation block of the software component environment, at a defined point in time, which the reads the current DataElement value from the simulated RTE.
Note the connector ComSpec has a maxTransferTime and maxJitter attribute which could be used for simulation purposes and should be parameterized in the bus delay blocks.

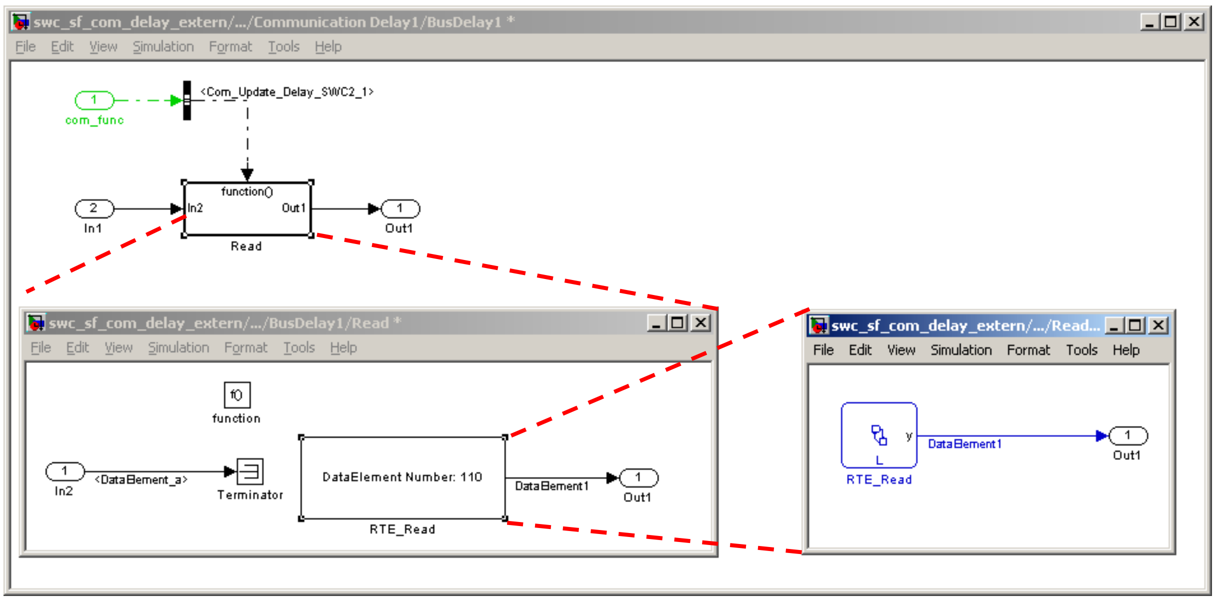

swc_sf_com_delay_extern/.../Communication Delay1/BusDelay1 *
File Edit View Simulation Format Tools Help
1
com_func
<Com_Update_Delay_SWC2_1>
2
In1
function()
In2
Out1
Read
1
Out1
swc_sf_com_delay_extern/.../BusDelay1/Read *
File Edit View Simulation Format Tools Help
f()
function
1
In2
<DataElement_a>
Terminator
DataElement Number: 110
RTE_Read
DataElement1
1
Out1
swc_sf_com_delay_extern/.../Read...
File Edit View Simulation Format Tools Help
y
L
RTE_Read
DataElement1
1
Out1

Figure 30: Realization of the communication delay block by using the Stateflow approach

3.5 Simulation of Communication Specification – ComSpec

To provide the status signals during simulation the DataReceiverComSpec and DataSenderComSpec blocks of Figure 14 may internally contain a Stateflow Chart block, which provides the connection to the simulated RTE. That is to say, the internal RTE status logic is simulated with a global state machine in the environment model, which has chart blocks as "satellites" in every DataReceiverComSpec and DataSenderComSpec block.

3.5.1 ComSpec – initValue

The initValue is handled within the RTE simulation layer.

3.5.2 ComSpec – Acknowledgement Request

The AcknowledgementRequest requests acknowledgements that data has been sent successfully. Success or failure is reported via a SendPoint of a Runnable. The AcknowledgementRequest is assigned to individual DataElementPrototypes. The attribute timeout specifies the number of seconds before an error is reported.

The attribute type specifies the part of communication the acknowledgement is requested for [4]:

- "transmission" refers reaching the receiving port,

where "reception" refers to the value being actually passed to the receiving component code.
When `transmission_ack` is specified, the RTE will inform the sending component that the signal has been sent correctly [2].

3.5.2.1 ComSpec – canInvalidate

The SenderComSpec defines communication attributes for a sender port (P-Port and sender-receiver interface). The canInvalidate flag indicates whether the component can actively invalidate data. The component could set this flag perhaps because the data is out of date or other application specific reasons. The canInvalidate Flag is assigned to individual DataElementPrototypes. The RTE provides an Invalidate API for any DataSendPoint that references a provided DataElementPrototype that is marked as invalidatable [2].
For simulation issues the invalidation of Data Elements has to be transferred from the Runnable to the simulation environment. Figure 16 shows the usage of 2 Invalidate Signals *DataElement_1_Invalidate* and *DataElement_2_Invalidate* which are passed from the *Runnable3* to the subsystem *Data Send and SenderComSpec*. Inside the subsystem *Data Send and SenderComSpec* the simulation environment is updated.

3.5.3 DataElement Filter

Data-Filtering is handled by the RTE respectively COM. Data Filters are assigned to individual DataElementPrototypes [4].
By means of the `FILTER` attribute an additional filter layer can be added on the receiver side [2]. Value-based filters can be defined, i.e. only signal values fulfilling certain conditions are made available for the receiving component. The possible conditions are the same as listed in OSEK COM version 3.0.2. While receiving messages, only the message values allowed by the filter algorithms pass to the application [14]. If a value has been filtered out the last message value that passed through the filter is provided.
For simulation purposes the filter mechanism should be provided within Simulink. Therefore subsystems providing the DataElement values from the simulated RTE can be extended with additional filter functionality.

4 References

4.1 Normative References to AUTOSAR documents

[1] Specification of the Virtual Functional Bus
AUTOSAR_VirtualFunctionBus.pdf

[2] Specification of RTE Software
AUTOSAR_SWS_RTE.pdf

[3] Specification of Interaction with Behavioral Models
AUTOSAR_InteractionBehavioralModels.pdf

[4] Metamodel
AUTOSAR_Metamodel.eap

[5] Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

[6] Glossary
AUTOSAR_Glossary.pdf

[7] AUTOSAR Services
AUTOSAR_Services.pdf

[8] Requirements on Interoperability of Authoring Tools
AUTOSAR_RS_InteroperabilityAuthoringTools.pdf

[9] Methodology
AUTOSAR_Methodology.pdf

[10] Specification of System Template
AUTOSAR_SystemTemplate.pdf

[11] Specification of ECU Resource Template
AUTOSAR_ECUResourceTemplate.pdf

[12] Specification of Feature Definition of Authoring Tools
AUTOSAR_FeatureDefinition.pdf

[13] Specification of Graphical Notation
AUTOSAR_GraphicalNotation.pdf

4.2 Normative References to Non-AUTOSAR documents

[14] OSEK COM Spec 3.0.3
http://www.osek-vdx.org/mirror/
OSEKCOM303.pdf

[15] Simulink
http://www.mathworks.co.uk/products/simulink/