ASD:Suite User Manual

ASD:Suite Release 4 v9.2.7

TABLE OF CONTENTS

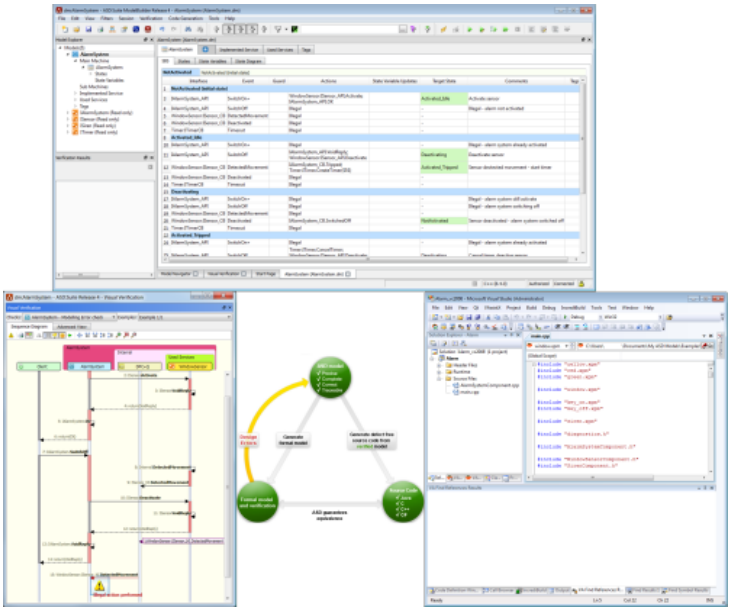
The ASD:Suite software design platform

User Manual

ASD:Suite Release 4 v9.2.7

The ASD:Suite is a software design (CAD) platform based upon Verum's patented Analytical Software Design (ASD) technology. ASD makes it possible to create systems from mathematically verified components.

The ASD:Suite is used to define and (automatically) verify models, and to (automatically) generate fully executable source code from these models. The models specify both structure and behaviour of services, and of components that implement and use these services. See the "ASD Concepts" section for details.



See "Installation and Start-up" for guidelines about installing and setting up the ASD:Suite.

Note: Starting with the ASD:Suite Release 3 v7.2.0 you have the possibility to install the ASD:Suite ModelCompare, a feature that allows you to find and eliminate differences between two versions of an ASD model or between two related or unrelated ASD models.

The set of User Manuals for the ASD:Suite consists of:

- The ASD:Suite Release Notes (see archive for latest and older versions).
- The ASD:Suite User Manual (this document; see archive for older versions).
- The ASD:Suite ModelCompare User Guide (see archive for latest and older versions).
- The ASD:Suite Visual Verification Guide (see archive for latest and older versions).
- The ASD Runtime Guide (see archive for latest and older versions).

A simple Alarm system example, consisting of a set of interface models and design models together with the related source code can be downloaded from here. This is a fully executable system that can be built using Visual Studio (for C++ and C#) and Eclipse (for Java).

To uninstall the ASD:Suite Release 4 v9.2.7 use the "Start -> All Programs -> ASD Suite Release 4 V9.2.7 -> Uninstall" item.

ASD:Suite Installation Overview

Supported platforms

The ASD:Suite client-side software is available for the Microsoft Windows platform and various Linux platforms. It is tested with the following versions (including correction packages):

Note: this section is about the platforms on which the ASD:Suite modelling tools run, not the platforms that you can run the generated code on.

- Windows (see Installing on Windows):
  - Microsoft Windows XP, Service Pack 3
  - Microsoft Windows 7, Service Pack 1
- Linux (see Installing on Linux):
  - Xubuntu 13.04 (32 bit, i686)
  - ubuntu 12.04 LTS (32 and 64 bits)
- Solaris (see Installing on Solaris):
  - Sun Solaris 2.10 (32 bit sparc) (command-line tools only)

Supported Programming Languages and Compilers

The supported compilers for the various programming languages are specified below:

- For **C++** see "Supported compilers and boost versions for C++"
- For **C#** see "Supported compilers and execution platforms for C#"
- For **C** see "Supported compilers and execution platforms for C"
- For **Java** see "Supported compilers and execution platforms for Java"

3rd Party Dependencies

The ASD:Suite includes software developed by the following parties:

- **Boost.org** : "Boost Software License"
- **The OpenSSL Project** & **Cryptsoft** : "OpenSSL License and Original SSLeay License"
- **Arabica - XML Parser Library** : "Arabica XML and HTML Processing Toolkit License"
- **The Expat XML Parser** : "Expat License"
- **The libexecstream library** : "The libexecstream library license"
- **Apache Xerces C++ - XML Parser Library** : "Apache License, Version 2.0"
- **Graphviz - Graph Visualization Software** : "The Eclipse Public License - v 1.0"
- **XZip** : "XZip original author comments"

Installing on Windows

You can download the latest version of the ASD:Suite client tools from the ASD:Portal. Login to the ASD:Portal, select 'Downloads' in the top menu, and click on the green Download button. Extract the downloaded zip-file, start the installation from the file 'ASD Suite Release xxxxx Setup.exe', and follow the instructions. You have the choice of where the suite is installed, whether to install the Compare Tool or not, and whether to make file associations or not.

The ASD:Suite is by default installed into your program files directory. Settings are stored in your AppData\Roaming\Verum folder. Any log files end up in AppData\LocalLow\Verum.

Once installed, continue with ASD:Suite first time start-up.

Installing on Linux

There are two alternatives for installing the ASD:Suite client tools on linux: there are tarballs and Debian packages (which work for Ubuntu and Xubuntu) available. You can download the latest version of the ASD:Suite from the ASD:Portal. Login to the ASD:Portal, select 'Downloads' in the top menu. Download the desired package. Make sure you select the right package (32-bit or 64-bit) for your machine architecture.

Installing from a tarball

There are two different tarballs available: one containing all tools (asdsuite-4-x.y.z.architecture.tgz) and one containing only the command-line tools (asdsuite-cmdline-4-x.y.z-architecture.tgz). The command-line tools can be run all by themselves. The other tools (ModelBuilder, ModelViewer and Compare Tool) have some dependencies. Since you are simply unpacking a tarball, you need to resolve these dependencies yourself.

Unpack the tarball (e.g. by typing tar -xvf *filename*). It will extract the tools into a directory called asdsuite-4-x.y.z. For the ModelBuilder, ModelViewer, or Compare Tool to work, you will need to install Graphviz (at least version 2.23) from http://graphviz.org/. The other dependencies are most likely already present on your system. If not, you can use the "ldd" command on the executables to find any unmet dependencies.

Installing from a .deb package

For Ubuntu, select the .deb package. Once downloaded, you can double-click the package in the file browser to install it. Alternatively, you can use the command-line: e.g. "sudo dpkg -i *debian-package-file*" for a Debian package. Note however that the command-line versions given here do not automatically resolve dependencies.

The following files will be installed. Note that also symlinks will be created that allow you to type abbreviated names for the tools on the command-line, e.g. "asdmb" for "asdmodelbuilder".

- Binary files in /usr/lib/asdsuite
- Symlinks to the tools in /usr/bin (see the full list using "ls -l /usr/bin/asd*")
- Documentation in /usr/share/doc/asdsuite
- Settings files in $HOME/.verum

Note: it is not possible to install multiple versions of the ASD:Suite using the packages.

Once installed, continue with ASD:Suite first time start-up.

Installing on Solaris

For Solaris, only the command-line tools are available. They are statically linked against the libraries they need, so unpacking them is the only thing that is necessary. You can download the latest version of the ASD:Suite from the ASD:Portal. Login to the ASD:Portal, select 'Downloads' in the top menu. Download asdsuite-cmdline-4-x.y.z-sparc-sun-solaris2.10.tgz. Unpack the tarball (tar -xvf *filename*).

The command-line tools are extracted into a directory called asdsuite-4-x.y.z. See Command-line Tools for how to use them.

Once installed, continue with ASD:Suite first time start-up. Note that since no graphical clients are available on Solaris, you will use the command-line asdauthorize tool to setup the server communication.

ASD:Suite first time start-up

Prerequisites

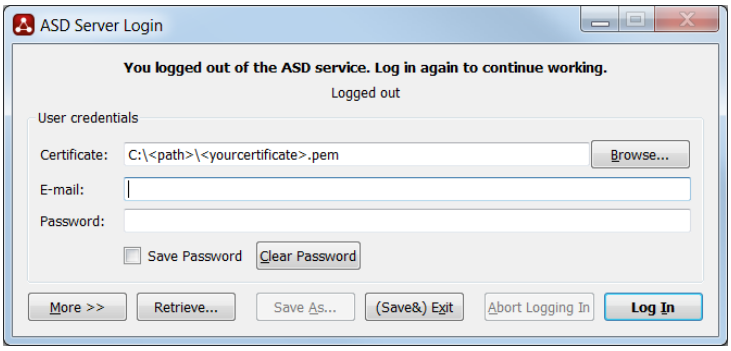
To get on your way with ASD:Suite, you'll need four things:

- An **Internet connection**.
  Note: In case you connect to the Internet through a proxy server, you will also need the proxy server settings. Typically you can get this information from your IT department.
- Your **username** and **password**. Once your ASD:Suite user account has been created by your ASD Administrator, you will receive an automated email with a temporary password. Go to the ASD:Portal (portal.verum.com), login using your email address and this temporary password, and set your personal password.
- The **ASD:Suite installation**. See Installing on Windows, Linux or Solaris for instructions.
- A **license certificate file**. You will receive this license certificate file from your ASD Administrator. The filename should end with a .pem extension.

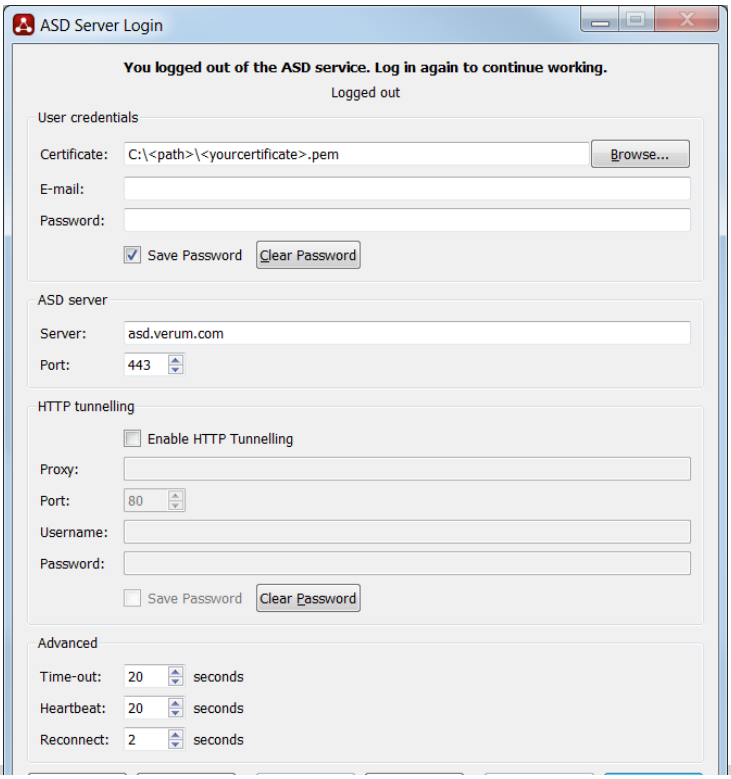
Setting up the ASD:Suite using the graphical client tools

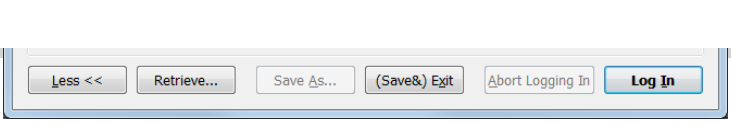
If you chose to install the ModelBuilder, ModelViewer and/or Compare Tool, then you can use these to setup the connection to the ASD Server. If you only installed the command-line tools (impossible on Windows, but can be done with a tarball installation on Linux and Solaris), see the next chapter below.

1. Copy the license certificate file to a local folder of your choice on your PC.
2. Start the ASD:Suite ModelBuilder, you'll see the ASD Server Login dialog:



3. Click on the 'Browse...' button, and select the license certificate file you copied in step 1.
4. Enter your email address and your personal ASD:Suite password. In case you want to save your password so that you won't need to fill in this dialog in the future, put a tick in the 'Save Password' checkbox.
5. In case your connection to the Internet goes via a proxy server, continue with step 6, otherwise click 'Log in'. A connection to the ASD Server will be established and the ASD:Suite ModelBuilder is ready for use.
6. Click on the 'More >>' button, this expands the dialog to show the Server Settings:





7. In case your connection to the Internet goes via a proxy server, then tick the checkbox 'Enable HTTP Tunnelling' and fill in the proxy server settings under 'HTTP Tunnelling'. In general you can obtain these settings from your IT department. The username and password are the username and password that you have to use to access the proxy server, not your ASD:Suite username and password.

8. Click 'Log In'. A connection the the ASD Server will be established and the ASD:Suite ModelBuilder is ready for use.

Setting up the ASD:Suite using the command-line client tools

The tool "asdauthorize" lets you setup the server connection from the command-line.

1. Copy the license certificate file to a local folder of your choice on your PC.

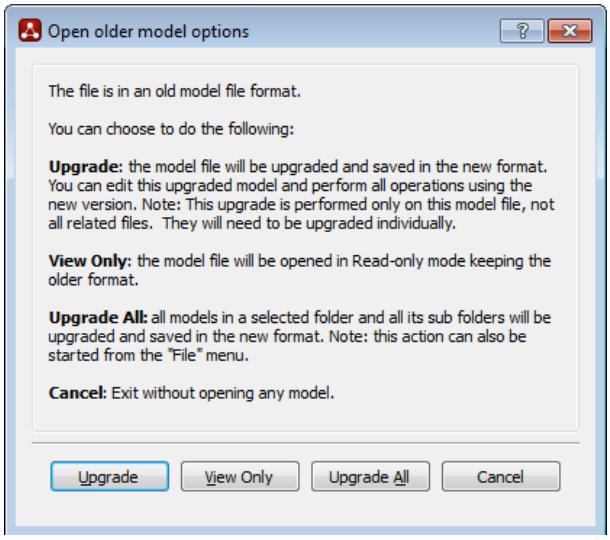
2. Use the "--register" command of the asdauthorize tool to save server connection settings. See asdauthorize for details.

Upgrading ASD models

The ASD:Suite provides the possibility to automatically upgrade ASD models which were built using a previous major release of the ASD:Suite.

> **Note:** The major release is indicated as the first number out of the three indicating the version of the released ASD:Suite.
>
> **Note:** on Linux and Solaris, upgrading models is only supported from models made with ASD:Suite version 9.0.0 or higher. On Windows, all models can be upgraded to the current version.
>
> **Tip:** Use the "File->Upgrade Models..." menu item if you want to upgrade all the models located in a folder and its sub-folders. Alternatively, use the command-line ASD:Convert tool.
>
> **Tip:** If you need to set a new code generator version for multiple models, you can use the AsdEdit command-line tool.

The following figure shows the dialog you see when you open a model of an older major version:



Upgrade dialog

The following table shows the effect of choosing one out of the four options presented above in case of an interface model:

Operation	Model upgraded	Model saved
Upgrade	Yes	Yes
View Only	Yes	No
Upgrade All	Yes	Yes
Cancel	No	No

The following table shows the effect of choosing one out of the three options presented above in case of a design model:

Operation	Design Model		Related Interface Models	
	upgraded	saved	upgraded	saved
Upgrade	Yes	Yes	Yes	No
View Only	Yes	No	Yes	No
Upgrade All	Yes	Yes	Yes	Yes
Cancel	No	No	No	No

ASD concepts

Components

ASD is a component-based technology in which systems are composed of a mixture of *ASD components* and *Foreign components*. Within ASD, a component is a common unit of architectural decomposition, specification, design, mathematical verification, code generation and runtime execution.

ASD components

ASD components are software components that are specified and designed using ASD. An *interface model* specifies the externally visible behaviour of a component. A *design model* specifies its inner working and how it interacts with other components. All ASD components must have both an interface model and a design model.

ASD components are mathematically verified. In the ASD:Suite this is done using a Software as a Service (SaaS) application. The necessary mathematical models are generated automatically from both design and interface models. The source code to implement an ASD component is generated automatically from its design model.

Foreign components

Foreign components are hardware or software components of a system which are not developed using ASD. As they have to be used by ASD components, they must correctly interface and interact with them. They may be third party components, legacy code or handwritten components representing those parts of a system that cannot be generated from ASD designs. All used foreign components must have an interface model which specifies the externally visible behaviour of the component. Foreign components do not have a corresponding design model.

The interface model of foreign components is used for three purposes:

1. For verifying ASD components that use these foreign components: formal models are generated automatically from the interface models. They are used to verify that an ASD component interacts correctly at runtime with the corresponding foreign component.
2. For code generation: to generate the correct interface header files.
3. For stub-generation: to generate stub-code that can easily be integrated with ASD component, and that can be edited and extended manually.

Note: The handwritten implementation provided for the foreign component must correctly implement all methods declared in the generated interface header files. This includes ASD specific methods like GetInstance, ReleaseInstance, GetAPI, and RegisterCB.

Models

ASD supports two types of models:

- Interface Model
- Design Model

Interface Model

An interface model is a model of the externally visible behaviour of an ASD component or Foreign component, i.e. the service that the component implements.

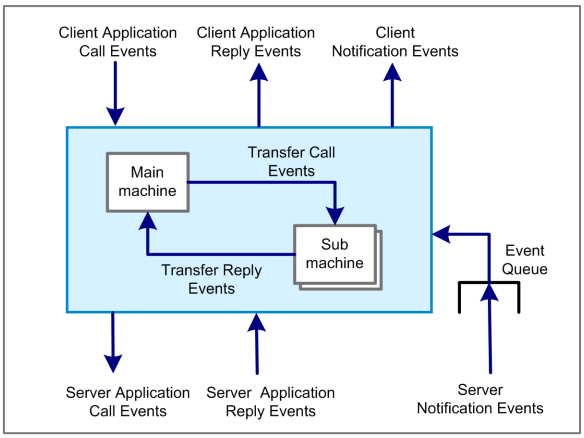
- It identifies the component's application interfaces and notification interfaces and specifies their associated events.
- It specifies the externally visible behavioural semantics of the component in the form of one Sequence-Based Specification.
- It may also specify modelling interfaces with associated events that are hidden from the client and are used to represent hidden internal behaviour of the implementation.
- The triggers in an interface model are occurrences of:
  - Call events on application interfaces;
  - Notification events on modelling interfaces.
- The actions in an interface model are occurrences of:
  - Reply events on application interfaces;
  - Notification events on notification interfaces.

Design Model

A design model is a model of the internal behaviour of an ASD component.

- Its implemented service and used services are specified by interface models;
- It fully and deterministically specifies the internal logic of the component as one or more Sequence-Based Specifications;
  - A simple design is represented by a single Sequence-Based Specification;
  - A complex design is partitioned hierarchically into a main machine and one or more sub machines. Each of these is specified by a Sequence-Based Specification.
- If the design is partitioned:
  - There is one *transfer interface* defined for each sub machine through which the main machine and sub machine communicate.
  - Transfer interfaces are not visible to clients or servers.
- The triggers in a design model are occurrences of:
  - Call events on application interfaces of the implemented service;
  - Reply events on the application interfaces of the used services;
  - Notification events on the notification interfaces of the used services;
  - For a main machine, reply events on transfer interfaces;
  - For a sub machine, call events on its transfer interface.
- The actions in a design model are occurrences of:
  - Call events on application interfaces of the used services;
  - Reply events on application interfaces of the implemented service;
  - Notification events on notification interfaces of the implemented service;
  - For a main machine, call events on transfer interfaces;
  - For a sub machine, reply events on its transfer interface.

The following figure shows the various types of events in a design model:



The various types of events in a design model

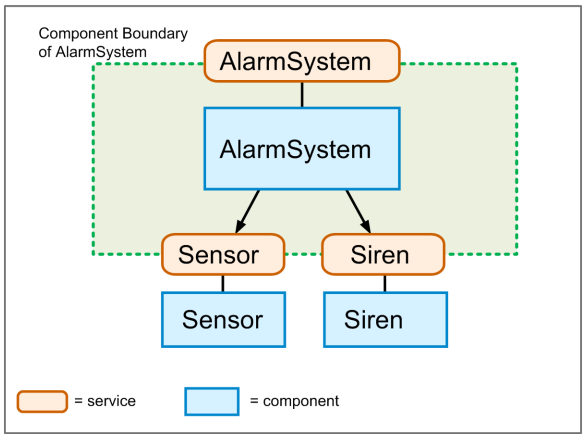
Sequence Based Specifications

According to the IEEE Standard Glossary of Software Engineering Terminology, a *specification* is a complete, precise and verifiable description of the characteristics of a system or a component. Within ASD the distinction between an *interface specification* (interface model) and a *design specification* (design model) is fundamental.

The interface model describes the externally visible behaviour of a component and is as implementation-free as possible. This means that the model defines *what* the component does under every circumstance but not *how* the component will do it. The external behaviour is specified independently of any specific implementation. It is an abstraction of the component or system implementation that every compliant design is required to implement.

The design model describes the internal behaviour of a component. It rigorously and completely defines one of the many possible implementations that faithfully comply with the interface model.

An ASD component implements a *service* (specified by the interface model) that is used by its *clients*. This *implemented service* is exposed by means of *application interfaces* through which clients can send *call events*. The ASD component can respond to a call event on an application interface by means of a *reply event* on the same application interface and notification events on *notification interfaces*. In this process, an ASD component can also invoke *used services* that are implemented by other components: the *servers* of the ASD component. Collectively, the services between a component and its clients and servers form an imaginary border, called the *component boundary*. Information crosses the component boundary in the form of events.



Component boundary

A component "knows" only information passed *into* it across the component boundary in the form of the *triggers* it receives. A trigger can be:

- A call event from a client through an application interface;
- A reply event from a server through an application interface;
- A notification event from a server through a notification interface.

Similarly, a component exposes information to its clients and servers across the component boundary in the form of the *actions* it sends. An action can be:

- A call event to a server through an application interface;
- A reply event to a client through an application interface;
- A notification event to a client through a notification interface.

An interface model is defined in terms of only those events that pass between a component and its Clients. A design model is defined in terms of events that pass between the component, its Clients and its Servers.

Within ASD, both interface models and design models are defined in the form of *Sequence-Based Specifications* (SBS). Behaviour is specified in a tabular form as a total Black Box function, by mapping all possible sequences of triggers to the corresponding actions.

The following figure shows part of an SBS specified in the ASD:Suite:

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	**NotActivated (initial state)**							
4	IAlarmSystem	SwitchOn+		IWindowSensor::ISensor.Activate; IAlarmSystem::Ok		Activated_Idle	*Activate sensor*	
5	IAlarmSystem	SwitchOff		Illegal		-	*Illegal - alarm not activated*	
6	IWindowSensor::ISensor_NI	DetectedMovement		Illegal		-		
7	IWindowSensor::ISensor_NI	Deactivated		Illegal		-		
8	Timer::ITimerCB	Timeout		Illegal		-		
9	**Activated_Idle**							
12	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm system already activated*	
13	IAlarmSystem	SwitchOff		IWindowSensor::ISensor.Deactivate; IAlarmSystem::VoidReply		Deactivating	*Deactivate sensor*	
14	IWindowSensor::ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Timer::ITimer.CreateTimer(5)		Activated_Tripped	*Sensor dectected movement - start timer*	
15	IWindowSensor::ISensor_NI	Deactivated		Illegal		-		
16	Timer::ITimerCB	Timeout		Illegal		-		

An SBS in the ASD:Suite

The method used to create these specifications, is called *Sequence Enumeration*. This requires the systematic enumeration of all possible input sequences of triggers, ordered by length, starting with the empty sequence. Triggers can be repeated within a sequence and since sequence length is not restricted, the set of all possible sequences is infinite. In practice, systems do not display an infinite set of unique, non-repeating behaviours. They cycle through a finite set of states and repeat a finite set of behaviours. Thus the infinite set of input sequences of triggers can be reduced to a finite set of equivalence classes.

Each class is identified by a minimal length sequence, called *Canonical Sequence*. All sequences in a given equivalence class have the same *future* system behaviour. They are said to be *Mealy Equivalent*. The equivalence classes form the set of states in a *Mealy Machine*.

The theory underlying this approach tells us that by reasoning about the behaviour of the finite set of Canonical Sequences, we can

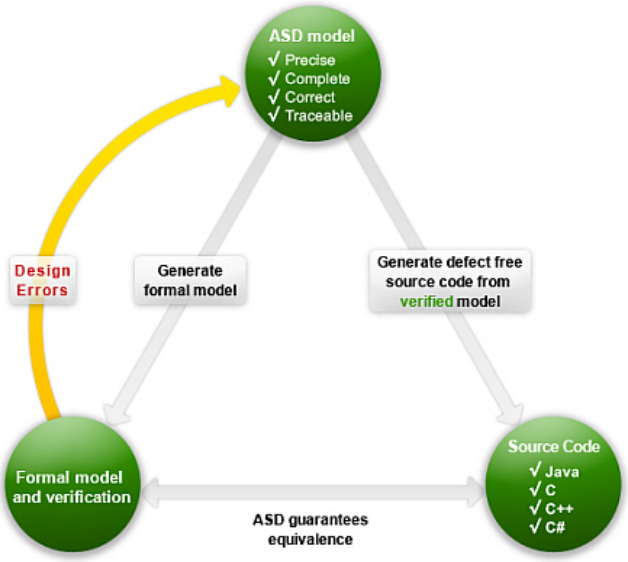
reason about the behaviour of every possible input sequence. The Sequence Enumeration method used in ASD thus defines the Black Box function as a total function between the finite set of Canonical Sequences and the corresponding actions.

The ASD Triangle and Correctness

Architects and designers can use the ASD:Suite to create models of software components, verify their completeness and (behavioural) correctness, and generate source code from verified design models. At its core, ASD guarantees the (mathematical) equivalence of:

1. An ASD model;
2. A formal representation of that model;
3. The source code generated from the ASD model.

This equivalence is called the *ASD Triangle*:



The ASD Triangle

Operational semantics of rule cases

At runtime, a single detail row (a rule case) is interpreted as follows:

- When the trigger occurs and the guard evaluates to "true" (or is omitted), then, as a single atomic action, all of the following occur:
  - The actions are executed in the order in which they are specified in the "Actions" column
  - The state variable updates are performed (using simultaneous assignment semantics)
  - The state transition takes place.
- If an action is defined as "valued", i.e. one that gives back a synchronous reply, it must be the last action in the action list of this rule case. The reply is processed as a trigger that occurs after the state transition has taken place.

The following figure shows a set of rule cases specified in an SBS:

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	**NotActivated (initial state)**							
4	IAlarmSystem	SwitchOn+		WindowSensor:ISensor.Activate; IAlarmSystem.Ok		Activated_Idle	*Activate sensor*	
5	IAlarmSystem	SwitchOff		Illegal		-	*Illegal - alarm not activated*	
6	WindowSensor:ISensor_NI	DetectedMovement		Illegal		-		
7	WindowSensor:ISensor_NI	Deactivated		Illegal		-		
8	Timer:ITimerCB	Timeout		Illegal		-		
9	**Activated_Idle**							
12	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm system already activated*	
13	IAlarmSystem	SwitchOff		WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating	*Deactivate sensor*	
14	WindowSensor:ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Timer:ITimer.CreateTimer(5)		Activated_Tripped	*Sensor dectected movement - start timer*	
15	WindowSensor:ISensor_NI	Deactivated		Illegal		-		
16	Timer:ITimerCB	Timeout		Illegal		-		

There are four "abstract" actions that have special meaning in a rule case:

- Illegal
  When the action in a rule case is *Illegal*, this means that the corresponding trigger event is *not allowed to happen* in that state. This is a specification or design choice. Note that in this case, the environment can generate such a trigger event. The *Illegal* action is the ASD equivalent of a programmer writing "assert(false)". For example, when a component offers an initialisation method on its interface, it is required that initialisation is only to be performed once. In such case, the initialisation method may be *Illegal* after initialisation has been performed.

- Blocked
  This only applies to a design model.
  When the action in a rule case is *Blocked*, this means that the corresponding trigger event *will not and cannot occur* due to ASD semantics; it is physically not possible. For example, when a component A has sent a call event to some other component B, it can only "see" the synchronous reply event. Since component A has to preserve run-to-completion semantics, it cannot handle other call events performed at its interface and neither can it process asynchronous notification events. In other words, these are then *Blocked*. There is a limited number of places where Blocked can be used in an ASD Design Model. These are slightly different for the various state types and for the Main- and Sub-state machine SBSs:

  - In a Normal state, all synchronous reply events from used components and transfer events are *Blocked.*
    As there is no call to a used component or a sub-machine active, these reply events cannot occur.

  - In a Synchronous Return state, all client call events, used notification events and transfer events are *Blocked.*
    As the component is busy with a call to a used component, only a synchronous reply event from that used component can occur.

  - In a Super-state or Initial Sub-state, all client call events, client reply events and used notification events are *Blocked.*
    As in that state the main or sub-machine is waiting for a transfer event, all other types of events cannot occur.

  The ASD:ModelBuilder will ensure correct use of the *Blocked* actions in an ASD Design Model, when it can determine the type of state. If it cannot determine the type of state (e.g. when you create a floating state, or when there are different types of transitions into the state (one with a void call and one with a valued call)), the ASD:ModelBuilder will leave the action cells empty.

- Disabled
  This only applies to an interface model.
  In an ASD Interface Model *Disabled* is only used for modelling events, and its semantics imply that the modelling event is disabled in that particular state. In other words, it is a specification or design choice that the implementing component will not perform the internal behaviour corresponding to the modelling event in that particular state. The ASD:ModelBuilder ensures that the use of the *Disabled* action is limited to modelling events only. However you as designer have to decide where the modelling events have Disabled actions and where they have other actions. Note that in an interface model, the modelling events cannot have an *Illegal* response. The ASD:ModelBuilder will ensure this is the case.

- NoOp
  This is the equivalent to "do nothing". In other words, there is no action to the corresponding trigger event.

Note that these abstract actions cannot be combined with other actions in the same rule case.

Client requests

- All triggers on the Client Application Interface are implemented as method calls.
- When an Application Interface trigger is executed, the execution takes place under the context of the Client's thread and the Client code thus can't be executed until the synchronous call returns.
- The response to the Client trigger, and thus its return to the client caller, takes place when the component issues an action on the Client Application Interface. Until this occurs, the Client remains synchronously blocked.
- A trigger implemented as a "void" method takes a "VoidReply" action as a signal to return to the Client.
- A trigger implemented as a method returning a synchronous reply value, requires the corresponding action in order for the Client to continue execution.
- While the Client is blocked, the component can continue receiving notifications but it can not receive any other trigger from ***any*** Client thread via any of its Client Application Interfaces. As seen by its Clients, an ASD component has Monitor semantics.

Notification interfaces

- Notification interfaces exist to provide notifications to Clients.
- Notification interfaces are implemented by the Client.
- An action which maps onto a notification interface is always non-blocking.
- All notifications are "void" events. They can have parameters, if required.
- Circular control dependencies that occur when independent ASD components are composed into a system may cause deadlocks. To prevent these, notifications are decoupled via a queue
- Every ASD component which uses a service with at least one notification interface will automatically include the queue anfd a decoupled calling mechanism. In the *Multi-threaded* execution model the ASD component also has it's own active DPC thread to process the notification events from the queue.

ASD Timers and the Timer Cancel Guarantee

- ASD components can make use of the ASD Timer service by instantiating as many Timers as they need. To instantiate a Timer you have to specify the ITimer model as a used service in the design model of the ASD component and you have to specify the desired instances of the respective service.

  Note:
  - The ITimer interface model which needs to be specified as a used component is built-in in the ASD:Suite. There is no separate ITimer model.
  - The "Used" checkbox of each of the application and notification interfaces of the ITimer service must be checked in the design model.
  - The ASD timer is a special used service that can not be shared among component designs, i.e. the design model must use all interfaces of the ITimer model (both the ITimer application interface and the ITimerCB notification interface).
  - As the ITimer model is built-in, it cannot not be changed.
  - The implementation of the ASD timer is completely integrated in the ASD Runtime.
- ASD Timers implement the *CreateTimer*, *CreateTimerMSec* and *CreateTimerEx* triggers to start the timer for a specified duration in seconds, milliseconds, or seconds and nanoseconds, respectively. Timer completion is signalled via the *TimeOut* notification event.
- All ASD Timers support a *CancelTimer* trigger. The ASD Runtime guarantees that once a timer has been cancelled, the *TimeOut* trigger will never occur, not even if the timer has actually expired and the *TimeOut* event is waiting in the queue to be processed by the component's own DPC-thread.

State types in a design model

Within a design model, five types of state are distinguished:

- **Initial state in the main machine**. This is the state in the main machine where the component starts after construction. The inital state is a "Normal state", and any Normal State can be made Initial State.
- **Super state**. These are states in the main machine in which a sub machine is active. In these Super-states, all triggers must have "Blocked" as associated action except for the transfer reply events that correspond to the active sub machine.
- **Initial state in a sub machine**. This is the state in which the sub machine is not active (i.e. the main machine is not in the corresponding Super state). In the Initial states of a sub machine, all triggers must have "Blocked" as associated action except for the transfer call events that correspond to this sub machine.
- **Synchronous return state**. These are states following a valued action, where the design is waiting for a valued reply from the called used service (note that these synchronous return states may exist in the main machine as well as in a sub machine). In these Synchronous return states, triggers must have "Blocked" as associated action except for the application reply events that correspond with the called used service.
- **Normal state**. These are all other states. In these states NONE of the external triggers (i.e. implemented service application call events and used service notification events) may have a "Blocked" action. If no action is allowed or possible for these triggers in a normal state, the action must be set to "Illegal". For example, when the application interface is "closed" (the Client is waiting for a reply to an application call event). On the other hand, all application reply events and transfer reply events must have a "Blocked" action in a normal state, since there is no application call event to a used service active, nor is there any sub machine active.

Action Sequence

An *action sequence* is everything that happens starting from when a thread enters the ASD component monitor until it exits the component monitor. This thread can be either the client thread or the DPC thread. Note that the client thread can still be blocked on the client barrier after the action sequence (i.e. the thread does not necessarily exit the entire component). For more details on the component monitor and the client barrier concepts, see the Monitor Semantics section in the Runtime Guide.

Action sequences can span multiple rule cases and even multiple state machines (i.e. main and sub machines).

The concept of action sequences is important because e.g. it often occurs that data variables are used to retain data during an action sequence. ASD provides a mechanism to invalidate such data variables at the end of the action sequence, so you are protected from "leaking" data into subsequent calls.

Definition:

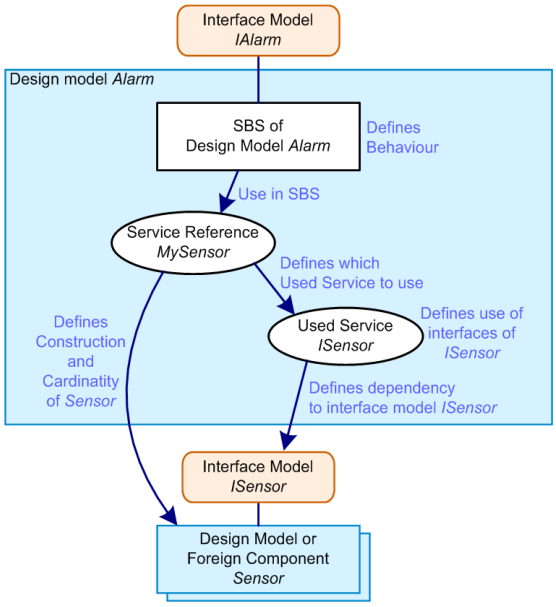
- An action sequence starts when a trigger arrives in a normal state.
- An action sequence ends after a transition is made to a normal state.

Corollaries:

- An action sequence is started either by a client call event or by a used service notification event.
- An action sequence that is started on a rule case, can end on the very same rule case if the rule case has a normal state as target state.
- Any rule case that has a valued call at the end of its Actions cell does *not* end an action sequence - the sequence continues across the following synchronous return state(s).
- Any rule case that has a transfer call at the end of its Actions cell does *not* end an action sequence - the sequence continues into the sub machine.
- An action sequence started by a call event can continue across several synchronous return states, if subsequent valued calls are made.
- An action sequence started in the main machine can continue into a sub machine and end there.
- An action sequence started in a sub machine can continue into the main machine and end there.
- An action sequence started in the main machine can continue into a sub machine, exit the sub machine, and end in the main machine.
- An action sequence does *not* necessarily end when a rule case has a reply event in its Actions.

Used Services

A design model can use instances of other services. This is done using various concepts:



How a design model uses other services

Used Service

A Used Service specifies two things:

1. The dependency to a used interface model. This comprises the relative path to the interface model, and the collection of interfaces and events, and the "Broadcast" flags.
2. The usage of that particular interface model. This comprises which interfaces are actually in use by the design model, and for notification events also the "Observed" and "Singleton" flags

Each Used Service is defined by:

1. A name, this is in most cases equal to the service name.
2. The (relative) path and filename of the interface model
3. Per interface an indication whether the interface is used in this dependency
4. In case of a used notification interface, per notification event an indication whether the event is a Singleton event or not
5. In case of a "Broadcast" used notification interface, per notification event an indication whether the event is Observed or not

This together defines how the design model makes use of a used interface model.
There can be more than one Used Service for the same used interface model, which then can differ in the usage of interfaces and/or notification events.

Service Reference

Service References are references to instances of components implementing a specific service. They are similar to programming language variables that contain references to a component.

Each service reference is defined by:

1. A reference name,
2. The cardinality, which is the number of component instances,
3. Construction information, this can either be a component name (either an ASD component or Foreign component) that specifies the service instance's internal behaviour or a *use* statement to inject a component. See "Service References" for details.
4. A Used Service, indicating the dependency to the actual service that specifies the behaviour of the service reference,

The service references determine the grouping of rule cases. Triggers coming from used service instances that are part of one service reference are all processed by the same set of rule cases.

Service references can be used to send an event to multiple service instances at once. The actions are sent to the service instances in the order that the services instances have in the service reference.

The content of a service reference cannot be changed after construction (in contrast to state variables of type Used Service reference). Service references need to be initialized with instances. There are two ways of doing this:

- Let ASD handle it. In this case, you need to provide the name of the component (which can be different than the service name = interface model name) so that ASD can call the GetInstance function of that component.
- Provide an instance in hand-written code during construction of the parent component. See "Construction Parameters" for how to do this.

Used service reference state variable

A state variable of type Used Service Reference is a service reference whose contents can change dynamically. It can be used in guards, actions and state variable updates. See "State Variables for Used Service References" for details.

Unlike service references, state variables can contain the same instance multiple times, which could be used to send the same action multiple times to the same service instance. In contrast, service references cannot (by construction) include the same service instance multiple times (under the assumption that the components are "multiples", see "Specifying component type" for details).

Purpose of windows

The main window of the ASD:Suite, also referred as the Master window, is by default filled in by the *Start Page* pane.

Next to the ***Start Page*** you can load in the Master window one or more of the following "dockable" windows:

- "**Model Explorer**", used to display the structure of the loaded ASD models.
- "**Model Editor**", used to view or edit an ASD model. There is a model editor per loaded ASD model, which you open by double-clicking the model in the Model Explorer.
  Note: the Model Editor has the modelname (and filename between brackets) in the title bar.
- "**Verification Results**", used to display the set of checks for verification. See the "ASD:Suite Visual Verification User Guide" for details.
- "**Visual Verification**", used to display the information needed to fix the failed checks. See the "ASD:Suite Visual Verification User Guide" for details.
- "**Conflicts**", used to display messages associated to specification conflicts. See "Conflicts" for details.
- "**Output Window**", used to display the progress of certain time consuming operations, like loading of an ASD model and/or generating code.
- "**Find Results**", used to display the results of a "Find All" operation.
- "**Model Navigator**", used to visualize the relationships between ASD models in a given directory. See "Model Navigator" for details.
- "**Un-saveable Data**", which shows syntactically incorrect table cells that cannot be stored into the model file. See "Un-saveable Data" for details.

Personalisation of windows

You can drag dockable windows (like the Output Window or Model Editor) out of the Master Window to create a so-called Slave Window. Multiple dockable windows can be dragged into the same Slave Window.

> **Tip**: double-click the title bar of a dockable window to automatically undock it and create a new slave window.

You can drag or load as many "dockable" windows in an already created slave window, but you can not have a ***Start Page*** pane in it.

You can change the layout of a (Master or Slave) window by dragging the "dockable" windows around by their title bar to various places in the respective window. If you drag a "dockable" window to another window, it will remember the place it last had in that window and dock itself there.

> **Note:** You can NOT drag Slave windows - you need to grab the title bar of the contained "dockable" window instead.

Empty Slave windows are closed automatically. The following list reflects alternatives to close a slave window:

- Select the Slave window and press Alt+F4
- Press the  button in the top-right corner of the window
- Drag and drop all its "dockable" windows in the Master window or in an other Slave window
- Close all windows docked in the Slave window. A dockable window can be closed by one of the following:
- Press the X button in the blue window title bar
- Press the  button if the window is tabbed in the Slave window
- Deselect the item in the View menu of the Slave window (not for Model Editor windows)

Note:

- The Master and Slave windows are normal application windows that can be a.o. minimized and maximized.
- The window layout, the location of Master and Slave window(s) on the screen, is remembered per screen setup. This means that if you switch from single to dual screen setup, you have to reorder your windows to accomodate for the dual screen setup.

Meaning of colours in the SBS tab

The various colours used in the SBS tab of the "Model Editor" each have a meaning.

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	**NotActivated (initial state)**							
4	IAlarmSystem	SwitchOn+		WindowSensor:ISensor.Activate+		Activating	*Activate sensor*	
11	**Activating (synchronous return state)**							
16	WindowSensor:ISensor	OK		IAlarmSystem.Ok		Activated_Idle		
17	WindowSensor:ISensor	Failed		IAlarmSystem.Failed		NotActivated		
21	**Activated_Idle**							
25	IAlarmSystem	SwitchOff		WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating	*Deactivate sensor*	
28	WindowSensor:ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Timer:ITimer.CreateTimer(5)		Activated_Tripped	*Sensor dectected movement - start timer*	
31	**Deactivating**							
39	WindowSensor:ISensor_NI	Deactivated		IAlarmSystem_NI.SwitchedOff		NotActivated	*Sensor deactivated - alarm system switched*	
41	**Activated_Tripped**							
45	IAlarmSystem	SwitchOff		Timer:ITimer.CancelTimer; WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating	*Cancel timer, deactive sensor*	
50	Timer:ITimerCB	Timeout		Siren:ISiren.TurnOn		Activated_Alarm...	*Timeout - turn siren on*	
51	**Activated_AlarmMode**							
55	IAlarmSystem	SwitchOff		Siren:ISiren.TurnOff; WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating	*Turn siren off, deactive sensor*	
61	**EntryError (floating state)**							
64	IAlarmSystem	SwitchOn+		IAlarmSystem.Failed		EntryError		
68	WindowSensor:ISensor_NI	DetectedMovement		NoOp		EntryError		

Colours used in the SBS tab

The blue, grey or orange rows indicate a state. The rows under a state row list all the possible triggers to the component and the corresponding actions when the triggers occur in that state. The orange row in the previous figure indicates a "floating" state. These are states that are not reachable from the initial state. In the example above, the EntryError state is not present anywhere in the "Target State" column, and thus is not reachable from the initial state.

Blue states are Normal States or the Initial State. Grey states are either Synchronous Return States or Super States. Orange states are either floating (there are no transitions to it) or ambiguous (there are both valued and non-valued transitions to it).

The green cells in the "Target State" column indicate that the respective rule case defines the first transition to that particular state. This identifies the shortest possible sequence of triggers to that particular state (the canonical sequence).

The light-blue cells in the "Target State" column (e.g. line 33 in the previous figure) indicate a transition to the same state (called "self-transition").

The light-grey line indicates the currently "active" rule case (e.g. line 32 in the previous figure).

The dark-grey cell/line indicates the currently selected cell/line.

The context field above each SBS

The ASD:Suite provides detailed information about a selected field in the SBS tab in an information (non-editable) field located just above the SBS table.

The left part of the context field shows the state to which the currently selected rule case belongs. This is handy if the state row is scrolled out-of-view.

The right part of the context field shows extra information about the currently selected cell. For instance, if you select a cell in the "Event" column, it shows the event declaration including the parameter directions and parameter types.

AlarmNotActivated API.Digit([in]digit:int): valued

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**AlarmNotActivated (initial state)**					
4	API	Set+		Sensor:ISensorActivate+		AlarmActivating
5	API	Clear		Illegal		-
6	API	Digit(digit)+		Illegal		-
9	Sensor:ISensor_NI	DetectedMovement		Illegal		-
10	Sensor:ISensor_NI	Deactivated		Illegal		-
11	Sensor:ISensor_NI	asd_Unsubscribed		NoOp		AlarmNotActivated
14	Timer:ITimerCB	Timeout		Illegal		-

The Context information field in the SBS tab

The File menu

Menu Item	Purpose
New	Create a new ASD model. See "Creating an Interface Model" or "Creating a Design Model" for details.
Open	Open an ASD model or a model verification results file, closing the current model(s).
Open built-in ITimer model	Open the built-in ITimer model (this cannot be loaded from disk using the regular "Open" menu item), closing the current model(s)
Add Model(s)	Load more models (read-only) into the ModelBuilder, without replacing the existing ones.
Save	Save the current ASD model.
Save <model_name> As...	Save an exact copy of the currently selected model or to create a new model based on the current one. See "Save As" for details.
Save to .zip file	Save the main model (and related IMs if it is a design model) to a .zip file.
E-mail model	Starts your default e-mail client and attaches a .zip file with the main model and any related IMs. Note that you need a configured e-mail program to use this.
Close Model	Close the selected model.
Close All	Close all models.
Print	Print (parts of) the current ASD model.
Open in Model Navigator	Open the map in which the selected model occurs. See "Model Navigator" for details.
Properties	Open the Properties dialog of the active model. **Note:** This dialog is used for specification of properties to be used in verification and code generation.
Open in ASD:Suite ModelCompare	Open the current model in the ASD:Suite ModelCompare tool, so you can compare it to another (version of the) model. **Note:** o The selected model is loaded as the Master. o This option is greyed out (disabled) if the ASD:Suite ModelCompare is not installed or there is no ASD model loaded. See the "ASD:Suite ModelCompare User Guide" for details.
Open in a new instance of ASD:Suite ModelBuilder	Open the selected model in a new instance of the ASD:Suite ModelBuilder.
Open in this instance of ASD:Suite ModelBuilder	Set this model as the main model.
Upgrade Models	Upgrade models made with older versions of ASD:Suite to the current version. See "Upgrading ASD models" for details.
Exit	Exit the application.

Note: In addition to the presented items in the File menu you will see a list of recently opened models. You can set the maximum number of models to be listed by specifying the size of the recent models list in the Options dialog obtained via "Tools-->Options" under the "Appearance" tab.

The Edit menu

Menu Item	Purpose
Undo	Undo an action.
Redo	Redo an undone action.
Find	Search for text in (part of) the ASD model.
Replace	Replace text in (part of) the ASD model.
Clear Find Results	Clear the Find Results table and the green highligting after a Find All.

The View menu

Menu Item	Purpose
Toolbars	Show/hide each of the toolbars
Model Explorer	Open or close the "Model Explorer", which shows all loaded models.
Output Window	Open or close the "Output Window", which shows status and progress.
Conflicts	Open or close the "Conflicts" window, showing specification conflicts. See "Conflicts" for details.
Find Results	Open or close the "Find Results" window, showing the results of a Find All operation.
Verification Results	Open or close the "Verification Results" window, which shows the progress and results of verifying a model. See "Verification" for details.
Visual Verification	Open or close the "Visual Verification" window, which shows information about the current verification error in the model. See "Verification" for details.
Un-saveable Data	Open or close the "Un-saveable Data" window. See "Un-saveable Data" for details.
Model Navigator	Open or close the "Model Navigator" window, which shows the relationships between ASD models in a given directory. See "Model Navigator" for details.

The Filters menu

Menu Item	Purpose
Hide Illegal	Hide all rule cases that have "Illegal" as only action. See "SBS Filters" for details.
Hide Blocked	Hide all rule cases that have "Blocked" as only action. See "SBS Filters" for details.
Hide Disabled	Hide all rule cases that have "Disabled" as only action. See "SBS Filters" for details.
Hide Invariant	Hide all StateInvariant and DataInvariant rule cases. See "SBS Filters" for details.
Hide Self Transitions	Hide all rule cases that have their own state as Target State. See "SBS Filters" for details.
Filter by Text / Expression	Show only rule cases that match the filter criteria specified in the filtering toolbar (either text or filter expression). See "SBS Filters" for details.
Toggle Expression/Text Mode	Switch between interpreting the text in the filtering toolbar edit as basic text or as an expression. See "SBS Filters" for details.
Jump to Expression Edit	Set keyboard focus to the filtering edit box in the toolbar. See "SBS Filters" for details.
Expression Builder	Show a dialog to help defining an expression to filter on. See "SBS Filters" for details.
Filtering Help...	Show this help page: "SBS Filters".
Turn All Filters Off	Show all rule cases. See "SBS Filters" for details.
Apply Filters	Re-apply all filters to the SBS. See "SBS Filters" for details.

The Session menu

Menu Item	Purpose
Connection Status	Show details about the connection to the ASD:Server. See "Session Management" for details.
Log Out	Stop the active session with the ASD:Server. See "Session Management" for details.
Store Login Settings...	Store the current login settings to the registry for easy switching between different settings. See "Saving Connection Settings" for details.
Delete Login Settings...	Removes previously stored login settings from the registry. See "Saving Connection Settings" for details.

The Verification menu

See the "ASD:Suite Visual Verification Guide" for details about interactive visual verification.

Menu Item	Purpose
Verify...	Verify the selected ASD model. See "Verification" for details.
Verify With...	Same as Verify, but this asks you for a code generator version and language. **Note**: use with great care; if the generated code is created using a different version than the version with which the model is verified, there is no guarantee that the generated code will be error-free.
Verify All	Run all checks for the selected ASD model. See "Verification" for details.
Verify Again	Re-run the last verification. See "Verification" for details.
Open Verification Results	Open a model verification results file, containing the details of an model verification that was performed earlier.
Stop Verifying	Abort a verification.
Show Previous Failure	Show in the Visual Verification window the first example of the previous failed check.
Show Next Failure	Show in the Visual Verification window the first example of the next failed check.
Forward Step Over	Step over thStep over the current item in the currently focused SBS tab (i.e. do not step into the SBS of a sub machine or a used service).
Forward Step Into	Step into Step into the SBS of a sub machine or a used service from the current item in the currently focused SBS tab.
Forward Step Out	Step out froStep out from the SBS of a sub machine or a used service to the next item in the main machine.
Forward Step Rule Case	Step to the Step to the next rule case in the currently focused SBS.
Backward Step Over	Step backwards over the current item in the currently focused SBS (i.e. do not step into the SBS of a sub machine or a used service).
Backward Step Into	Step backwarStep backwards into the SBS of a sub machine or a used service from the current item in the currently focused SBS.
Backward Step Out	Step out froStep out from the SBS of a sub machine or a used service to the previous item in the main machine.
Backward Step Rule Case	Step to the Step to the previous rule case in the currently focused SBS .
Step To First	Step to the Step to the first item in the trace.
Step To Last	Step to the Step to the last item in the trace (which is typically the error (warning sign)).

The Code Generation menu

Menu Item	Purpose
Generate Code	Generate code from the current ASD model. See "Generating code from an ASD model" for details.
Generate Code With...	Same as Generate Code, but this asks you for a code generator version and language. See "Generating code from an ASD model" for details.
Generate All Code	Generate code for all loaded models (except ITimer). See "Generating code from an ASD model" for details.
Generate Stub...	Generate stub code from the selected Interface model, to build your own implementation of the interface. See "Generating stub code from an ASD interface model" for details.
Download Runtime...	Download the ASD Runtime. See "Downloading the ASD Runtime" for details.
Determine Model Size	Show the size of your model in ASD Function Points in the Output Window.
Stop Generating	Abort a code generation.

The Tools menu

Menu Item	Purpose
Check Conflicts	Check the ASD model for specification conflicts. See "Conflicts" for details.
Clear Conflicts	To clear the Conflicts window and the orange and yellow highlights in your model. See "Conflicts" for details.
Options	To change the global settings for the Appearance, Model Verification and File Association properties within the ASD:Suite.

Status bar

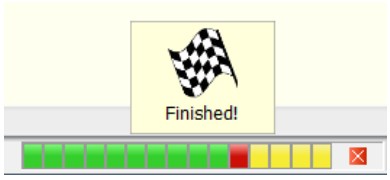
The status bar is located at the bottom of the main window.

The following information is shown in the status bar, from left to right:

- Verification progress bar. Via this progress bar you can follow the progress of verification. The following figure shows the progress of verification,

Verification progress in the status bar

while the next figure shows the reporting of a verification end:

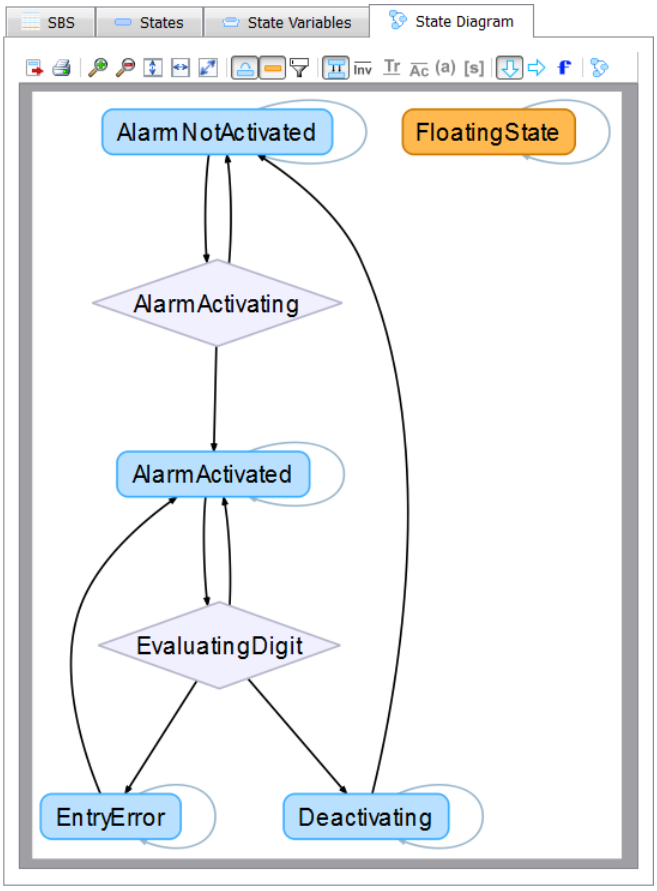


Verification end shown in the status bar

Note: The number of the rectangles shown in the verification progress bar is the same as the number of (to be) performed checks. The following list contains an explanation for the items which appear in the verification progress bar:
  - The red cross: the button to stop verification or debugging
  - A green rectangle: a successful check
  - A red rectangle: a failed check
  - A grey rectangle: a check waiting to be performed
  - A light blue rectangle with a running circle: a performing check
  - A blue rectangle: a failed check due to an internal error
  - A yellow rectangle: a skipped check
- The target language for code generation and the version of the generator to be used. Possible values of this information field:
- Empty: when no model is opened.
- LANGUAGE (VERSION): when an ASD model is opened but neither the target language nor the code generator version are specified.
- <target_language> (<generator_version>): For example "C (8.4.0)" when target_language is "C" and the generator_version is "8.4.0".
  Note: Code generation language and version properties are captured in the model properties section and are stored in the ASD model file. This allows you to specify the desired target language and code generator version for each model when information is missing or not according to your expectations.
  See "Specifying target language and code generator version" for details about the selection and specification of the target language and code generator version.
- The session status: Authorized, Logging in, Logging out, or Not Authorized. This detemines if you are allowed to use the ModelBuilder.
  See "Session Management" for details.
- The connection status: Connected, Disconnected, or Reconnecting. This determines whether you can generate code, verify models or download the ASD Runtime.
  See "Session Management" for details.

State Diagram

The State Diagram tab in the Model Editor contains a graphical representation of the selected Main Machine or Sub Machine:



State Diagram

The colouring scheme for the states correspond to the colouring scheme in the SBS: Normal states are indicated in blue, super states have a lighter colour, synchronous return state have a diamond shape to indicate a choice is being made, and floating state are coloured orange. Double-clicking on a state takes you to that state in the SBS tab.

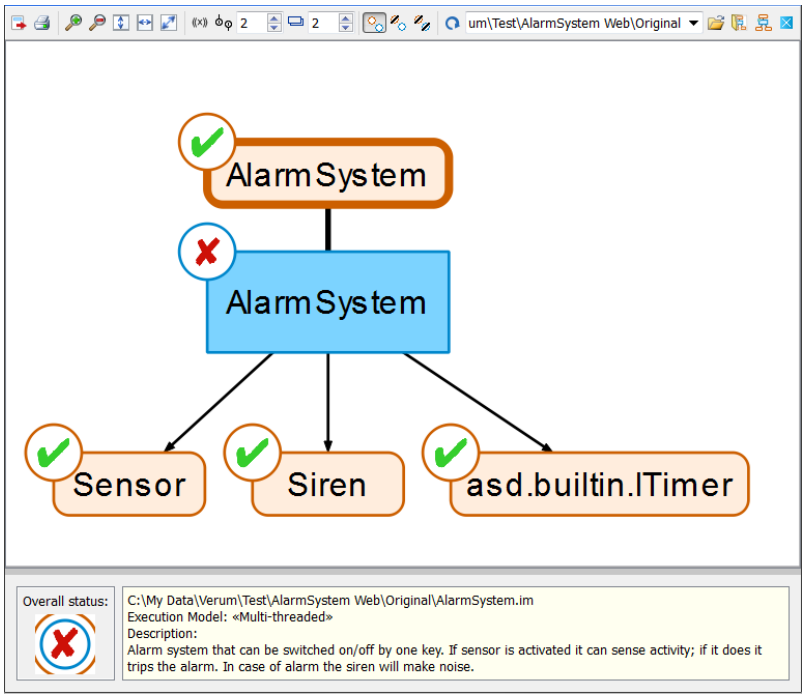
The State Diagram tab has its own toolbar, where the various buttons have the following meaning:

Picture	Name	Purpose
	Export	Export the state diagram for the selected machine to an image file. Note that you can control the resolution (DPI) of the exported image from the Tools-Options dialog.
	Print	Print the current state diagram.
	Zoom In	Zoom-in in the current state diagram.
	Zoom Out	Zoom-out in the current state diagram.
	Fit height	Fit the state diagram in the available window height.
	Fit width	Fit the state diagram in the available window width.
	Fit page	Fit the state diagram in the available window size.
	Show self transitions	Enable/disable showing of self transitions in the state diagram.
	Show floating states	Enable/disable showing of floating states in the state diagram.
	Follow custom filter	Display data in accordance with the custom filter settings. For example, if there is a custom filter defined and it is selected, and the "Follow custom filter" button is selected, the filtered out data is not shown in the state diagram.
	Merge transitions	Shows the state diagram with only single transitions between states, and no transition labels. This is the main switch. If disabled then "Show triggers", "Show actions", "Show arguments" and "Show guards" are enabled.\
Inv	Show Invariants	Enable/disable showing of invariants with the states in the state diagram
Tr	Show triggers	Enable/disable showing of triggers in the state diagram.
Ac	Show actions	Enable/disable showing of actions in the state diagram.
(a)	Show arguments	Enable/disable showing of arguments in the state diagram.

[s]	Show guards	Enable/disable showing of guards and state variable updates in the state diagram.
	Ordering top to bottom	Change the orientation of the state diagram to "top to bottom".
	Ordering left to right	Change the orientation of the state diagram to "left to right".
	Set fonts	Change the font settings for the data displayed in the state diagram.
	Refresh state diagram	Refresh the data displayed in a state diagram after a change in the SBS of the associated machine.

Model Navigator

With the Model Navigator you can show an overview of all ASD models in your project and the dependencies between them. From the Model Navigator you can open the model in the ModelBuilder and check their verification status and other properties. To generate a Model Navigator map, fill in one or more directory paths (separated by semi-colons) into the combobox on the toolbar and press ENTER. Alternatively, press the "..." button to get a dialog to select directories. In this dialog, you can also save sets of directories under a name of your choice. The result is an overview of all models in your project:



Model Navigator

The Design models are indicated in blue, the Interface models are indicated in orange. A thick border indicates the model that is currently selected in the Model Navigator. The dark fill colour indicates the main model that is currently opened in the ModelBuilder.

The design models and interface models can have a verification status indicator, showing the result of the last verification. This verification status is saved into the models and refreshed in the Model Navigator after verification. This indicator can have four possible values:

- Green tick-mark: the model has been verified successfully, no errors are found.
- Red cross: the model verification failed; when you double click on the red cross icon, the corresponding verification results will be loaded in the ModelBuilder.
- Red open circle: only the "relaxed livelock" check of the model verification failed; this could be a genuine livelock error, but could also report a livelock where none would occur in a real system (for instance when a "polling loop" is used). When you double click on the red circle icon, the corresponding verification results will be loaded in the ModelBuilder.
- Blue question-mark: the verification status cannot be determined; this can either mean that:
  - The model has not been verified before
  - In the last verification not all checks were selected
  - The verification results are outdated because of a significant change in either the design model or one or more of the related interface models. Note that e.g. changing comments does not affect verification status.

Tips:

- Double-click a model to open it.
- Double-click a verification status to debug a problem.
- Right-click a model for more options.
- You can verify all models in a directory structure with the command-line tools, more specifically with AsdVerify.exe.
- In your build system, you could integrate checking that all models are verified or that all code is generated from verified models using AsdVerify.exe.
- You can generate code for all models in a directory structure with the command-line tools, more specifically with AsdGenerate.exe.

The Model Navigator window has its own toolbar, where the various buttons have the following meaning:

Picture	Name	Purpose
	Export	Export the model overview to an image file. You can control the resolution (DPI) of the exported image file from the Tools-Options dialog.
	Print	Print the current model overview.
	Zoom In	Zoom-in in the current model overview.
	Zoom Out	Zoom-out in the current model overview.
	Fit height	Fit the model overview in the available window height.

	Fit width	Fit the model overview in the available window width.
	Fit page	Fit the model overview in the available window size.
	Show stereotypes	Toggle the display of stereotype information in the model boxes.
	Decouple Interface Models	To avoid a lot of crossing arrows, you can decouple interface models that are used by multiple design models in the overview. You can set the value for the decoupling threshold in the selection box.
	Group Design Models	To maintain a high-level overview of your (sub-) systems, you can group design models that implement the same interface model. You can set the value for the grouping threshold in the selection box.
	Show/Hide verification status	With these buttons you can turn the circles with verification statuses on or off (all / DM only / DM +IM). Note that the overall status which is displayed in the lower left corner of the Model Navigator responds to this setting as well.
	Refresh Map	Refreshes the previously generated diagram while keeping the zooming settings intact.
	Select single folder to show	Allows to select a directory and generates a diagram of all models in this directory and its subdirectories
	Select multiple folders to show	Allows to select multiple directories. Generates a diagram of all models in these directories and their subdirectories
	Generate	Generates a diagram of the directory or directories in the combo box on the left.
	Clear Map	Clears the model navigator diagram.

SBS Filters

There are several kinds of filters you can use to hide and show exactly what you need in the SBSes.

Basic Filters

- **Hide Illegal**
  Hides all rule cases that have "Illegal" as Actions.
- **Hide Blocked**
  Hides all rule cases that have "Blocked" as Actions.
- **Hide Disabled**
  Hides all rule cases that have "Disabled" as Actions.
- **Hide Invariant**
  Hides all the StateInvariant and DataInvariant rule cases.
- **Hide Self Transitions**
  Hides all rule cases that have their own state as Target State.
- **Filter by Text**
  You can filter the SBS on any text by typing into the text edit on the Filtering toolbar. Just type any text and press Enter to show only rule cases containing that text. The text is not case sensitive. Note that this filter switches off automatically when you load a new model.

Expression Filters

It is also possible to make a more intelligent filter by typing a Boolean expression. For instance, to filter on a name "myVariable" only in the Guard column, type:

guard contains "myVariable"

This shows all rows in the SBS where the Guard column contains the text "myVariable". To also show rows where the State Update column contains the variable, use Boolean connectives. Note also we use **containsword** instead of **contains** to match whole words only:

guard containsword "myVariable" **or updates containsword** "myVariable"

And this is how you find all rule cases having self transitions (i.e. the source state is the same as the target state):

source equals target

Regular expressions are also supported. The following will match all events in the Event column that start with "Switch", e.g. "SwitchOn" and "SwitchOff":

event matches "Switch.*"

Note that the syntax highlighting only comes on either when you press the "Toggle Expression/Text Mode" button, or after the first correct expression is entered. The edit box has two modes (expression or text), and it tries to switch between them based on what you do. You can override this by using "Toggle Expression/Text Mode".

The column names you can use are:

- **source**: the source state (the state the rule case is in)
- **interface**: the Interface column
- **event**: the Event column
- **guard**: the Guard column
- **actions**: the Actions column
- **updates**: the State Variable Updates column
- **target**: the Target State column
- **comments**: the Comments column
- **tags**: the Tags column

The relational operators are:

- **contains**: true if a cell contains the given string somewhere (case sensitive)
- **containsword**: similar to contains, but matches whole words only
- **equals**: true if a cell equals the given string exactly (case sensitive)
- **matches**: true if a cell matches a given regular expression (case sensitive)

The Boolean operators are:

- **or**: true if either operand is true
- **and**: true if both operands are true
- **xor**: true if exactly one operand is true
- **not**: negates the expression

String literals are surrounded by double-quotes. Escaping characters is done in C-style. You can use the following escape sequences:

- **\n**: newline
- **\r**: carriage return
- **\t**: tab
- **\"**: double quote
- **\'**: single quote
- ****: backslash
- **\b**: backspace

- **\f**: formfeed

Regular Expressions

The **matches** operator takes a string containing a Perl-style regular expression. The precise syntax can be found on this external web page. A short overview is given below. Note that the regular expression you enter may still have characters that need to be escaped according to the rules in the previous paragraph.

In Perl regular expressions, all characters match themselves except for the following special characters:

`.[{()\*+?|^$`
The '.' character matches any single character. The '^' character matches the start of a line, and the '$' characters matches the end of a line.

(Sub-)expressions can be repeated, e.g. 'a+' matches any sequence of one or more 'a' characters. Below is a list of repeat operators. Note that all of these are "greedy" - add a question mark at the end to make them non-greedy (e.g. '*?' is the non-greedy version of '*').

- '*' matches the preceding atom zero or more times
- '+' matches the preceding atom one or more times
- '?' matches the preceding atom zero or one time
- '{n}' matches the preceding atom *n* times
- '{n,}' matches the preceding atom *n* or more times
- '{n,m}' matches the preceding atom between *n* and *m* times inclusive

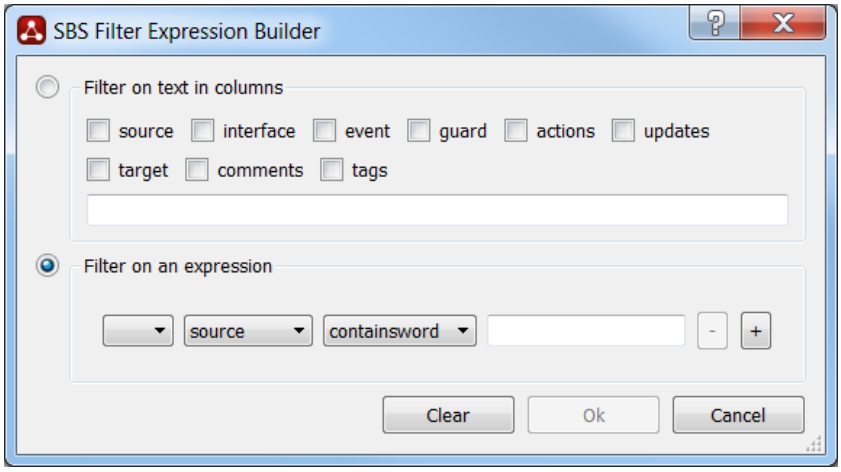
Alternation is specified using '|', e.g. 'abc|def' matches either 'abc' or 'def'

Character sets are defined between square brackets, e.g. '[abc]' matches 'a', 'b', or 'c'. Sets can be negated by adding a '^' after the opening bracket, e.g. [^a-c] matches everything *except* a, b or c. There are predefined character classes which can be included between brackets, e.g. '[[:digit:]a]' matches any decimal digit and the 'a' character.

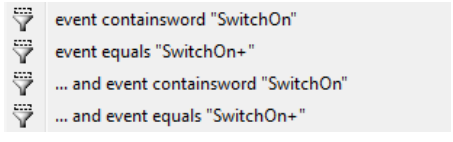
- [:alnum:] any alphanumeric character
- [:alpha:] any aphabetic character
- [:blank:] any whitespace character that is not a line separator
- [:cntrl:] any control character
- [:digit:] any decimal digit
- [:graph:] any graphical character
- [:lower:] any lower case character
- [:print:] any printable character
- [:punct:] any punctuation character
- [:space:] any whitespace character
- [:upper:] any upper case character
- [:word:] any alphanumeric character or the underscore
- [:xdigit:] any hexadecimal digit

Creating Expressions Easily

It is usually not necessary to type expressions manually. There are several dialogs and features that help build expressions for you.
Expression Builder
Using the "..." button next to the filtering edit box, you get a dialog that you can use to build expressions. The dialog has two parts. You can use either part to make an expression. The top part allows to easily filter multiple columns on the same text. The bottom part allows to make expressions by selecting operators from combo boxes.



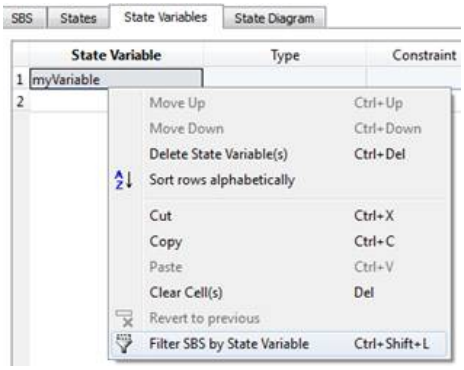
Filter Suggestions
Right-clicking any cell in the SBS yields one or more filter suggestions at the bottom of the context menu:



Clicking on a filter suggestion sets it as an Expression filter. Note that it is also possible to select multiple cells and get a filter suggestion based on that. Handy filters are:

- Select two cells in the Event column to see a filter to show only those events in every state.
- Select two states and use the offered suggestion to show only those states
- Select a cell in the Interface column to get a suggestion to filter on that interface in both the Interface and Actions columns. This gives an overview of where that interface is used.

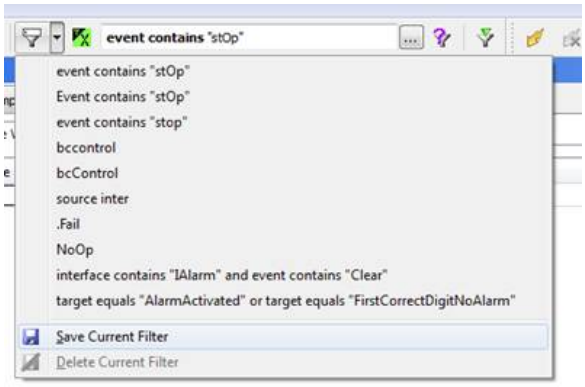
Filtering from Other Tables
Next to the SBS, most other tables also offer filter suggestions (e.g. States, State Variables, Data Variables, Tags, Service References, and the various Events tables).:



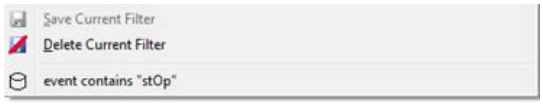
Reusing Filter Expressions
Filter expressions (and filter text) is automatically stored in a most-recently-used list. You can access it in two ways:

1. Click the filtering edit box and use the Up and Down arrow keys to scroll through the list.
2. Through the drop-down menu of the Filter Text button.

Next to the most-recently-used list, you can also store expressions permanently. To do so, click the Save Filter button to save the current filter.



This adds the filter to a special section of the drop-down menu, where you can select it:



To delete a saved filter, first select it and then use "Delete Current Filter".

Un-saveable Data

Not everything that you can type in a table cell can be stored in the ASD model format. For example, a syntactically incorrect event declaration cannot be stored. Whenever something is typed that cannot be saved, the table cell gets a red border.

Fixing un-saveable cells

If you see a table cell with a red border, there are three things you can do:

1. First of all, you can edit the cell to be syntactically correct.
2. Alternatively, you can use Edit-Undo to undo the incorrect change.
3. However, if the change that made the cell red was not the last thing on the undo stack, you can also right-click the cell and choose "Revert to previous". This resets the cell to the previous correct value.

Un-saveable Data window

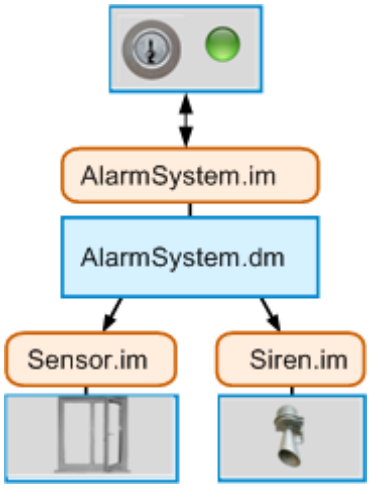
When you try to save, verify, conflicts check, or generate code from a model with un-saveable data, you get a warning. To locate all the red cells in your model, look at the "Un-saveable Data" window. This window shows the current and previous cell values. Double-clicking an entry in the window sets focus to the cell in question.

Modelling

To build an ASD component using the ASD:Suite, follow these steps:

1. Create an interface model that specifies the service that is going to be implemented by the component. See "Creating an interface models" for details.
2. Identify the used services, i.e. services that provide functionality that is going to be used in the design of the component.
3. For each used service, identify one or more components that implement that service.
4. Create the design model for the considered component:
   - Specify the service that is going to be implemented.
   - Specify the services that are going to be used.
   - Designate components for the used services.
   - Build the SBS for the design model.

The construction of an Alarm System is used to illustrate the above mentioned steps. The following figure presents the main component of such a system (the box named "AlarmSystem"), together with the service to implement, AlarmSystem, and the services to be used: Sensor and Siren:

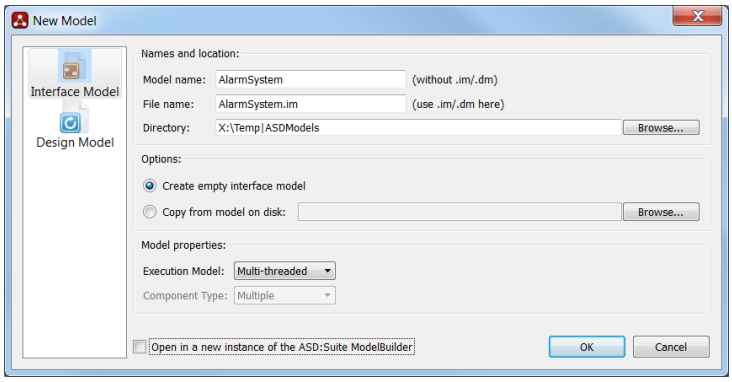


Creating an Interface Model

These are the steps to create a new interface model:

1. Open the "New Model" dialog. This can be done by:
   - Selecting the "File->New" menu item, or by
   - Clicking "New" in the Toolbar, or by
   - Selecting the "Create a new model..." item on the *Start Page*.

2. Specify a name for the service under "Model name:"
   Note:
   - If no file name was specified yet, the model name you type will also be copied to the File Name field.
   - You can enter a namespace before the model name (e.g. "alarmsystem.sensors.Sensor". Here "alarmsystem.sensors" is a namespace and "Sensor" is the model name.
   - If a model was already loaded, the namespace of this model (if any) is filled in in the Model Name field.



The "New Model" dialog for interface models after specifying a service name

3. Specify the execution model for this interface model. See "Specifying the execution model" for details.
4. Click "OK" to create and save the interface model.

Note:

- To change the name of the service, select the service name in the "Model Explorer", press F2 or double click, and type in a new name. This does not change the name of the file.
- The following tabs are shown in the "IAlarmSystem (AlarmSystem.im)", which is the "Model Editor" for the IAlarm.im:
  - ***IAlarmSystem***: a tab for the main machine, containing the following sub-tabs:
    - ***SBS***: shows the SBS for the machine;
    - ***States***: shows the list of states defined in the machine and facilitates the specification of informal design information about the states;
    - ***State Variables***: shows the list of state variables defined for the machine and facilitates the declaration and specification of state variables.
    - ***State Diagram***: shows the state diagram for the machine.

    Note: In an interface model there is only one machine.
  - ***Application Interfaces***: shows the set of call events and reply events for each defined application interface and facilitates the specification of new application interface events.
    Note: There is one sub-tab per defined interface.
  - ***Notification Interfaces***: shows the set of events for each defined notification interface and facilitates the specification of new notification interface events.
    Note: There is one sub-tab per defined interface.
  - ***Modelling Interfaces***: shows the set of events for each defined modelling interface and facilitates the specification of new modelling events.
    Note: There is one sub-tab per defined interface.
  - ***Type Definitions***: state variable type definitions.
  - ***Tags***: shows the list of requirements defined for the component and facilitates the specification of additional requirements that emerge during the design phase.

- Names and identifiers must adhere to these syntactical rules.

Interfaces and Events

The first thing to do when creating an interface model, is to define its interfaces and events. An event is analogous to a method or callback that your component exposes. An interface groups multiple events together.

Application Interfaces and Application Events

All interface models must have at least one application interface.

- Go to the "Application Interfaces" tab and click the blue plus sign to create a new interface.
- In the dialog that appears, type a name and click OK. The name must begin with a letter and continue with letters, underscores or numbers
- Click OK. A new tab appears for the application interface.

Application interfaces have two types of events: Call Events and Reply Events. You see a table for each.

Call Events

The declaration of a call event consists of:

- A name: this identifies the call event
- Parameters (optional): used for passing arbitrary data through ASD components, see Parameters.
- A return type: valued or void

An example of a call event is `DoSomething([in]p1:string, [out]p2:int):void`. Here, "DoSomething" is the event name, p1 and p2 are parameters, and "void" is the return type.

Parameters are described in detail in Parameters.

The return type determines what the possible reply events are for the call event. A call event with a "void" return type always has the standard "VoidReply" reply event. A call event with a "valued" return type can use all user-defined reply events.

Reply Events

Reply events are the possible return values to the call events. Usually they are translated to enumeration types in the generated code. When you create an application interface, you get one reply event for free: "VoidReply". This is a special event that is used in the SBS as a reply to void call events. Next to this event, you can create your own reply events for any valued call events (e.g. Yes, No, Ok, Fail etc).

Note:
If you have no void call events, but only valued ones, you can optionally delete the "VoidReply" event.

Reply events are declared merely by typing a unique name.

Notification Interfaces and Notification Events

Notification events are analogous to callbacks in a programming language. In ASD, you define a notification interface as follows:

- Go to the "Notification Interfaces" tab and click the blue plus sign to create a new interface.
- In the dialog that appears, type a name and click OK. The name must begin with a letter and continue with letters, underscores or numbers
- Click OK. A new tab appears for the notification interface.

At the top of the notification interface tab there is a checkbox marked "Broadcast". If this is checked, the notification interface can be used by multiple clients and the clients can subscribe and unsubscribe to its events at run-time. This is described in more detail in Broadcasting Notifications.

Notification Events

The declaration of a notification event consists of:

- A name: this identifies the notification event
- Parameters (optional): used for passing arbitrary data through ASD components, see Parameters.

An example of a notification event is `ProcessingDone([in]result:string)`. Here, "ProcessingDone" is the event name, and "result" is a parameter.

Parameters are described in detail in Parameters. Notification events can only have [in] parameters.

The Notification Events table has a column called "Yoking treshold". This column is only useful if the interface model is a specification of a foreign component. You use it to say that the notifications will arrive at a slow enough rate to be processed by clients without their queue becoming full. It is a promise to the outside world. The yoking treshold is an integer number that indicates the maximum number of events that will end up in the client's queue at any given time. This is described in more detail in Yoking Notification Events.

Note:
Yoking makes a promise to the user of the interface model, which ASD cannot verify. Therefore, if the foreign component does not adhere to the promise, ASD model verification will not catch this. Often, it is possible to use safer constructs such as singleton notification events or to define a protocol that limits the notifications.

Modelling Interfaces and Modelling Events

If the interface model has so-called spontaneous behaviour (e.g. it sends a notification event without being told to do so by the outside world), you need a *modelling event* to be able to specify this in the SBS. After all, every line in the SBS needs a trigger but there is no outside event to be the trigger. Modelling events are not real events, but only used for modelling.

- Go to the "Modelling Interfaces" tab and click the blue plus sign to create a new interface.
- In the dialog that appears, type a name and click OK. The name must begin with a letter and continue with letters, underscores or numbers
- Click OK. A new tab appears for the modelling interface.

Modelling Events

Modelling events are declared just by typing a name in the Modelling Event column.

Next to the Modelling Event column, there is a column called "Abstraction Type". Each modelling event is either *Inevitable* or it is *Optional*.

Optional means that the event may or may not occur.

Inevitable means that when no other events occur, this event will occur. Note that an Inevitable event is not guaranteed to occur always. If there are other (optional or inevitable) events that can occur, the *inevitable* event may or may not occur. So it is only *inevitable* in the absence of other events.

The difference between the two is how the modelling event is handled by the model verification. For inevitable modelling events, the verification only checks the cases where the event does occur, i.e. it assumes that the event will eventually occur if no other events occur. For optional modelling events, the model verification also checks the cases where the event never happens (which could be a cause of deadlock).

Note: when a combination of *optional* and *inevitable* events is used in a single state, then it is only guaranteed that one of the modelling events will occur, but it is not guaranteed that the *inevitable* event will always occur. Similarly, when more than one *inevitable* modelling event is used in a single state, is it guaranteed that one of the *inevitable* events will always occur, but the other may never occur.

Yoking Notification Events

Yoking only applies to the Multi-threaded execution model.

The ASD:Suite enables you to specify the maximum number of occurrences at any given time in the queue of notification events. In ASD this number is called the Yoking Threshold. This number can be determined after a thorough analysis of the timing-behaviour of the system and after determining the arrival- and service intervals of the queue.

Many real world designs must accept notification events generated by the environment and which can not be controlled by the design. The real executing system works without problems because the arrival intervals of the notification events are much longer than the service times and thus, they do not flood the queue or starve other system activity. Also, in reality, the queue is always "long enough" that it never becomes blocking.

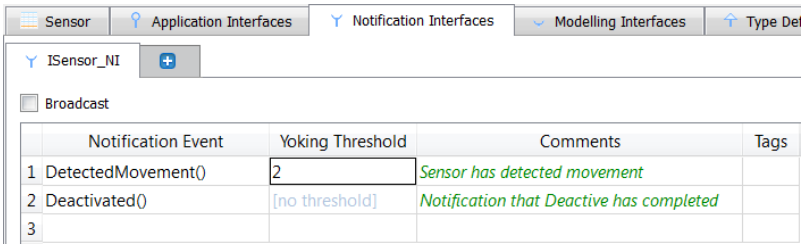
In verification using the ASD:Suite there is no direct way of representing this temporal behaviour and the queue must be kept small enough in size to avoid state explosion and endless verification. The remedy for this is based on the "yoking" concept. The idea is to approximate the effect of temporal behaviour by limiting the number of such notification events that can be in the queue at any given time. Such a limit is called an event threshold and is specified as "Yoking Threshold" for each notification event to be limited / restricted.

Conceptually, by specifying an event threshold for a notification event you state that under expected runtime conditions, the arrival rate and service time of the event are such that the number of unprocessed notification events of the specified type will never exceed the specified limit.

Notification events originate from interface models of used services and are either an action to an application interface trigger or an action to a modelling event. ASD semantics require that executing a notification as an action is never blocking so the notification can not be prevented, instead the trigger that results in the notification is prevented. The solution is to limit notifications that are actions to modelling events by enabling and disabling the modelling events. Notification events arising as actions to application interface triggers can not be limited because the trigger can not be disabled.

WARNING: Yoking should be used with caution! When the assumptions about arrival intervals and service intervals are not correct, then the verification results may give a false impression. If the arrival rate during execution is higher than assumed with Yoking, it is possible that the queue overflows. In the C code this leads to a runtime assertion since the queue size in C is fixed. In the other languages the queue size is only bound by memory and therefore it could be less critical. Further, if the arrival rate during execution is higher than assumed with Yoking, it is also possible that the respective component handling the yoked notifications seems to be deadlocked, but actually the component is diverging as the queue always gets priority.

The ModelBuilder enables the indication in the interface model for each notification event whether the respective event should be restricted by means of Yoking. For each notification event you can indicate the event threshold by typing a value in the "Yoking Threshold" column:



Client notification with yoking threshold

Note:

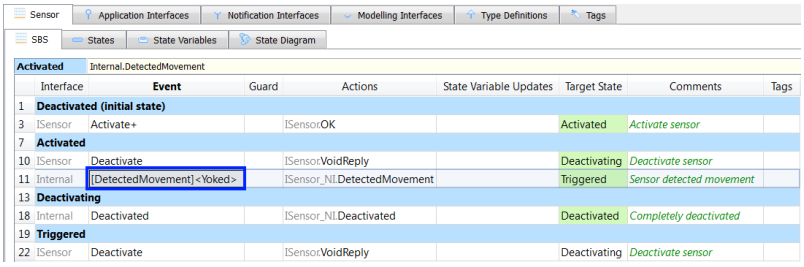
- A threshold should be smaller than or equal to the queue-size in the using design model (otherwise a queue size violation modelling error can still occur).
- If the threshold is set to zero or remains empty, this implies there is no threshold, and the notification event is not restricted.
- The Yoking Threshold is to be specified in the interface model because in effect it is a statement about the frequency with which the implementation of the interface model will generate notification events.
- The Yoking Threshold has to be specified per notification event and NOT per triggering modelling event. This is because of the same reasons as for the previous point; it is a statement of the frequency with which a notification event will be generated.

A modelling event will be yoked (i.e. the corresponding rule case will be disabled) if the number of any of the restricted notification events in the sequence of actions already in the queue is larger than or equal to the defined event threshold (irrespective of the total number of notification events in the queue). In this case, the complete rule case is disabled, and thus no response is triggered, no state variable update is performed and the specified state transition will not occur.

> **Note**: A rule case with a non-modelling event as trigger will never be disabled, i.e. there will be no check to see if the number of any of the restricted notification events in the sequence of actions already in the queue is larger than or equal to the defined event threshold.

A modelling event will NOT be yoked (i.e. the corresponding rule cases will NOT be disabled) when the number of all of those restricted notification events already in the queue is less than the defined event threshold for each restricted notification event respectively.

The following figure shows an SBS in which the modelling event is marked as <yoked> since in the Actions column a yoked notification event is specified, in this case the DetectedMovement notification:



Yoked modelling event

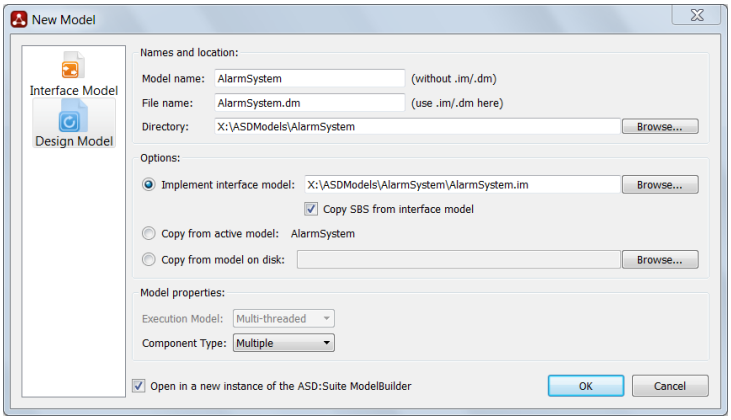
Creating a Design Model

Note: In order to create a design model, you must already have created the interface model for the implemented service. See "Creating an interface model" for guidelines on creating an interface model.

These are the steps to create a new design model:

1. Open the "New Model" dialog. This can be done by:.
   - Selecting the "File->New" menu item, or by
   - Clicking "New" in the Toolbar, or by
   - Selecting the "Create a new model..." item on the *Start Page*.
2. Select the design model icon in the left pane.
3. Specify a name for the design model in "Model name:"
   Note:
   - If no file name was specified yet, the model name you type will also be copied to the File Name field.
   - You can enter a namespace before the model name (e.g. "alarmsystem.sensors.Sensor". Here "alarmsystem.sensors" is a namespace and "Sensor" is the model name.
   - If a model was already loaded, the namespace of this model (if any) is filled in in the Model Name field.

   The following figure shows the "New Model" dialog after you have selected the design model icon in the left pane and you specified "AlarmSystem" as name for your design model:



The "New Model" dialog for creating design models

4. Specify the implemented service by filling in the path or by selecting the file using the "Browse..." button next to the "Implement interface model:" field.
   Note: You are able to specify if you would like to create the SBS of the design model based on the SBS of the specified implemented interface model. This can be done by checking the "Copy SBS from interface model" checkbox.
5. Specify the component type for the ASD component. See "Specifying component type" for details.
6. Click "OK".
   When you click OK, a new design model is created, saved and opened.

Note:

- To change the name of the component, double click on the component name and type a new name. You may also do this by selecting the component name in the "Model Explorer" window and pressing F2. This does not change the name of the file
- The following tabs are shown in the "AlarmSystem (AlarmSystem.dm)", which is the "Model Editor" for the AlarmSystem.dm:
  - ***AlarmSystem***: a tab for the main machine, containing the following sub-tabs:
    - ***SBS***: shows the SBS for the machine;
    - ***States***: shows the list of states defined in the machine and facilitates the specification of informal design information about the states;
    - ***State Variables***: shows the list of state variables defined for the machine and facilitates the declaration and specification of state variables.
    - ***State Diagram***: a graphical representation of the states and transitions in the state machine

    Note: In a design model, there may be more machines: one main machine and zero or more sub machines. See "Sub Machines" for details about adding and using sub machines.
  - ***Data Variables***: the list of defined data variables, these are used to remember data across rule cases. See also "Data Variables".
  - ***Service References***: Shows the list of service references. See also "Service References"
  - ***Implemented Service***: shows the interfaces and events of the implemented interface model.
  - ***Used Services***: shows a tab for each used service (service dependency). See "Defining Used Services" for details about used services.
  - ***Type Definitions***: state variable type definitions.
- ***Tags***: shows the list of requirements defined for the component and facilitates the specification of additional requirements that emerge during the design phase. See also "Tags".
- Names and identifiers must adhere to these syntactical rules.
- The ASD:Suite ensures that the set of all triggers in each state of each machine of the design model is consistent with the set of events of the implemented service and used services.
- The following is copied from the interface model into the design model if you have checked the "Copy SBS from interface model" check box:
  - The states of the main machine with all information stored in the interface model (user columns and descriptions).
  - The state variables of the main machine.
  - The tags.
  - The SBS of the main machine, with the exception of the rule cases that have a modelling event as trigger.

Service References

In a design model, you can make use of other components, provided that they have an ASD interface model that specifies their services.

To use a component, you create "Service References". Service references are like reference variables in a programming language: they contain references to *instances* of a used service. In ASD, service references are immutable, i.e. once defined, you cannot change their content. Also service references are always *sequences*. Per service reference, the SBS shows all the triggers and actions of the associated interface model.

Service References do not directly refer to an interface model. Rather, they refer to a Used Service (found under the Used Services tab) and this Used Service refers to the interface model. This setup allows to change settings per used service and have multiple service references use those settings. For instance, which interfaces are used, Observed and Singleton event settings can be done per Used Service. The Service Reference defines the number of instances of a service; the Used Service defines how an interface model is used by the DM.
To allow different uses of an interface model, it is possible to create multiple Used Services that refer to the same interface model.

Defining Service References

You can find the service references of a design model in the Service References tab. **Note** that it is easiest to add a service reference not by typing in the table cells but by right-clicking the table and selecting "New Service Reference". This shows a dialog to help you load an interface model and create references to it.

A service reference is defined by the following:

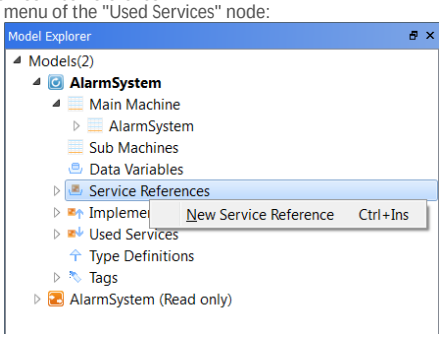
- **Reference Name**: the name of the service reference.
- **Cardinality**: the number of instances in the service reference. This must be either:
  - A fixed number between 1 and 128. In that case, ASD will create the instances for you when the component is constructed.
  - The asterisk symbol (*). In that case, you have to define a construction parameter for the design model and use it in the Construction field (see below). This way, the decision of how many instances to create is left until construction time.
- **Verification Cardinality**: if a number is filled in here, the Cardinality column is ignored in the model verification and this number is used instead. This is required when the Cardinality column contains an asterisk (*). It is optional if the Cardinality column contains a number. It can be used to reduce model verification time.
  Warning / disclaimer: if the number of elements in a reference used for verification differs from the number of elements indicated in the design, it is up the user to prove that the verification results hold for more elements in the reference. The ASD:Suite can only guarantee what has been explicitly verified.
- **Construction**: used in the generated code during construction of the used service reference. This is what you can specify in this cell:
  - A component name - If the used service is an ASD component, this is the name of the design model, if the used component is a foreign component, the actual component name of that foreign component must be used. The component name may include a namespace. The parts of the namespace are separated by dots. E.g.: alarmdemo.Sensor.
    Note that the component name will be postfixed with "Component" in the generated code.
  - A component name followed by a list of arguments for the GetInstance call to the component. The arguments can be literals (between dollars ($)), construction parameters or even already defined service references.
  - A use statement in the form of "use arg" to inject a component using either a service reference or a construction parameter.

  See "Construction Parameters" for details about the concepts of Construction Parameters, and see "Defining construction parameters" for details about defining construction parameters for your ASD component.
- **Service**: the Used Service (specified by an interface model) that the instances of the reference implement. You can add, remove and edit the dependencies of a Design Model to Used Services under the Used Services tab. See also Defining Used Services
- **Comments**: any comment.
- **Tags**: references to Tags.

Note: A service reference can not be empty and is immutable (the contents are defined during construction and are not changed at runtime)

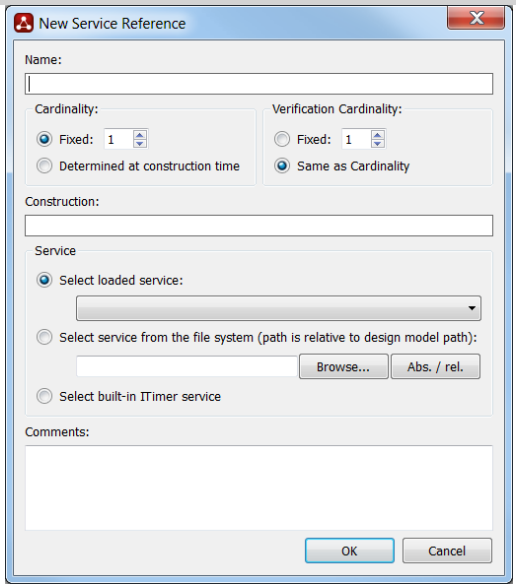
Take the following steps to create a service reference:

- Select the "Service References" node in the "Model Explorer" window and open the context menu by pressing the right button of the mouse. In the context menu, select "New Service Reference".
  The following figure shows the context menu of the "Used Services" node:



The "New Service Reference" context menu item under "Service References"

- Fill in the data in the "New Service Reference" dialog window. The following figure shows an empty "New Service Reference" dialog window:

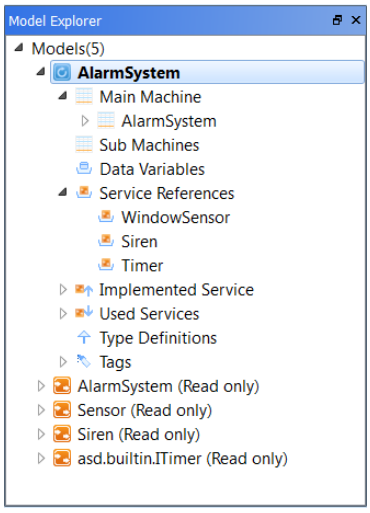


The "New Service Reference" dialog window

Note: The currently loaded interface models, with the exception of the one for the implemented service, are shown in the list of loaded services. If you want to select a service which is not already loaded, you have to set the "Select Service from the file system" radio-button and select the interface model of the respective service after pressing the Browse button. In this case when the service reference is created the service dependencies are also created / updated.

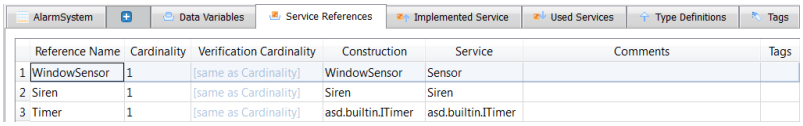
- Repeat the previous steps for all required service instances.

The following figure shows the Service References defined for the Alarm system:



A design model with used services

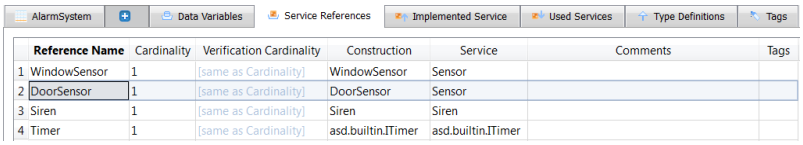
The following figure shows the "Service References" tab after creating the service references:



The "Service References" tab with the specified used services

Specify different service references to the same service

The following figure shows the situation where two service references for service "ISensor" are specified, one named DoorSensor and the other named WindowSensor.



Different components with the same service

Defining Used Services

Used Services define the link between the design model and the used interface model. Also, they define how the design model wishes to make use of the interface model: e.g. which notification events are observed and which interfaces are used or unused. Each used service contains the relative path to the interface model, and a copy of all the interfaces and events of the interface model. The latter allows to independently edit the design model and the interface model without losing data. See also Refactoring ASD Models.

You can add, edit and delete Used Services from the Used Services tab by right-clicking the tabs and selecting New Used Service, Rename, or Delete. Alternatively, you can also add Used Services by simply creating Service References using right-click

and selecting New Service Reference in the Service References table.

In the rare case that your design model needs to make use of an interface model in multiple different ways (having different sets of used interfaces, observed notification events, or singleton notification events), you can add multiple Used Services that refer to the same interface model. To make this easier, you can duplicate an existing used service by right-clicking its tab and selecting Duplicate Used Service.

Usually, when you have only one Used Service per interface model, you will want to keep the name of the used service in sync with the name of the interface model. To do this, right-click the used service and select "Rename like IM". This will find the name in the interface model and rename the used service accordingly.

Marking Interfaces as Used or Not Used

You do not need to use all the interfaces in a used service. E.g. you can share a Singleton component between two clients, where each client uses a different interface of the component.

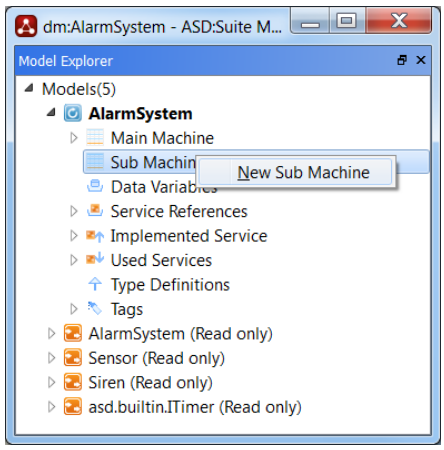
To mark an interface as "used" or "not used", check or uncheck the corresponding checkbox at the top of each of the interface tabs in the design model. You can find these tabs under the service-specific tab in the Used Services tab. Checking or unchecking the box also adds or removes the corresponding rule cases and actions in the SBSes of the design model.

Sub Machines

The ASD:Suite supports hierarchical machines for design models. This is restricted to one level of sub machines. In other words, a sub machine can not have a Super state.

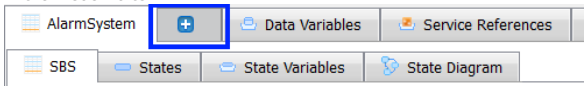
You can add a sub machine in one of the following ways:

- Right-click on the Sub Machines node in the "Model Explorer":



Menu item to create a sub machine

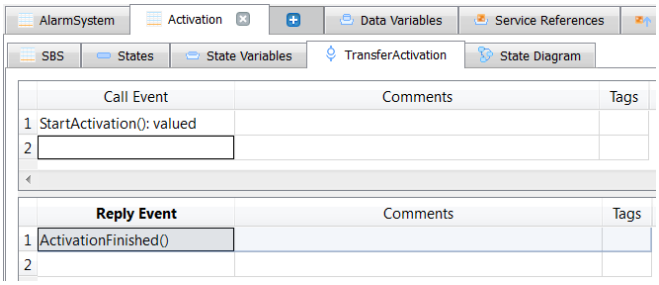
- Click-on/Push the button in the Model Editor:



The button to add a sub machine

- Press the "Ctrl+T" hotkey in the Model Editor of a design model

The new sub machine will be created after you specify a name. Additionally, the corresponding transfer interface is created; this interface is used for the synchronisation between the main machine and the sub machine. This transfer interface inherits its name from the name of the sub machine. For each transfer interface one or more call events must be declared, as well as one or more reply events. The declarations works in the same way as defining call events and reply events for an application interface with the observation that transfer call events are always of valued type and can carry zero or more *in*, *out* and/or *inout* parameters and that transfer reply events can carry zero of more *in* parameters.



Transfer call events and reply events

The specified transfer call events are automatically added to the set of triggers of the created sub machine, and the transfer reply events are added to the set of triggers of the main machine.

In the main machine all newly created transfer reply events will get a default "Blocked" action, since the transfer reply event is only expected in the corresponding Super state.

> **Note**: The border of the cell in which you specify an event is coloured red if the declaration is not syntactically correct. The event is not stored in the model until the declaration is correct.

In the newly created sub machine all triggers except the transfer call events get a "Blocked" action in the initial state of the sub machine.

Then the new sub machine must be correlated to a Super state in the main machine. First, a new state must be created that will become the corresponding Super state.

Then, on the rule case that will define the transition to the newly created state, add a transfer call event corresponding to the sub machine as the last action in the sequence of actions, and select the newly created state in the "Target State" column. Now, the new state has become a Super state. The "Blocked" action is filled in for all triggers in the new Super state with the exception of the transfer reply events. Then fill in the proper action and target state for the transfer reply event(s) that correspond with the sub machine. The Super state is now ready.

Then, the sub machine remains to be completed. This is done in the normal way of defining the SBS, with one addition: after a transfer reply event is used as an action in a sub machine, the target state always must be the initial state of the sub machine. The transfer reply event returns control to the main machine, and renders the sub machine inactive. When the main machine after some time re-activates the sub machine, this will again have to start from the initial state.

State Variables for Used Service References

A state variable of type Used Service reference (in short a USR state variable) is a sequence of used component instances. USR state variables can be used in guards, actions and state variable updates. A common use case is to keep track of the used service references when iterating over them. See "Used Services" for details about used service references, and see "State Variables" for details about state variables.

Operators for Used Service Reference State Variables

Component instances have a specific set of operators, which are shown in the table below.

Operator	Description	Definition
#S	cardinality: number of elements in S (duplicates included)	
<>	empty reference	#<> == 0
S1 == S2	equality operator	S1 == S2 ≡ #S1 == #S2 ∧ ∀(n : 1 ≤ n ≤ #S1 : S1[n] == S2[n])
S1 != S2	inequality operator	S1 != S2 ≡ not(S1 == S2)
S1 = S2	assignment	establishes S1 == S2
S1+S2	concatenation: pasting S2 behind S1	S == S1+S2 ≡ #S == #S1 + #S2 ∧ ∀(n : 1 ≤ n ≤ #S1 : S[n] == S1[n]) ∧ ∀(n : 1 ≤ n ≤ #S2 : S[n+#S1] == S2[n])
head(S1)	head: reference containing the first element of reference S1 if present and otherwise <>	S == head(S1) ≡ #S1 < 1 => S == <> ∧ #S1 ≥ 1 => S == S1[1]
tail(S1)	tail: reference containing all but the first element of S1 (if present)	S == tail(S1) ≡ #S1 < 1 => S == <> ∧ #S1 ≥ 1 => #S == #S1-1 ∧ #S1 ≥ 1 => ∀(n : 1 ≤ n ≤ #S1-1 : S[n] == S1[n+1])
S1[n]	indexing: reference containing the nth element of S1 if present and otherwise <>	S == S1[n] ≡ n < 1 ∨ n > #S1 => S == <> ∧ 1 ≤ n ≤ #S1 => S == S1[n]
S1 in S2	subset: each element in S1 is present in S2	S1 in S2 ≡ ∀(n : 1 ≤ n ≤ #S1 : ∃(m : 1 ≤ m ≤ #S2 : S1[n] == S2[m]))
that	that: a sequence of references to component instances containing a single reference representing the component instance that generated the trigger of the rule case	

Note: the "in" operator deserves some additional explanation. It implements the subset operation and not the subsequence operation. This means that order and multiplicity are not considered in the comparison. The expression "(S1 in S2) and (S2 in S1)" means that S1 and S2 are *set-equal*. For example, for two arbitrary references S1 and S2, the expression "S1+S1+S2" is set-equal to "S2+S1", but "S1+S1+S2 == S2+S1" only holds if S1 equals <>.

Examples

Examples of how to use the operators in the "Guard", "State Variable Updates", and/or "Actions" columns:

Operator	Description	Guard example	Actions example	State Variable Updates example
#	cardinality	i==#S	-	i=#S
<>	empty reference	S==<>	-	S=<>
==	equality operator	S1==S2	-	b=(S1==S2)
!=	inequality operator	S1!=S2	-	b=(S1!=S2)
=	assignment	-	-	S=S1
+	concatenation	S==S1+S2	-	S=S1+S2
head	head	S==head(S1)	head(robots):IRobot.Start()	S=head(S1)
tail	tail	S==tail(S1)	tail(robots):IRobot.Start()	S=tail(S1)
[]	indexing	S==S1[2]	robots[2]:IRobot:Start()	S=S1[2]
in	subset	S1 in S2	-	b=S1 in S2
that	that	S[3]==that	that:IRobot.Start()	S=that

Note: S, S1 and S2 are all used service reference state variables; variable i is an integer state variable; variable b is a boolean state variable.

Here are a few examples of how to use used service reference state variables and their operators in the SBS of a design model:

1. Iterate over a number of sensors to activate them one by one:

	Interface	Event	Guard	Actions	State Variable Updates	Target State	
1	**NotActivated (initial state)**						
4	IAlarmSystem	SwitchOn+		head(activate):ISensor.Activate+	activate = tail(activate); deactivate = <>	Activating	*Activate sensor*
5	IAlarmSystem	SwitchOff		Illegal		-	*Illegal - alarm not activated*
8	Sensors:ISensor_NI	DetectedMovement		Illegal		-	
9	Sensors:ISensor_NI	Deactivated		Illegal		-	
10	Timer:ITimerCB	Timeout		Illegal		-	
11	**Activating (synchronous return state)**						
16	Sensors:ISensor	OK	activate != <>	head(activate):ISensor.Activate+	activate = tail(activate); deactivate = deactivate + that	Activating	
17	Sensors:ISensor	OK	otherwise	IAlarmSystem.Ok		Activated_Idle	
18	Sensors:ISensor	Failed	deactivate != <>	deactivate:ISensor.Deactivate-		Deactivating	
19	Sensors:ISensor	Failed	otherwise	IAlarmSystem.Failed	activate = Sensors	NotActivated	

Iterate over a used service reference state variable member by member

Note:
- "activate" is a used service reference state variable for all the to-be-activated sensors defined for the model. Its initial value is all the sensors, i.e. all door and window sensors: DoorSensor+WindowSensor
- "deactivate" is a used service reference state variable for all the sensors to be deactivated. Its initial value is the empty sequence "<>".

2. Deactivate all sensors and wait for the result of deactivation for each sensor respectively:

	Interface	Event	Guard	Actions	State Variable Updates	Target State	
11	**Activating (synchronous return state)**						
16	Sensors:ISensor	OK	activate != <>	head(activate):ISensor.Activate+	activate = tail(activate); deactivate = deactivate + that	Activating	
17	Sensors:ISensor	OK	otherwise	IAlarmSystem.Ok		Activated_Idle	
18	Sensors:ISensor	Failed	deactivate != <>	deactivate:ISensor.Deactivate		Deactivating	
19	Sensors:ISensor	Failed	otherwise	IAlarmSystem.Failed	activate = Sensors	NotActivated	
23	**Activated_Idle**						
33	**Deactivating**						
36	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm syst*
37	IAlarmSystem	SwitchOff		Illegal		-	*Illegal - alarm syst*
40	Sensors:ISensor_NI	DetectedMovement		NoOp		Deactivating	*Sensors being deac*
41	Sensors:ISensor_NI	Deactivated	deactivate in deactivated + that	IAlarmSystem.Failed	deactivated = <>; activate = Sensors	NotActivated	*Sensor deactivated*
42	Sensors:ISensor_NI	Deactivated	otherwise	NoOp	deactivate = deactivate + that	Deactivating	

Perform an action on all members of a used service reference variable and wait for all of them to respond

Note:

- "deactivate" is a used service reference state variable for all the sensors to be deactivated.
- "deactivated" is a used service reference state variable for all the sensors which are deactivated.
- The "deactivate in deactivated + that" guard evaluates to true if all members of "deactivate" are present in "deactivated + that". "deactivated + that" translates to the "deactivated" sequence together with the current service instance

You can refer to used service references (service references or used service reference state variables in combination with their operators) in several ways when adding actions:

Syntax	Description
myReference:IMyAPI.Start()	call Start on every service instance in myReference
myReference[3]:IMyAPI.Start()	call Start on the third service instance in myReference
that:IMyAPI.Start()	call Start on the service instance that generated the trigger
head(myReference): IMyAPI.Start()	call Start on first service instance in myReference
tail(myReference): IMyAPI.Start()	call Start on every service instance in myReference, except the first one

where myReference denotes a service reference or a used service reference state variable.
Note:

- The used service reference of a valued action must always be a singleton, i.e. a reference of length 1.
- The used service reference of a void action must contain at least one element.

Singleton Notification Events

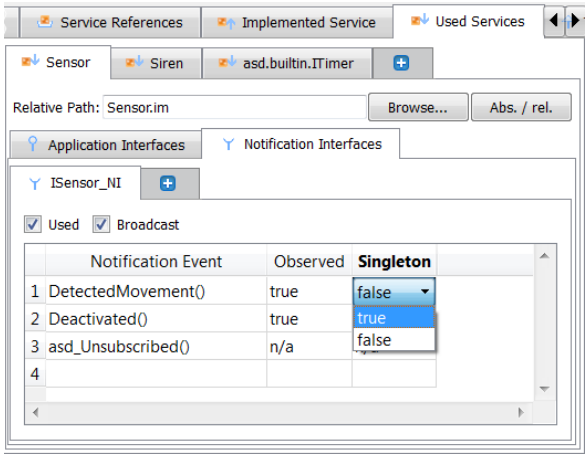
The ASD:Suite enables you to specify that at any given moment in time a specific notification event from a used service instance should occur only once in the queue of your component. This is done by flagging the respective event as a Singleton.

Note:

- Flagging an event as singleton does not mean that the respective event can not occur anymore until taken out of the queue. It only means that when the respective notification event occurs and there is already one in the queue, the latest occurrence will be considered irrelevant and will be discarded.
- Good candidates for singleton notification events are, for example, notifications from a driver reporting new data or progress notifications.

In case you want to, in your design model, flag an event on a used service notification interface as Singleton, you should open the Used Services tab, select the relevant used service, and you should view its events in the "Notification Interfaces" tab.

The following figure shows how to flag the ISensor_NI.DetectedMovement event as a Singleton:



Flag an event as Singleton

ASD guarantees that per instance of a used service notification interface there is at most one notification event in the queue for the events flagged as Singleton. For example, if a notification event "A"of interface "IPublisher"is flagged as "Singleton", then:

1. If one instance of IPublisher is used, at most one "A" is in the queue at any given time;
2. If there are N instances of IPublisher, at most N "A"s are in the queue at any given time; This is independent of whether the underlying Publisher component is a Singleton Component or Multiple Component.

Behaviour in an ASD model

In order to specify behaviour in an ASD model you have to fill-in **each line** in the SBS tab, also known as rule cases, with guards, actions, state variable updates, target state, comments, and tags. By this you will define behaviour for each event which can trigger an action, or more actions, in various states of your system. These events are called triggers.

Note:

- In order to specify guards and state variable updates you need to define state variables.
- For each state you are defining you can specify additional (design) information.

The ModelBuilder ensures that the following events are present in each state specified in the SBS tab of the main machine of an interface model as triggers:

- All application call events, and
- All modelling events

The ModelBuilder ensures that the following events are present in each state specified in the SBS tab of the main machine of the design model as triggers:

- All implemented service application call events.
- All used service notification events for all the used service notification interfaces that are observed.
- All used service application reply events for all the used services application interfaces specified as used interfaces.
- All transfer reply events for all transfer interfaces defined in the design model.

The ModelBuilder ensures that the following events are present in each state specified in the SBS tab of a sub machine of the design model as triggers:

- All implemented service application call events.
- All used service notification events for all the used service notification interfaces that are observed.
- All used service application reply events for all the used services application interfaces specified as used interfaces.
- All transfer call events for the transfer interface of the sub machine

Note:

- See "Specifying used services" for details about used services.
- See "Sub machines" for details about using sub machines.

State Variables

State variables provide means to capture fine-grained history as opposed to states which capture the more coarse-grained history. In the SBS, *guards* are used to base control-flow decisions on state variables and *state variable updates* are used to change the values of state variables.

> **Note:** Because ASD has strict separation of control and data, state variables can not be used as arguments of events and event arguments can not be used in a guard or in a state variable update expression.

State variables are declared in the "State Variables" tab. Each state machine has its own set of state variables, and hence also its own State Variables tab.

SBS | States | State Variables | State Diagram

	State Variable	Type	Constraint	Cardinality	Initial Value	Comments	Tags
1	isChecked	Boolean	n/a	n/a	false	*Boolean state variable for keeping track of whether the alarm has been checked or not.*	
2	Counter	Integer	[0:5]	n/a	0		

State variable specification for the "IAlarm" machine

The State Variables table has the following columns:

- **Name**: a unique name for the state variable. Note that state variable names may not be identical to enumeration values, states, or - in case of a USR type state variable - service references.
- **Type**: the data type of the state variable. This can be a built-in type, or refer to a type definition. This is explained in more detail below.
- **Constraint**: a restriction on the values of the state variable. For instance, for integer variables the *range* can be defined. This is explained in more detail below.
- **Cardinality**: only for Used Service Reference type state variables. It specifies the maximum number of elements that the state variable can hold.
- **Initial Value**: the value that the state variable gets at component construction.
- **Comments**: any comment.

Boolean State Variables

Boolean state variables can have values *true* or *false*. To create a Boolean state variable, select "Boolean" in the Type field. Boolean variables cannot have a constraint or a cardinality. You must supply an initial value - either *true* or *false*.

Integer State Variables

Integer state variables have the natural numbers -32767...+32767 as values. However, using state variables with such a broad range increases the state space and thus model verification time. For this reason, integer variables can have a *range* specified in the Constraint field. To create an integer state variable, select "Integer" in the Type field. Then, define a range for the state variable in the Constraint field, e.g. "[0:3]" for values 0,1,2 and 3 (note that ranges are *inclusive* at both ends). Supply an initial value (which must be within the given range) in the Initial value field. Integer variables cannot have a Cardinality.

It is possible to define a range separately and to use it for multiple state variables: you can create a type definition. First define the type definition (see "Type Definitions") and then select it in the Type field. You cannot specify a constraint anymore, since that is already done in the type definition.

Tip: the model verifier checks that the value of an integer state variable stays within its given range. Make the range as narrow as possible to reduce model verification time.

Enumeration State Variables

Enumeration state variables have a given user-defined set of identifiers as possible values. To create an enumeration state variable, you must first define an enumeration *type*. See the section on "Type Definitions" for how to do this. Once you have an enumeration type, you can select it in the Type field. Use one of your defined enumeration values in the Initial Value field. Enumeration state variables cannot have a constraint or a cardinality.

Used Service Reference State Variables

USR state variables are mutable sequences of component instances. In other words: they can contain elements of Used Service References. You can only declare USR state variables in design models (as interface models do not have used service references). They are useful e.g. for keeping track of which instances you have received notification events from.

You can declare a USR state variable by selecting "Used Service Reference" in the Type field. Select a Used Service in the constraint field - the state variable will only store instances of components implementing this used service. In the Cardinality column, you specify the *maximum* number of instances that will be stored in the variable. The model verifier will check that this maximum is never exceeded. The initial value of a used service reference state variable is a string that can be constructed from the names of service references and the operators described in "State Variables for Used Service References".

Using State Variables in the SBS

See "Specifying guards" for details about the usage of state variables in guards, and see "State Variable Updates" for details about using state variables in state variable updates.

Type Definition

You can create re-usable state variable types in the Type Definitions tab of the Model Editor. When you create such a type definition, it becomes available for selection in the Type column of the State Variables tables.

The Type Definitions table has the following columns:

- **Name**: a unique name for the type definition.
- **Type**: the type to base the type definition on. This can be either Integer or Enumeration.
- **Constraint**: a restriction on the type. For instance, for integer variables the *range* can be defined. This is explained in more detail below.
- **Comments**: any comment.

Integer Type Definition

Integer types have the natural numbers -32767...+32767 as values. However, using state variables with such a broad range increases the state space and thus model verification time. For this reason, integer types can have a *range* specified in the Constraint field. An example range is "[0:3]" for values 0,1,2 and 3 (note that ranges are *inclusive* at both ends). The model verifier ensures that all variables of the type stay within its range. This reduces the model verification time.

Tip: you do not need to type the square brackets around the range. The ModelBuilder will automatically add missing brackets.

Enumeration Type Definition

Enumeration types have a given user-defined set of identifiers as possible values. To define an enumeration type, select Enumeration in the Type column and specify the enumeration values in the Constraint column. Separate values by commas.

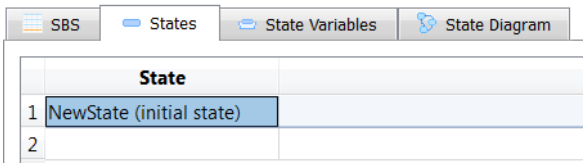
Note: enumeration value names must be unique with respect to all other enumeration values in the model, to states, state variables, and service references.

Tip: the ModelBuilder automatically puts each enumeration value on a separate line. This allows you to put a line in the Comments field for every enum value.

State Information

Since at the creation of the interface model an initial state was created and automatically named "NewState", you may want to change this name and provide a description for the respective state.

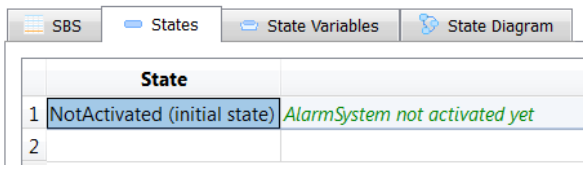
The following figure shows the "States" tab, which is used to create new states or rename existing states and provide a description for them:



The ASD:Suite - the "States" tab

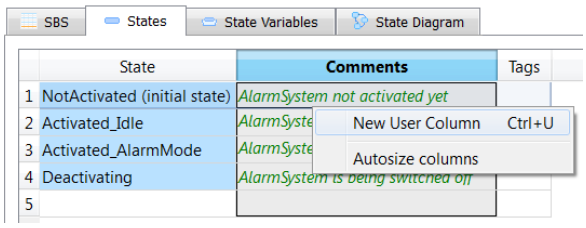
To rename an existing state and provide a description for it, type a different name in the "State" column and enter a description in the "Comments" column.

The following figure shows a filled-in "States" tab for machine "IAlarm":



State specification for machine "IAlarm"

The ASD:Suite provides the possibility to add design information to a state description. This is achieved via adding so called user columns next to the description. You have to select the "New User Column" context menu item obtained by right-clicking with the mouse on one cell of the state declaration (see next figure) or you have to press "Ctrl+U".



The context menu item to add a new user column

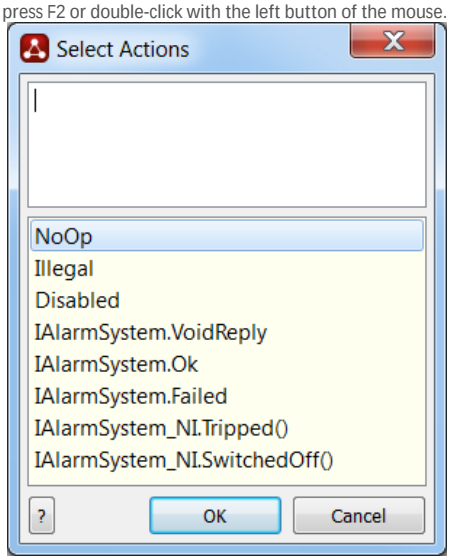
The following operations can be used to edit a user column:

- **New User Column**: add a new user column
- **Delete User Column**: remove the selected user column
- **Rename User Column**: change the name of the selected user column
- **Autosize columns**: set the size of the columns to fit the size of the text in the cells and the column titles

Actions

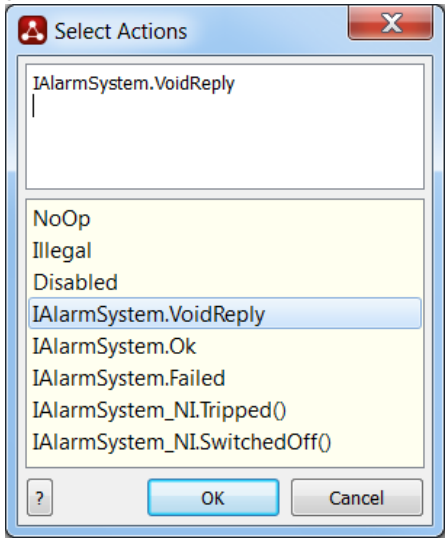
These are the steps to insert an action in a rule case:

1. Select the cell in the Actions column and press F2 or double-click with the left button of the mouse. The "Select Actions" dialog appears:



The ASD:Suite-the "Select Actions" dialog

2. Select the action from the list that appears:

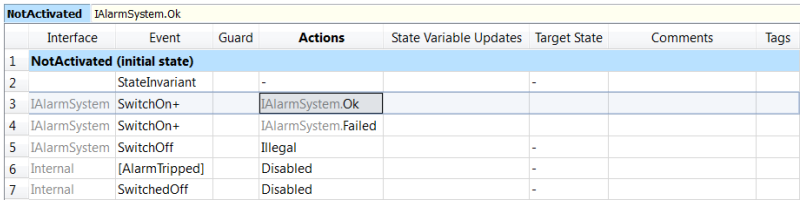


Selecting an event to be added as action

3. Repeat the previous step with other actions from the list if you wish to add more actions
   Note:
   - The "Select Actions" dialog is split in two panes: one text editor and a list.
   - Press the Tab key or Select the panes with the mouse to switch between the panes.
   - The text editor enables you to type the name of the actions and/or to set the order of actions.
   - While typing, the actions in the list are filtered out using sub-string matching easing up your job to specify the desired action.
   - Press Shift+Enter or Alt+Enter in the text editor to insert a blank line between two already specified actions.
   - Pressing Tab in the text editor while you specify an action performs prefix completion for the respective action, i.e. it extends the currently specified name to the longest common prefix within the existing actions which matches the currently specified text.
   - Cut-Copy-Paste-Delete operations are enabled in the text editor part in the same way as in any text editor.
   - Use Ctrl+Up or Ctrl+Down to move a selected action up/down in the list

4. When all the actions are specified, click "OK" or switch to the text editor of the "Select Actions" dialog and press Enter to add the actions in the SBS.
   Note: If there are wrongly specified actions, those will be underlined, and you cannot click "OK" until the actions are syntactically correct.

The following figure shows the SBS tab after adding actions.



NotActivated IAlarmSystem.Ok

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	**NotActivated (initial state)**							
2		StateInvariant		-		-		
3	IAlarmSystem	SwitchOn+		IAlarmSystem.Ok				
4	IAlarmSystem	SwitchOn+		IAlarmSystem.Failed				
5	IAlarmSystem	SwitchOff		Illegal		-		
6	Internal	[AlarmTripped]		Disabled		-		
7	Internal	SwitchedOff		Disabled		-		

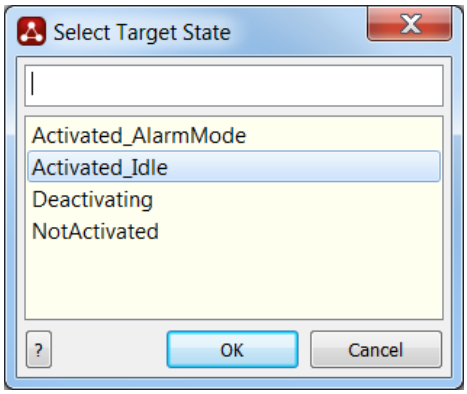
The SBS tab after specifying actions

Note: You can select multiple cells in the Actions column to insert the same action(s) into them. The cells do not have to belong to consecutive rule cases.

Target State

These are the steps to specify a target state in a rule case:

1. Select the cell in the "Target State" column and press F2 or double-click with the left button of the mouse. The "Select Target State" dialog appears:



The ASD:Suite-the "Select Target State" dialog

2. Select the state from the list
   Note:
   - The "Select Target State" dialog is split in two panes: one text editor and a list.
   - You can switch between the panes of the "Select Target State" dialog by pressing the Tab key or by selecting the panes with the mouse.
   - The text editor enables you to type the name of the desired state or to create a new state.
   - While you specify the state manually the states in the list are filtered out using sub-string matching easing up your job to specify the desired state.
   - Pressing Tab in the text editor while you type a state name performs prefix completion for the respective state name, i.e. it extends the specified name to the longest common prefix within the existing state names which matches the currently specified text.
   - To create a new state and to specify the respective state as the target state you have to specify a non existing valid name for the respective state.

     Note: If the desired state name is part of an already existing state name, the sub-string matching will select one of the existing states containing the respective string and you might end up in selecting the respective state instead of creating a new one. To avoid this, we suggest you add an extra space at the end of the desired name which will disable sub-string matching and press Enter to create the new state. The space at the end of the name will be automatically removed since no spaces are allowed in specified names.

The following figure shows the situation when the specified target state is a new state.

NotActivated Activated_Idle

	Interface	Event	Guard	Actions	State Variable Updates	**Target State**	Comments	Tags
1	**NotActivated (initial state)**							
2		StateInvariant		-		-		
3	IAlarmSystem	SwitchOn+		IAlarmSystem.Ok		Activated_Idle	*Succesfully activated alarm system*	
4	IAlarmSystem	SwitchOn+		IAlarmSystem.Failed				
5	IAlarmSystem	SwitchOff		Illegal		-		
6	Internal	[AlarmTripped]		Disabled		-		
7	Internal	SwitchedOff		Disabled		-		
8	**Activated_Idle**							
9		StateInvariant		-		-		
10	IAlarmSystem	SwitchOn+						
11	IAlarmSystem	SwitchOff						
12	Internal	[AlarmTripped]						
13	Internal	SwitchedOff						

The SBS tab after target state specification

Note: The ASD:Suite ensures that the proper triggers are present in each state of an SBS.

It is possible to specify the current state as a target state, i.e. to create a self transition, without using the "Select Target State" dialog. These are the alternatives:

- Select the "Self Transition" item in the context menu obtained by clicking the right mouse button while selecting the cell of the rule case situated in the "Target State" column, or
- Press "Ctrl+Space" when the cell in the "Target State" column is selected.

Comments

You may specify a description for each rule case. In order to do so, select the field in the "Comments" column on the line reflecting the rule case, press F2 or double-click with the left button of the mouse and type the desired text.

The following figure shows the SBS tab after some comments have been entered for the rule cases.

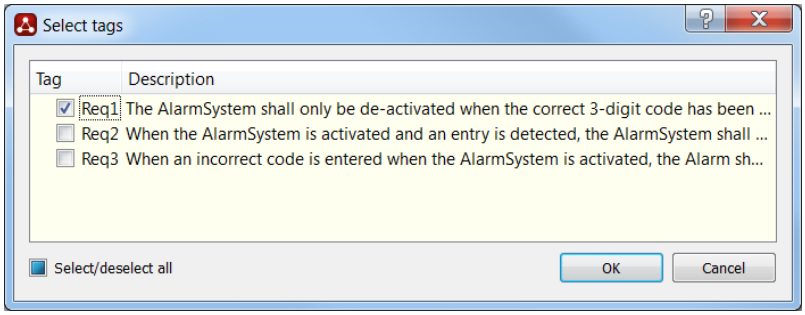
NotActivated Succesfully activated alarm system

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	**NotActivated (initial state)**							
3	IAlarmSystem	SwitchOn+		IAlarmSystem.Ok		Activated_Idle	*Succesfully activated alarm system*	
4	IAlarmSystem	SwitchOn+		IAlarmSystem.Failed		NotActivated	*Illegal - alarm not activated yet*	
5	IAlarmSystem	SwitchOff		Illegal		-	*Illegal - alarm not activated yet*	
6	Internal	[AlarmTripped]		Disabled		-		
7	Internal	SwitchedOff		Disabled		-		

The SBS tab after filling in rule case comments

Specifying tags

To attach tags (which you can define in the Tags tab of your model) to a rule case, click the cell in the Tags column. A dialog is shown containing all your defined tags. Select the tags you want to have and click Ok.



Tag Selection Dialog

Guards

Guards are Boolean expressions that are used to make fine-grained control decisions in an SBS. Specifying guards is done by filling in the "Guard" column of the SBS tab. A rule case can only be executed when its guard evaluates to *true*. You can insert multiple rule cases for the same trigger, and use guards to choose between them.

> **Note:** if you leave the cell in the "Guard" column empty, it will be interpreted as "not guarded", i.e. it is considered to be equivalent to *true*.

Guard Expression Syntax

A guard is a boolean expression, built out of state variables, literal values, and the following operators:

- **Arithmetic**: "+", "-"
- **Comparison**: "==", "!=", "<", ">", "<=", ">="
- **Logical**: "and", "or", "not"

For Design Models, additional operators are available for USR State Variables. See "State Variables for Used Service References".

Otherwise Keyword

Often, a guard for one rule case will be the opposite of the guard of the other rule cases in the rule. For instance, if a rule has two rule cases, one could have a guard "a == false" and the other a guard "a == true". An easier way to specify this is to replace one of the two expressions by the keyword "otherwise", meaning "none of the other guards are true".

There are some rules concerning the use of "otherwise":

- It can not be part of a larger expression; it is a complete term on its own.
- It can not be specified if the rule has only one rule case
- In an interface model, more than one rule case in a given rule can have "otherwise" as the guard, indicating a nondeterministic choice between them. However there must be at least one rule case for that rule having a guard other than "otherwise"
- In a design model, only one rule case in a given rule can be guarded by "otherwise".

Example

The figure below shows guards in the case of the Alarm system example.

- If the alarm has not yet been checked, i.e. "isChecked==false", when triggered by "Check" one of the two actions might occur: CheckSuccessful or CheckFailed.
- If the Alarm has already been tested, i.e. "isChecked==true", acting to a "Check" trigger is illegal since the Alarm should not be checked again.

Activated_Idle isChecked == false

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
9	**Activated_Idle**							
11	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm already activated*	
12	IAlarmSystem	SwitchOff		IAlarmSystem.VoidReply		Deactivating	*Busy deactivating*	
13	IAlarmSystem	Check+	isChecked == false	IAlarmSystem.CheckSuccessful	isChecked = true	Activated_Idle	*Alarm checked successfully*	
14	IAlarmSystem	Check+	isChecked == false	IAlarmSystem.CheckFailed	isChecked = true	Activated_Idle	*Failure while checking the alarm*	
15	IAlarmSystem	Check+	otherwise					
16	Internal	[AlarmTripped]						

An SBS with guards

State Variable Updates

When you have defined state variables for an SBS, you will want to modify their value in the SBS. You do this by putting *state variable update expressions* into the State Variable Updates column.

A state variable update expression is either:

- An assignment statement of the form "statevariable = expression"
- An increment statement of the form "sv++" (meaning "sv = sv + 1")
- A decrement statement of the form "sv--" (meaning "sv = sv - 1")

For instance, if you have an integer state variable called "a", you can write "a = 5". You can update multiple variables at once by writing multiple assignments, with colons (;) to separate them, e.g. "a = 5; b = 6" or "a = 5; b++".

> **Note:** the assignments within a State Variable Updates cell occur *simultaneously*. For example, if the current value of a is 5, and you enter the state variable update expressions "a=1; b=a", then the effect of executing the state variable update is: a=1 and b=5.

The following figure shows an SBS with some state variable updates:

Activated_Idle isChecked = true

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
9	**Activated_Idle**							
11	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm already activated*	
12	IAlarmSystem	SwitchOff		IAlarmSystem.VoidReply		Deactivating	*Busy deactivating*	
13	IAlarmSystem	Check+	isChecked == false	IAlarmSystem.CheckSuccessful	isChecked = true	Activated_Idle	*Alarm checked successfully*	
14	IAlarmSystem	Check+	isChecked == false	IAlarmSystem.CheckFailed	isChecked = true	Activated_Idle	*Failure while checking the alarm*	
15	IAlarmSystem	Check+	otherwise					
16	Internal	[AlarmTripped]						

SBS with state variable updates

Expression Syntax

On the right-hand-side of a state variable update expression, you can build an expression out of the state variables, literal values, and the following operators:

- **Arithmetic**: "+", "-"
- **Comparison**: "==", "!=", "<", ">", "<=", ">="
- **Logical**: "and", "or", "not"

For Design Models, additional operators are available for USR state variables and component instances. See "State Variables for Used Service References".

Invariants

In each state of your SBS, you can specify a *state invariant* and, for design models, a *data invariant*. Each invariant is an assumption you make about the value of variables in the given state. This assumption is then checked by the model verification. In the state invariant, you give your assumptions about state variables. In the data invariant, you can give your assumptions about data variables.

Note: the StateInvariant and DataInvariant rule cases can be hidden or made visible using the "Filters->Hide Invariants" menu item - make sure it is not currently hidden.

State Invariants

A state invariant is a Boolean expression on state variables that should be *true* whenever your component enters the state. The model verifier checks that the state invariant always holds. You specify these invariants in the Guard cell of the special StateInvariant rule case that is present at the top of each state.

Expression Syntax

State invariants are Boolean expressions, just like guards. See "Guards" for an explanation.

Example

In the example below, "AlarmOn" is a Boolean state variable. In the AlarmNotActivated state, the invariant asserts that the state variable is *false*. If there is any possibility for the component to enter the AlarmNotActivated state while the variable is *true*, the model verifier will notify you with a model verification error.

	Interface	Event	**Guard**	Actions	State Variable Updates	Target State
1	**NotActivated (initial state)**					
2		StateInvariant	not AlarmOn	-		-
3		DataInvariant		-		-
4	IAlarmSystem	SwitchOn+		WindowSensor:ISensor.Activate; IAlarmSystem.Ok		Activated_Idle

Example of a StateInvariant rule case

Data Invariants

Like state invariants, data invariants are used to check an assumption about the value of datavariables upon entry of a state. In this case, it is an assumption about your data variables being valid or invalid. As ASD has no view on data other than data flow through your model, the only thing ASD can know is whether the variables are valid or not, and it cannot know about the actual content of the variables.

Note that data invariants are only available in design models, as interface models do not have data variables. The model verifier checks the validity of data invariants in the Data Variable Check.

Expression Syntax

Data Invariants are lists of statements of the form "datavariable.isValid()" or "datavariable.isInvalid()". The statements should be separated by semi-colons.

	Interface	Event	**Guard**	Actions	State Variable Updates	Target State
1	**NotActivated (initial state)**					
2		StateInvariant	not AlarmOn	-		-
3		DataInvariant	dvTimeOut.isInvalid()	-		-
4	IAlarmSystem	SwitchOn+		WindowSensor:ISensor.Activate; IAlarmSystem.Ok		Activated_Idle

Example of a DataInvariant rule case

Non-deterministic Behaviour

An interface model may describe non-deterministic behaviour. That is, a given trigger may result in different actions. The choice of which action will be executed depends on conditions within the component that are not visible on the interface level.

Note: Non-deterministic behaviour can be specified only in case of an interface model, it can not be specified for design models. The rationale behind it is that an interface model specifies the behaviour of the component at an abstract level. At this level the non-deterministic behaviour abstracts out the implementation details and only specifies that in response to a trigger, the component can react in alternative ways. Verification using the ASD:Suite guarantees that no non-deterministic behaviour is specified in a design.

To specify non-deterministic behaviour in an interface model, you must specify more than one rule case with non-exclusive guards for a trigger. See "Adding or deleting a rule case" for details.

Adding or deleting a rule case

This is how to add a new rule case:

- Below the selected rule case: Select a rule case in the SBS tab and press "Ctrl+Insert" or select the "New Rule Case Below" item in the context menu obtained when clicking on the right mouse button.
- Above the selected rule case: Select a rule case in the SBS tab and press "Ctrl+Alt+Insert" or select the "New Rule Case Above" item in the context menu obtained when clicking on the right mouse button.

Note: The selected rule case is going to be replicated (only the data in the "Interface" and "Event" columns).

This is how to delete a rule case:

- Select a line in the SBS tab and press "Ctrl+Delete" or select the "Delete Rule Case" item in the context menu obtained when clicking on the right mouse button.

Note: You will not be able to delete the one and only rule case for a rule since there must always be at least one rule case for each trigger.

Inserting or replacing rule cases

When you want to insert one or more filled-in rule cases or you want to replace a set of rule cases you have to first select the "to-be-copied" rule cases and press Ctrl+C. Alternatively you can select the "Copy Rule Case(s)" item in the context menu obtained from (one of) the selected rule case number. Depending on what you want to do next you have to press Ctrl+V after selecting cells, rule cases or a state line.

When you want to insert the rule cases:

- Above the rule cases having the same trigger(s) as the to-be-copied rulecases: Select a cell in a rule case with the same trigger as (one of) the to-be-copied rule case(s) and press Ctrl+Alt+V or select the "Insert Rule Case(s) Above" menu item in the context menu.
- Below the rule cases having the same trigger(s) as the to-be-copied rulecases: Select a cell in a rule case with the same trigger as (one of) the to-be-copied rule case(s) and press Ctrl+V or select the "Insert Rule Case(s) Below" menu item in the context menu.

When you want to replace one or more rule cases with the same trigger, but not all of them: Select the respective rule cases and press Ctrl+V. Only the selected rule case(s) will be replaced.

When you want to replace all rule cases having the same triggers as the triggers in the to be copied set of rule cases: Select the state line, usually blue or orange, of the target state and press Ctrl+V or select the "Paste Rule(s)" item in the context menu of the state.

> **Note:** When you copy whole rule cases defining self-transitions in the source state, they will define self-transitions in the target state too.

Parameters

You can declare parameters for the events in your models to pass data around. The data is not used in the model verification: ASD allows data to flow "transparently" through ASD components. The data can be anything your programming language allows, although there are a few requirements on the data types used.

Parameters vs Arguments

In most programming languages, you first declare a function or method with its *parameters*, and then you use the function or method, passing it *arguments* to the parameters. In this manual, this is how we will use the terms "parameters" and "arguments". Another term for parameters would be "formal parameters" and another term for arguments would be "actual parameters".

- You can define parameters for an event in an interface model, together with a direction and type. This is described in "Parameter Declaration".
- In design models, you supply arguments to the parameters. To pass data from one component to another, you can use so-called "rule case-local variables" and "data variables", or specify *literal values*. See "Parameter Usage".

Parameter Declaration

Parameters are declared in the interface models, in the event tables. Also, you can declare them in the transfer interface of a sub machine of a design model. After the event name, you specify one or more parameters between round brackets "(" and ")", using commas to separate them.

Each parameter declaration consists of three things:

- The direction: [in], [out] or [inout]
- A name: a unique name for the parameter
- A type: a data type in the programming language that you generate code for, e.g. "std::string" for a C++ string.

Here is an example parameter declaration: `[in]numberOfOranges:int`. In this declaration, [in] is the direction, "numberOfOranges" is the parameter name, and "int" is the parameter type.

Here is an example event with multiple declarations: `SupplyFruit([in]numberOfOranges:int, [in] numberOfApples:int):void`.

Parameter Direction

The direction of the parameter defines which way the data flows.

- [in] means that the receiver of the event also receives the data.
- [out] means that the sender of the event receives the data.
- [inout] means that the data goes both ways.

Not every direction is applicable to every type of event:

- **Call Events** support [in], [out] and [inout] parameters.
- **Reply Events** do not support parameters.
- **Tranfer Call Events** support [in], [out] and [inout] parameters.
- **Transfer Reply Events** support [in] parameters.
- **Notification Events** support [in] parameters.
- **Modelling Events** do not support parameters.

Parameter Name

The parameter name must be a valid identifier in the language you generate code for. Also, it must be a valid ASD identifier. See "Syntactical rules for names used in ASD modelling" for details.

Parameter Type

The parameter type can be anything your programming language allows. The type must be copyable/cloneable though.

Note:
In C++, you can use `const` and & around your types, but this is not necessary. If you just fill in the plain datatype, the code generator will add the usual `const` and & where applicable.

Note:
For C++, template types are supported.

Parameter Usage

Parameters that you declared in an interface model (or on a sub machine transfer interface), can be used in a design model.

As ASD is transparent to data, the data is only used in the Event and the Actions columns of the SBS. This allows passing the data from one component to the other, until eventually it ends up in foreign (hand-written) components where it can be actually used. In other words, you cannot use parameters to make choices in guards and state updates. If you want to make a choice based on a parameter value, you can write a foreign component with a function that translates the parameter value into a Reply Event.

Initialisation of arguments

Depending on the direction of the argument, which can be [in], [out], or [inout], and where the argument is used, which can be a trigger or action, arguments may need to be initialised (i.e. given a valid value). It is no longer possible to have arguments not properly initialised leading to potential use of invalid data.

The [in] or [inout] argument of trigger is assumed to be properly initialised by the **calling** component. The [out] argument of a trigger needs to be initialised by:

- Using an argument that was an [out] parameter on one of the actions of the respective rule-case.
- Using a literal value.
- Retrieving a value from a data variable.

The [out] argument of an action is assumed to be properly initialised by the **called** component. The [in] or [inout] argument of an action needs to be initialised by:

- Using an argument that was an [in] or [inout] parameter of the trigger of the respective rule-case.
- Using an argument that was an [out] or [inout] parameter on one of the previous actions of the respective rule-case.
- Using a literal value (for [in] parameters only).
- Retrieving a value from a data variable.

Passing parameter values within a rule case

To pass a parameter value between a trigger and an action, or between actions, you use a rule case-local variable. You do this by simply specifying the same name both for the trigger argument and the action argument. Rule case-local variables do not need to be declared.

For example:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
11	**Activated_Idle**					
15	IAlarmSystem	SwitchOff(secretCode)		Evaluate:IEval.EvaluateCode(secretCode)+		EvaluatingCode
16	Sensor:ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Siren:ISiren.TurnOn		Activated_Tripped

Simple parameter passing example

Passing parameter values across rule cases

To remember a value in the ASD component, so that it can be used later on for a trigger or action on a different rule case, you can use a data variable. Data variables are declared in the "Data Variables" tab. For more info, see Data Variables. When using a data variable in arguments, you need to prefix its name with a so-called *storage specifier*. The storage specifier is either "<<", ">>", or "><". If you use the same name in this way in two rule cases, the value is remembered from one trigger to another and used.

The storage specifiers have a distinct meaning. Suppose the data variable is called "myVariable":

- **<<myVariable** means that a value is taken out of myVariable (retrieved). This is used with [out] parameters on triggers, or [in] parameters in actions.
- **>>myVariable** means that a value is put into myVariable (stored). This is used with [in] parameters on triggers, or [out] parameters in actions.
- **><myVariable** means that the value is both stored an retrieved. This is used with [inout] parameters.

As you can see, you can use only one type of storage specifier in each situation. The storage specifier exists merely to indicate that the variable is a data variable and not a rule case-local one. It also shows the direction of the data flow, which can make the SBS easier to read.

Data variables are local to a component but shared between all sub machines of the component. If the component is thought of as a single class, including all its sub machines, then the data variables are like private data members of the class.

There are two main use cases for data variables:

1. To pass a parameter value across a rule-case, but within the action sequence, in a single thread.
   This happens when the parameter needs to be passed on later in the action sequence, in a synchronous return state or a super state or the initial state of a sub machine. After the thread returns the parameter value loses its relevance. For example:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Idle (initial state)**					
4	CoffeeMakerAPI	MakeCoffee(>>settings)+		CoffeeGrinder:GrinderAPI.Grind+		Grinding
7	**Grinding (synchronous return state)**					
11	CoffeeGrinder:GrinderAPI	GrindOK		CoffeeCreamer:CreamAPI.AddMilkAndSugar(<<settings); CoffeeMakerAPI.CoffeeOK		Idle
12	CoffeeGrinder:GrinderAPI	GrindFAILED		CoffeeMakerAPI.CoffeeFAILED		Idle

   Here the milk and suger preferences are temporarily stored in the data variable *settings* at line 3. After the grinding has finished successfully, milk and sugar must be added, therefore the value of the *settings* data variable is read at line 9. After the coffee is made the *settings* lose their relevance, as the next time someone wants to have coffee, new preferences will be entered.

2. To pass a parameter value across threads.
   This happens when a parameter value is to be remembered for later use, on a different thread. For example:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**NotActivated (initial state)**					
3	IAlarmSystem	SwitchOn(>>TimeOutPeriod)+		WindowSensor:ISensor.Activate; IAlarmSystem.Ok		Activated_Idle
8	**Activated_Idle**					
11	IAlarmSystem	SwitchOff		WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating
12	WindowSensor:ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Timer:ITimer.CreateTimer(<<TimeOutPeriod)		Activated_Tripped

Here the timerperiod for the alarm is entered when the alarm is switched on. This timerperiod is stored in the data variable *TimeOutValue* (on line 3), and is only used when the sensor is triggered, and the timer is started (line 12). In this example the latter occurs on a notification, on a different thread.

Data variables must not be confused with state variables. The state variables are part of the SBS state and as such there is no sharing between sub machines. Also, data variables can not be used in guards and state variable updates.

Literal values

In addition to passing values from one component to another, you can also supply literal values to events. Literal values are specified between dollar signs ($) and can be anything that your programming language allows. If you need to specify a dollar sign within the literal, use two dollars instead ($$) to distinguish it from your closing $ sign.

You can use a literal value for an [out] parameter in a trigger, or an [in] parameter in an action.

For example:

SBS | States | State Variables | State Diagram

Activated_Idle Timer:ITimer.CreateTimer(5); IAlarmSystem_NI.Tripped

	Interface	Event	Guard	**Actions**	Variable Up	Target State	Comments
8	**Activated_Idle**						
12	WindowSensor:ISensor_NI	DetectedMovement		Timer:ITimer.CreateTimer(5); IAlarmSystem_NI.Tripped		Activated_Tripped	Sensor dectected movement - start timer

Parameter passing example with a literal value

Note:
You may not use literals to execute code that changes the state of the component, as this will invalidate the model verification.

Run-time execution

At run-time, the [in] and [inout] arguments of a trigger are copied to the rule case-local and data variables when the trigger event is executed. The arguments of actions are copied when the action is executed.

On the other hand, the [out] and [inout] parameters of a trigger event are assigned a value depending on what type of argument is used: if the argument is a data variable, the value is assigned at the moment that that the corresponding Reply Event is executed. If it is a rule case-local variable, the value last assigned to the variable is used.

Data Variables

Data variables are used to remember argument values across rule cases (see also Parameter Usage). Data variables can only be used in design models. They are declared in the "Data Variables" tab of the model editor, and they are used in triggers and actions in the SBS.

The rule checker and the model verification together can help you to make sure that data variables are valid before they are used. A data variable is considered to be valid when an argument value is stored into it. It is possible to explicitly initialise or invalidate a data variable to take full advantage of the verification and rule checks, e.g. to make sure that no "stale" data is accidentally re-used between calls. Next to that, you can have ASD automatically initialise variables at component construction or invalidate them at the end of each action sequence. This supports some frequently-occuring use cases. This is explained in more detail below.

Note: initialisation and invalidation of data variables is for verification purposes only. They have no effect in the generated code at runtime. The purpose of the data variable verification is to check that data variables are never read when they do not have a proper value. The initialisation and invalidation are merely an aid for the verification.

Data Variable Declaration

You can create data variables in the "Data Variables" tab of the model editor. The table has the following columns:

- **Data Variable**: a unique name for the data variable. This name may not be also used for a rule case-local variable.
- **Auto-Initialise**: indicates whether and when the variable is automatically initialised, and thus made valid. This field can have the following values:
  - **No**: the variable is considered to be invalid at component construction time.
  - **Yes**: the variable is initialised at component construction time.
- **Auto-Invalidate**: indicates whether and when variable is automatically invalidated. This field can have the following values:
  - **No**: the variable is never automatically invalidated. Use the Invalidate() action in the SBS to invalidate the variable.
  - **Yes**: the variable is invalidated at the end of every action sequence.
- **Data Type**: optionally, the data type of your variable. This can be any type of your target programming language (e.g. "std::string" for a C++ string) - the same as for event parameters. You must use a dollar sign ($) at the beginning and end of your type, because the type is language-specific. However, the ModelBuilder will automatically insert them for you if you leave them out. Any dollar within your type must be escaped by using two dollars ($$). Note that this field is *optional* - if you do not fill in a type then the code generator will try to deduce it from the SBS. The type must be default-constructible.
- **Comments**: any comment.
- **Tags**: any tags.

Explicit Initialisation and Invalidation

Within the SBS, you can use the built-in Initialise() and Invalidate() actions to explicitly make a data variable (in)valid. Both events have an [inout] parameter to which you can pass any data variable as argument. For instance, you can type "Initialise(><myDataVariable)". Using these events has the following effect:

- The model verification engine considers the variable to be valid or invalid after the call

Using this, you can protect yourself from using a data variable before it has been given a value, or from using old data that is no longer relevant. When a value becomes irrelevant to you, simply use Invalidate() to let the model verification know that the data variable should not be used anymore. You will then get a model verification error if there is a trace in the model where the data variable is used after being invalidated.

Note: you can *not* use Initialise() or Invalidate() on rule case-local variables.

Automatic Initialisation

Depending on the use case, it can be handy to initialise data variables at construction time. This is facilitated by the Auto-Initialise field of the Data Variables table. If this field is set to "Yes", the variable is considered by the model verifier to be valid at start-up. Therefore it can be used immediately, e.g. by passing it to an output parameter in a trigger.

When the Auto-Initialise field is set to "No", the data variable is assumed to be invalid, and the variable needs to be written to explicitly before it can be read. The model verifier will check whether this is always the case.

Automatic Invalidation

A frequently-occurring use case for data variables is to carry data for the duration of an action sequence. In this case it is not the intention for data to "leak" from one action sequence to another. Often, a rule case-local variable is sufficient for this, but when an action sequence stretches across multiple rule cases (e.g. because you do a valued call to a used component), you need a data variable.

To facilitate checking that no data leaks from one action sequence to another, you can set the Auto-Invalidate field of a data variable to "Yes". The data variable is then invalidated at the end of every action sequence. At the start of a next action sequence, the variable is considered to be invalid and you will get a model verification error if you try to use it before writing to it.

When the Auto-Invalidate field of a data variable is set to "No", the data variable is not invalidated at the end of an action sequence.

Note: you may not use both auto-initialise and auto-invalidate for the same data variable. This is also not useful, since initialising a value only at start-up and then invalidating it at the end of every action sequence is a-symmetric.

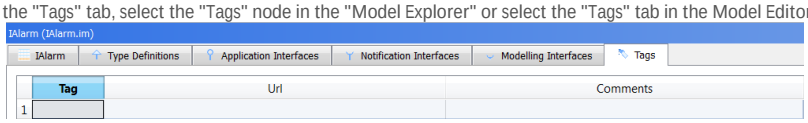
Refactoring Data Variables

For changing the name of all occurrences of a data variable, see Rename Data Variable.

Creating Tags

The Tags tab can be used to record requirements or to include a links to external requirements. These can be requirements that were already defined or requirements that emerge during the design process. Tags can be referred to in the SBS tab. See "Specifying tags" for details.

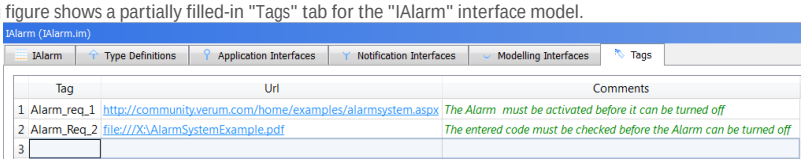
Note: To see the "Tags" tab, select the "Tags" node in the "Model Explorer" or select the "Tags" tab in the Model Editor.



The "Tags" tab

To create a tag, fill in the requirement identification in the "Tag" column and the text of the requirement in the "Comments" column. Optionally, you can fill in an URL in the URL column. This URL can be opened by Ctrl+clicking on it. You can specify any URL that your operating system knows how to open.

The following figure shows a partially filled-in "Tags" tab for the "IAlarm" interface model.



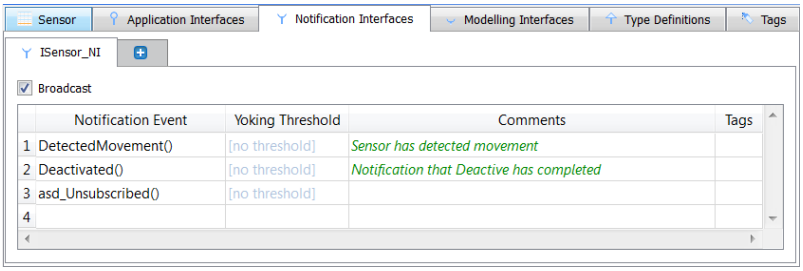
Filled in "Tags" tab

You can find all references to a tag by right-clicking on it and selecting "Find Tag References". This will highlight all references to the tag in the model in green. The results will also show in the Find Results window.

Broadcasting Notifications

In ASD by default notifications manifest themselves as a point to point communication between the ASD component and the used service where the notification interface was defined.

In case you want a component to broadcast events defined on a notification interface to more than one observer (client component using your ASD component) you have to define the respective notification interface as a broadcasting interface by check-marking the "Broadcast" check-box for the defined notification interface. The following figure shows an ISensor_NI notification interface defined as a broadcasting interface.

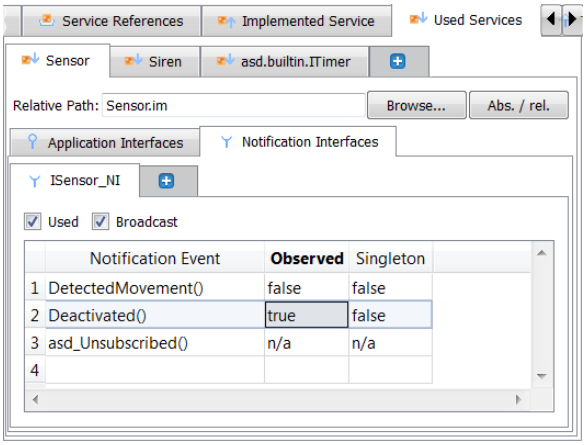


A notification interface flagged as broadcasting

In case you specify in a design model a broadcasting notification interface as a used interface you have the possibility of specifying which events on the respective notification interface you are going to observe.

Note: In case the respective notification interface is newly created and specified as used interface for the first time, its events will not be visible in the SBS of the design model. This is caused by the fact that the notification events are flagged as non observed by default. If you want to observe any notification event from the respective interface you have to flag the respective notification event as observed.

The following figure shows the situation in which you are interested only in Deactivated() notifications:



Setting notification events as observed or not

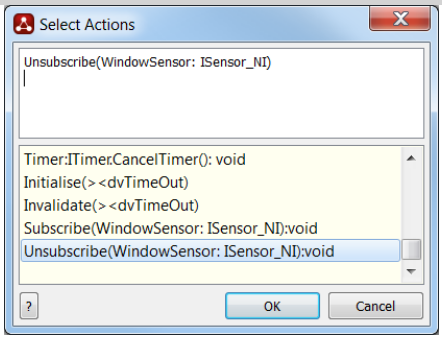
Note:

- IMPORTANT: Since initially the broadcasting interfaces are unsubscribed you have to explicitly subscribe to all used instances of notification interfaces which are flagged as broadcasting. For example, you have to specify Subscribe(sensor:ISensor_NI) if the used service instance name is "sensor" and the broadcasting notification interface is "ISensor_NI."
- You are able to subscribe, respectively unsubscribe at any moment by using the two actions Subscribe and, respectively Unsubscribe. The Unsubscribed status is reported by an "asd_Unsubscribed" notification event. Therefore, you will have to specify behaviour for the respective notification event in all places where it can occur. The unsubscribed status means that the unsubscribe request is processed and that there will be no more notification events on the unsubscribed interface in the queue after the asd_Unsubscribed event. See following figures for an example:
  - Subscribe to "WindowSensor:ISensor_NI" instance of the ISensor_NI broadcasting interface defined in the WindowSensor used service:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**NotActivated (initial state)**					
4	IAlarmSystem	SwitchOn+		Subscribe(WindowSensor:ISensor_NI); WindowSensor:ISensor.Activate; IAlarmSystem.Ok		Activated_Idle
5	IAlarmSystem	SwitchOff		Illegal		-
6	WindowSensor:ISensor_NI	DetectedMovement		Illegal		-
7	WindowSensor:ISensor_NI	Deactivated		Illegal		-
8	WindowSensor:ISensor_NI	asd_Unsubscribed		Illegal		-
9	Timer:ITimerCB	Timeout		Illegal		-

Data in the SBS tab showing the use of the Subscribe action

  - Unsubscribe from the "WindowSensor:ISensor_NI" instance of the ISensor_NI broadcasting interface defined in the WindowSensor used service:



Selecting the Unsubscribe action in the Action Editor

- Process the result of unsubscribing:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
37	**Unsubscribing**					
40	IAlarmSystem	SwitchOn+		Illegal		-
41	IAlarmSystem	SwitchOff		Illegal		-
42	WindowSensor:ISensor_NI	DetectedMovement		Illegal		-
43	WindowSensor:ISensor_NI	Deactivated		Illegal		-
44	WindowSensor:ISensor_NI	asd_Unsubscribed		IAlarmSystem_NI.SwitchedOff		NotActivated
45	Timer:ITimerCB	Timeout		Illegal		-

Data in the SBS tab showing the handling of the "asd_Unsubscribed" notification event

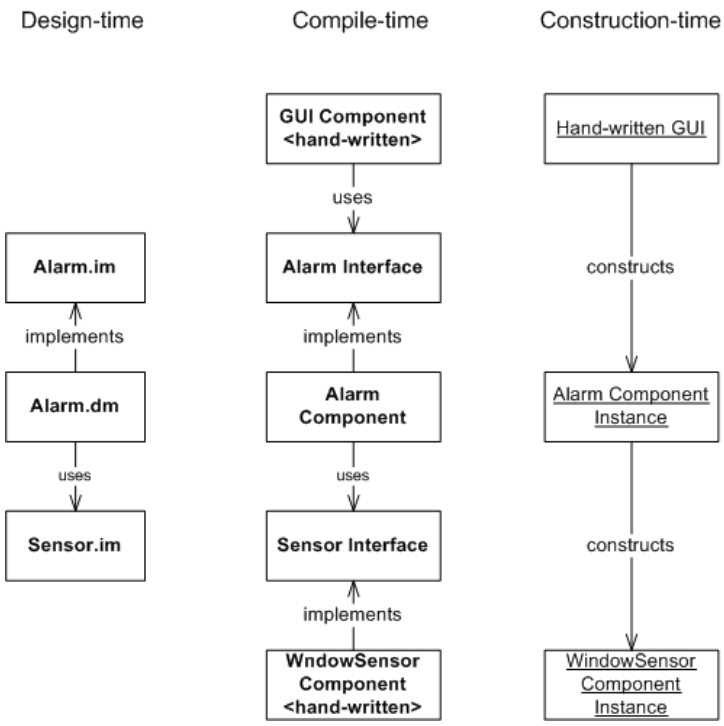
- In case you call Subscribe on an instance of a broadcasting notification interface more than once before unsubscribing only the first request is considered. In other words, calling Subscribe on an already subscribed broadcasting interface has no effect.
- Every Unsubscribe event will be followed by an asd_Unsubscribed notification event. Even when you call an Unsubscribe on a broadcasting interface multiple times, or when you are not subscribed.
- In case you specify one event as not observed, the event will not appear as trigger in the SBS tab for your design, i.e. you will not have to specify a (set of) rule case(s) for the respective notification event.
- Since the choice of observing a large set of notification events from the publishers you subscribed to, and the (sometime) large number of the respective events might cause a saturation of the queue, it might be useful to define one or more of the respective notification events as Singleton Event. See "Singleton Notification Events" for details.

Construction Parameters

Component instantiation - main concepts and default behaviour

From every interface model, an *interface* is generated in code. From every design model, a *component* is generated which implements its interface. Of this component, you can construct one or more *instances* at run-time. Also, you can define that an ASD component constructs instances of other components: you do this by defining service references.

Each service reference defines the service (interface) that it expects, and also the component to instantiate to implement the interface. The latter is specified in the "Construction" field.



models, components, instances

For instance, consider the picture above. In the leftmost column, we have an Alarm.dm design model which implements the Alarm.im interface model. In turn, it has a service reference which refers to the Sensor.im model. This service reference has "WindowSensor" in its Construction field, indicating that it should construct an instance of a component called "WindowSensor" at construction-time.

The second column depicts the compile-time situation: presumably, there is hand-written code all around the generated ASD code: a GUI, which uses the Alarm component, and a hand-written Sensor implementation which is called "WindowSensor".

Finally, in the third column, we get to the situation at construction-time: when the application is started, the GUI constructs an instance of the Alarm component, which in turn creates an instance of the WindowSensor component.

Every generated component has a static method called GetInstance() (or, in some languages, *_getInstance()*, or *getInstance()*). Every hand-written used component must have one too. This GetInstance() method is used to create instances of the component. So, at run-time, the GUI actually calls AlarmSystemComponent::GetInstance() to get a new instance, which in turn calls WindowSensorComponent::GetInstance().

For more details see information about component instantiation/integration in C++, C#, Java, C or TinyC

Instance construction - alternatives

There are various ways to influence how component instances are created.

Firstly, you can pass parameters to a component instance at construction time. Within an ASD component, you can pass these construction parameters along to used components. Examples are in "Passing parameters".

Second, for ASD components, you can set the Component Type to "Singleton" or "Multiple" (see "Specifying the component type" for details). Setting the Component Type to "Singleton" has the effect that only one instance is ever created. Setting the Component Type to "Multiple" causes a new instance to be created for every use.

But what if you want two ASD components to share an instance, without using a Singleton component? Or what if you want to determine the class used for a hand-written component at construction time? Instead of letting the parent ASD component construct an instance, you can also pass a used component instance to an ASD component as a parameter. This way, you have full control over how many instances of which type are constructed, and where they are used. Examples are in "Passing an instance of a used component", "Passing a vector of instances", "Passing a shared instance", or in "Passing a service reference".

Passing parameters to a component at construction time

It is not unusual for a hand-written component to need some data when it is constructed. In the Construction field of a Service Reference, you can enter construction arguments to be passed to the used component in its GetInstance call. These can be literal values, or they can be construction parameters that the parent component got from its own GetInstance call in turn.

Example: literal construction argument

This is an extension of the AlarmSystem example that you can download from the ASD:Suite Community website. Suppose that the hand-written WindowSensor component has a construction parameter "sensorID" of type string.

In the AlarmSystem design model, you can set this parameter in the Construction field of the service reference:

AlarmSystem | Data Variables | Service References | Implemented Service | Usec

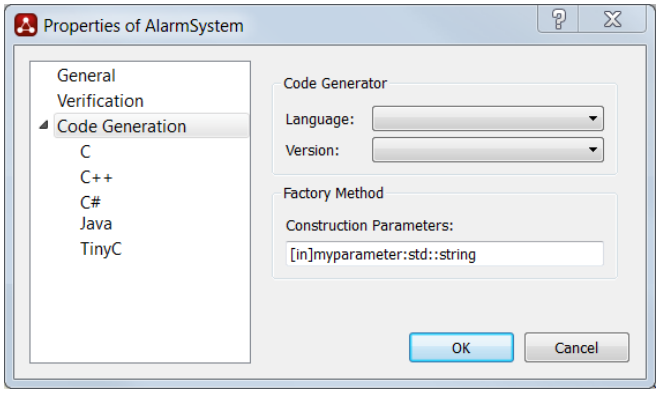
	Reference Name	Cardinality	Verification Cardinality	**Construction**	Service
1	WindowSensor	1	[same as Cardinality]	WindowSensor($"ID12"$)	Sensor
2	Siren	1	[same as Cardinality]	Siren	Siren
3	Timer	1	[same as Cardinality]	asd.builtin.ITimer	asd.builtin.ITimer

Literal construction argument for WindowSensor

As you can see, we pass the literal string value "ID12" to the WindowSensor component. Literal values are specified between dollar signs. They can be anything that is valid in the programming language you use. This argument is supplied in the call that AlarmSystem makes to the WindowSensor::GetInstance at construction time.

Example: passing a construction parameter as construction argument

Instead of supplying a literal value, you can also pass along another construction parameter from the parent component. In a design model, you can define your own construction parameters. Go to the Model Properties, and then click Code Generation. Make sure that the "Interface" radio button is selected. Now you can enter your own construction parameters.



Defining your own construction parameter

In this case, we have defined a parameter with name "myparameter" of type "std::string". The type can be anything that your programming language allows; in this case it is a C++ string. You can define multiple parameters, separated by commas.

The construction parameters you define here end up as formal parameters to the GetInstance method of the generated component. This is described in more detail in component instantiation/integration in C++, C#, Java, C or TinyC.

Now that we have defined a parameter, we can use it to pass values to our used components:

AlarmSystem | Data Variables | Service References | Implemented Service | Used Services

	Reference Name	Cardinality	Verification Cardinality	**Construction**	Service
1	WindowSensor	1	[same as Cardinality]	WindowSensor(myparameter)	Sensor
2	Siren	1	[same as Cardinality]	Siren	Siren
3	Timer	1	[same as Cardinality]	asd.builtin.ITimer	asd.builtin.ITimer

Passing a construction parameter to a used instance

As you can see, we have adapted the WindowSensor Construction field to include the "myparameter" parameter. Any value that is passed to an AlarmSystem instance is now passed to its WindowSensor instance in turn.

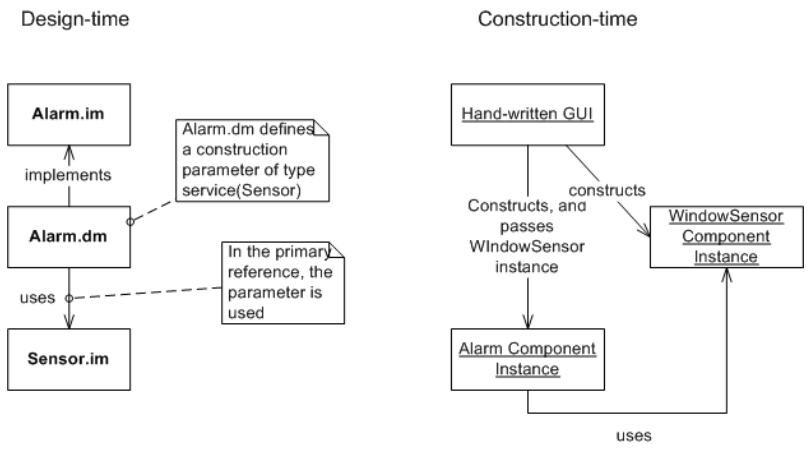
At construction time, the AlarmSystem component can now be instantiated as follows (C++ example):

```
myAlarmInstance = AlarmSystemComponent::GetInstance("my string value");
```

Passing an instance of a used component at construction time

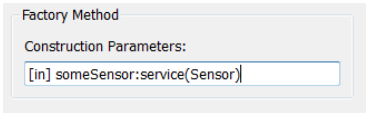
Instead of letting a component construct its own used instances, you can also pass instances to it at construction time.

For this to work, the component must define a special type of construction parameter, and then use this construction parameter in the service reference.



Passing a used component instance at construction time

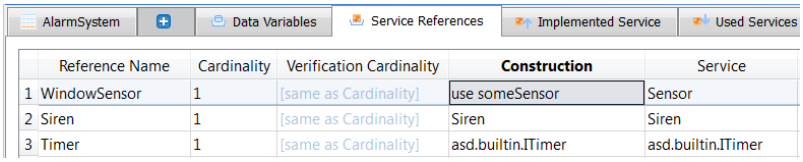
First, we define the construction parameter in the AlarmSystem design model. Go to the Model Properties and click Code Generation.



Construction parameter for a used service

This time, we use the special syntax "service(modelname)" for the parameter type. The model name must match the interface model name of the service reference - in this case, "Sensor" (i.e. NOT the file name!), see also the "Service" field in the next figure. Now, the AlarmSystem component requires a parameter at construction time. This parameter must be filled in with an instance of a component that implements the Sensor interface. The exact effect of this in your code is described in component instantiation/ integration in C++, C#, Java, C or TinyC.

What we still have to do, is make use of this parameter for the WindowSensor service reference:



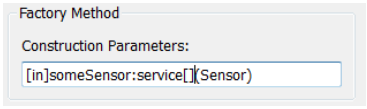
Using a construction parameter for a used instance

We have used the special syntax "use parametername" to denote that instead of constructing a new instance, the AlarmSystem component should use the construction parameter called "someSensor" that was defined in the model properties.

Passing a vector of instances at construction time

In "Passing an instance of a used component at construction time", exactly one sensor instance was passed to the component. But it is also possible to pass a vector of sensors to the component. This way, you can defer the decision of how many sensors to construct to the hand-written client.

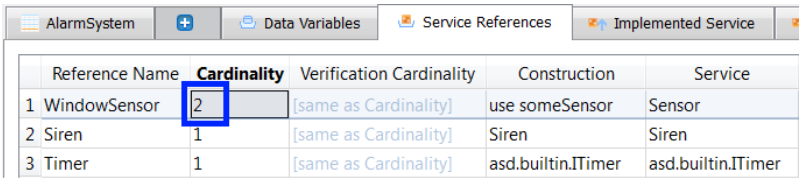
Firstly, we change the type of the construction parameter:



Service vector construction parameter

Note that there are now square brackets after the word "service". The parameter type "service[](*modelname*)" denotes a vector of instances of components that implement the given interface model. The exact effect of this declaration on the generated code is described in component instantiation/integration in C++, C#, Java, C or TinyC.

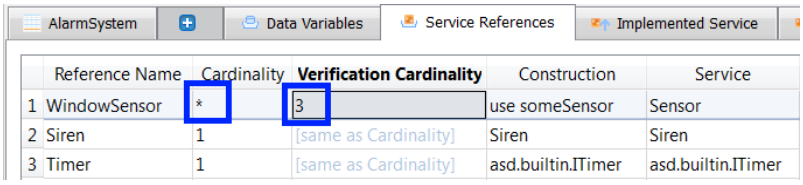
A construction parameter of type service[] can be used for any service reference that does not have length 1. Below is an example where the construction parameter is used for a service reference of length 2. Note that passing a vector of a different length at construction time will result in a run-time error.



	Reference Name	**Cardinality**	Verification Cardinality	Construction	Service
1	WindowSensor	2	[same as Cardinality]	use someSensor	Sensor
2	Siren	1	[same as Cardinality]	Siren	Siren
3	Timer	1	[same as Cardinality]	asd.builtin.ITimer	asd.builtin.ITimer

Using a service[] construction parameter

This construction parameter can also be used in combination with the asterisk feature. This allows passing different amounts of sensors at construction time (note that the model checking is limited to the value specified as "Verification Cardinality"):



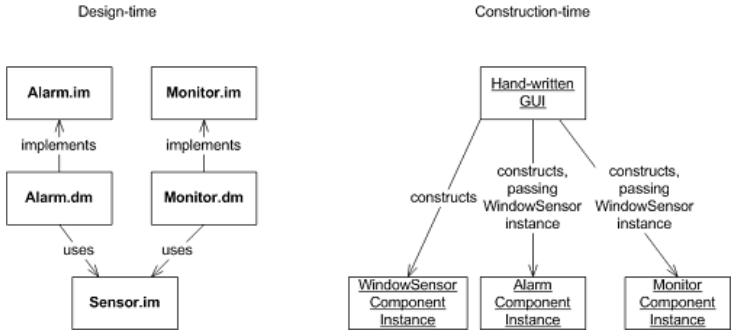
	Reference Name	Cardinality	**Verification Cardinality**	Construction	Service
1	WindowSensor	*	3	use someSensor	Sensor
2	Siren	1	[same as Cardinality]	Siren	Siren
3	Timer	1	[same as Cardinality]	asd.builtin.ITimer	asd.builtin.ITimer

Using a service[] construction parameter with the asterisk

Passing a shared instance at construction time

Suppose you want two ASD components to share the same instance of a used component, and that you don't want to make the used component Singleton. What you can do, is pass the shared instance as a construction parameter to both components.

In the example below, we have an extra "Monitor" component next to the AlarmSystem which monitors the sensor. It should monitor the very same sensor that the AlarmSystem is using, so we want to pass the same instance to both components.



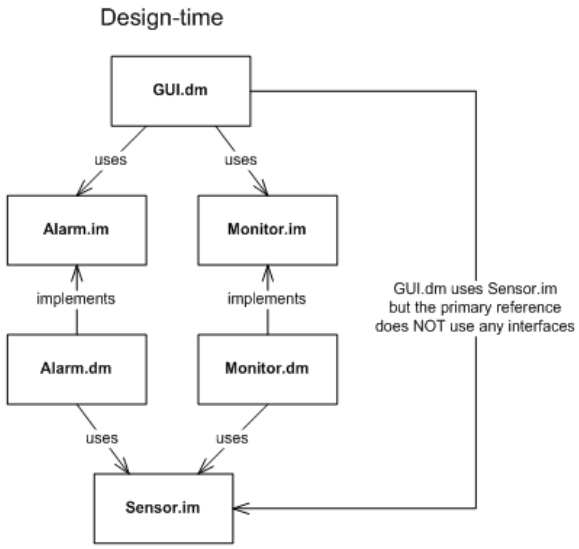
Passing the same instance to two components

In both Alarm.dm and Monitor.dm, we can define a construction parameter of type "service(Sensor)". At construction-time, we can create a Sensor instance and pass it to both components. In C++, this would look like:

```
mySensorInstance = WindowSensorComponent::GetInstance();
myAlarmInstance = AlarmSystemComponent::GetInstance(mySensorInstance);
myMonitorInstance = MonitorComponent::GetInstance(mySensorInstance);
```

Passing a service reference at construction time

Suppose there is an ASD component on top of the Alarm and the Monitor. This component can have a service reference to Sensor.im as well. It can pass this service reference to its other used components.



Defining an instance to pass along within ASD

In the top-level component, we define a service reference to Sensor. *However, in the Sensor tab under Used Services, we mark all of the application and notification interfaces as "not used"*. This is done by unchecking the "Used" checkboxes on each of the interface tabs within the Sensor tab.

We can now pass the mySensor service reference as a construction argument to the other service references:

GUI | Data Variables | Service References | Implemented Service | Used Servi

	Reference Name	Cardinality	Verification Cardinality	Construction	Service
1	mySensor	1	[same as Cardinality]	WindowSensor	Sensor
2	myMonitor	1	[same as Cardinality]	Monitor(mySensor[1])	Monitor
3	myAlarm	1	[same as Cardinality]	AlarmSystem(mySensor[1])	AlarmSystem

Passing along a service reference

This way, you can defer the decision of what instance to construct to a higher-level ASD component without any hand-written code. Effectively, the three lines of code in the previous example are now reduced to just one:

```
myGuiInstance = GuiComponent::GetInstance();
```

Refactoring ASD Models

Like code, ASD models are not static: you will usually want to change them over time. Changes to an ASD model, like changes to a code file, may cause other files to need changes as well. The ASD:Suite helps you with various functions to refactor a model, and other functions to automatically adapt a design model to changes in its interface models.

The relationship between Design and Interface Models

Design models refer to their implemented and used interface models by *relative file path*. Changing such a path (see "Replacing an Interface Model") makes the design model refer to another interface model instead

Interface models contain interfaces and events, and events contain parameters. A design model refers to these interfaces and events and parameters *by name*. This means that when you rename an event in an interface model, the design model no longer refers to it properly.

Since interface models and design models are edited independently, a mismatch can occur between them; e.g. an event can be deleted from an interface model but not yet from a design. This is similar to method declarations in C++ header and code files - they need to match. The mismatches are reported as conflicts. The conflicts have wizards to help you solve them. Alternatively, you can edit both models manually to make them match: in a design model, you can edit the events and interfaces just as you would in an interface model. The only difference is that in a design model, the event declarations do not contain all information, e.g. comments, parameter directions, parameter types etc are only present in an interface model. The sections below describe how to efficiently make changes to your models and keep them consistent.

Keeping Design Models Consistent with their Interface Models

To keep models consistent, there are a number of options:

- **Manually**: you can edit interfaces and events in design models as well as interface models. Making the same changes to a design model as well as the interface model ensures there are no mismatches.
- **Conflict Wizards**: any difference between a design model and its interfaces are reported as conflicts. The conflicts have wizards that automatically resolve each conflict in different ways. Click the "Solution" cell in the Conflicts Window to see the available wizards for a conflict. Note that only conflict *errors* need solving; models with conflict *warnings* can be verified and code can be generated from them.

Functions for Refactoring a Model

- Changing the Initial State for an SBS
- Duplicating a State
- Moving Events to a Different Interface
- Deleting Events or Interfaces
- Renaming Events or Interfaces
- Changing Event Parameters
- Changing the Order of Events or Interfaces
- Replacing an Interface Model
- Renaming Data Variables in the Entire Model

Changing the Initial State for an SBS

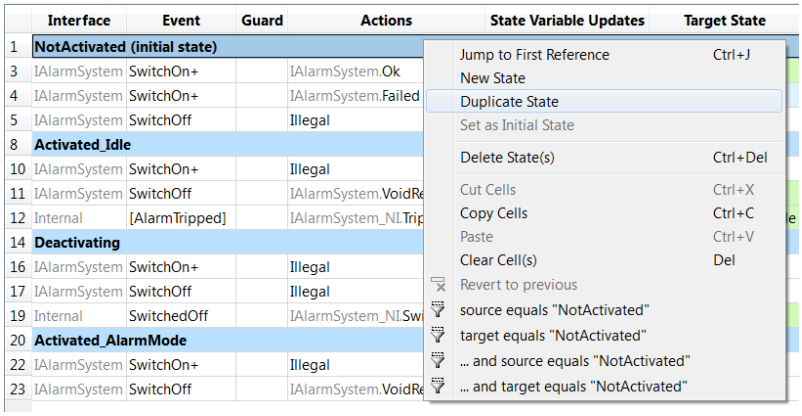
When you create a model, the SBS has one state: the initial state. If at some point you want to set another state as initial state, there are three ways to do this:

1. In the States table, move the desired state to the top of the table using Ctrl+Up
2. In the States table, right-click the desired state and choose "Set as Initial State"
3. In the SBS table, right-click the desired state and choose "Set as Initial State"

Duplicate a state

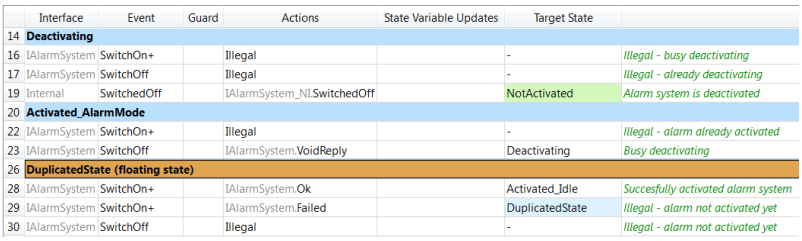
If you want to specify the same or at least similar behaviour in another state than in an existing one you might try to duplicate the existing state and then make the necessary modifications in the new state.

In order to duplicate a state you have to select the respective state and select the "Duplicate State" item in the context menu of state (see next figure).



Selection for Duplicate State

The following figure shows the result of naming the new state DuplicatedState:



The duplicated state

Moving Events to a Different Interface

You may want to re-categorize your events into different interfaces. However, deleting an event from one interface and adding it to another interface causes all the related rule cases in the SBS to be deleted. To solve this, *move* the events between the interfaces of a service instead.

- Within the interface model, select one or more events
- Right-click, and choose "Move Event(s) to Interface..."
- You now get a dialog where you can select the target interface

The events will be moved and all rule cases preserved.

Moving an event in an interface model causes mismatches with any design model that refers to it. There are two solutions: one is to perform the same move action in the design model. The other is to check the design model for conflicts and use the conflict wizard to do the move. Note however that since the design model refers to events *by name*, the conflict wizard cannot always tell whether you moved the event or deleted and added it. You are presented with both solutions, be careful to select the one you need.

Deleting Events or Interfaces

Deleting events and interfaces can be done by selecting them and pressing Ctrl+Delete. You can do this both in interface models and in design models. Their corresponding rule cases in the SBS will automatically be deleted as well. If any of these rule cases were non-empty, you will get a warning dialog first. Deleting an event or interface from an interface model will cause a conflict in any design that uses it. Conflict wizards are available to delete the event from the design model as well.

Renaming Events or Interfaces

Renaming events or interfaces is done by simply editing their name in their respective table. The names in the SBS automatically change too. However, since the link between a design model and the events in the interface models is *by name*, the rename must be performed in both models. Renaming only in one model causes a conflict. Conflicts wizards are available to solve the problem. In this case however they cannot reliably see whether you renamed the event or deleted one event and added another. The correct solution will usually be suggested to you first, but sometimes you may need to pick another solution than the top one from the list.

Changing Event Parameters

Changing event parameters in an interface model can be done simply by editing the event. The SBS of the interface model is not affected - it automatically shows the edits.

Any dependent design model, however, needs to have its SBS updated. For instance, if you add a parameter, the design model needs to supply arguments for that parameter. This is a design choice that cannot be made automatically. However, the ASD:Suite tries to help out for the most common cases. Any differences in parameters between a design model and its interface models are reported as conflicts.

The conflict wizards will offer you two choices: "Copy parameters and repair SBSes" and "Copy parameters to design model". The first solution uses heuristics to change the SBS. For instance, if you change only the order of parameters, the SBS arguments are re-ordered to match. If you delete a parameter, arguments are deleted; if you add a parameter, arguments are added.

Note, however, that parameters are matched by name, so a renamed parameter cannot be repaired. Choose the "Copy parameters to design model" solution in this case to avoid arguments being deleted and added.

Changing the Order of Events or Interfaces

You can change the order of events simply by selecting them and pressing Ctrl+Up or Ctrl+Down to move the table row up or down. Changing the order of interfaces is a simple matter of selecting the interface tab and pressing Ctrl+Left or Ctrl+Right to move the tab left or right. Similar actions are available in the Model Explorer. Changing the order in an interface model has no effect on design models that use it: their order does not change, and also there will be no conflict about a mismatch.

To make the order of events in a design model similar to the order in the interface model, right-click an event in the design model and choose "Sort like Interface Model". Note that the interface model must be loaded for this to work.

Replacing an Interface Model

A design model refers to its interface models by relative file path. You can change which interface models are used simply by changing that relative path.

- Go to the Implemented Service or Used Services tab
- Select the tab of the model you want to replace
- Use the "Browse..." button to select another interface model.

If the events in the new interface model are not the same as in the old one, the differences are reported as conflicts. You can then resolve the differences in all the ways described in "Refactoring ASD Models", usually by using the conflict wizards.

Changing a relative path is an undoable action: pressing Edit-Undo will simply change the relative path back and the conflicts are gone.

Renaming Data Variables

To rename a data variable, you can simply change its name in the Data Variables table. However, that does not rename the *uses* of that data variable in the SBS. To include these uses as well, do *not* rename the variable in the table, but right-click the variable name and choose "Rename all occurrences".

Save As

The ModelBuilder enables you to save your currently opened ASD model as a new model using the currently specified behaviour as starting point, or as an exact copy.

Note: The newly created model will be opened in the current instance of the ModelBuilder.

Using the "File->Save <model_name> As...->Copy..." menu item you can save an exact copy of the currently opened ASD model. Saving an exact copy comes handy in the following cases:

- If you are not able to save your ASD model because it was opened in a read-only mode or because the physical location is not available anymore, or
- In case you make changes and you save them and they do not turn out to be the right changes

Note:

- In case the currently opened ASD model is a design model only an exact copy of the design model will be saved, no copies are saved for the referenced, i.e. implemented or used, interface models.
- When creating an exact copy the currently opened model will not be saved.

Using the "File->Save <model_name> As...-> New Model..." menu item you can create a new model using the specified behaviour in the currently opened model as a starting point.

Note:

- If the model name or the main machine name matched the old file name, they are automatically renamed to the new file name.
- When creating a new model you have the opportunity to save your currently opened model before the newly created model is loaded in the ModelBuilder.

Guidelines for names and identifiers

Within an ASD model there is a number of keyword that cannot be used as identifiers. For example, you cannot use "Illegal" as a name for an event, as this conflicts with the built-in 'Illegal' action. Depending on the scope or location, some keywords are allowed in one location and disallowed in others.

To prevent problems we recommend not to use the following names for identifiers in you ASD models:

- Illegal
- Blocked
- Disabled
- NoOp
- StateInvariant
- DataInvariant
- VoidReply
- NullRet
- Integer
- Enumeration
- Boolean
- true
- false
- and
- or
- not
- nil
- head
- tail
- that
- otherwise
- service
- use

In addition to these keywords, we also recommend the following:

- you should not use any of the reserved words in the output languages of the ASD compiler (C++, C#, C, Java)
- you should not start an identifier name with "asd_"
- you should not start an identifier name with an underscore ("_")
- you should not start an identifier name with a digit
- you cannot use a namespace "asd.builtin."
- you cannot use non-alphanumerical characters
- you cannot use Unicode characters. Only ASCII characters are supported.

The following grammar defines the validity of a name used in ASD modelling:

- **ValidName** = Letter { _ | Letter | Digit}
- **Letter** = any character from "a" to "z" or from "A" to "Z"
- **Digit** = any number between and including 0 and 9

Conflicts

Your models have to conform to certain rules before they can be verified or before code can be generated from them. Most of these rules are syntactical, others are about the relationship between a design model and its interface models. The ModelBuilder can automatically check the rules and it will highlight the locations where the rules are violated.

Types of Conflicts

There are two types of conflicts:

- **Errors**, which need to be fixed before code can be generated or the model can be verified. Errors are highlighted in the model in orange.
- **Warnings**; Models with only warnings can be verified and code can be generated. Warnings are highlighted in the model in yellow.

All conflicts are displayed in the Conflicts Window. Double-clicking a conflict shows the location of the conflict in the model. Vice versa, hovering with the mouse over a highlighted location in the model gives the conflict message in a tooltip.

Note: When jumping to a filtered-out rule case, that single rule case will be unhidden. You can hide the respective rule case again, by (re-)applying the filter(s). See "SBS Filters" for details about applying filters.

Note: Each conflict has an associated error code. Please report that code if you have difficulties in fixing the conflict(s). Clicking with the mouse on the error code will open a webpage in your default browser with some guidelines to fix the conflict.

Checking for Conflicts

A conflicts check is done automatically when a model is opened, verified or when code is generated. You can also manually check for conflicts: Select the "Tools->Check Conflicts" menu item.

Continuous Conflicts Checking

If a conflicts check (either automatic or manual) results in conflicts, then the conflicts are re-checked every time you make a change to the model. Once the last conflict is fixed, re-checking conflicts is automatically disabled again. For large models, this continuous conflicts checking might be inconvenient. For this reason, you can disable this behaviour under Tools-Options-Appearance.

Clearing the Highlighting

You can remove the orange and yellow highliting from the model by pressing Tools-Clear Conflicts. This also clears the Conflict Results window.

Fixing conflicts

If your models have conflicts, you can change your model manually to fix the conflicts. The table below has suggestions for what to change for every conflict. However, the Suite can fix many conflicts automatically. In the Conflicts Window, there is a column called "Solutions". If there are automatic solutions available for a conflict, this will display a 'wizard' icon and the number of available solutions. To automatically solve a conflict:

- Check that the "Solutions" cell indicates that there are solutions available.
- Click the "Solutions" cell. A dialog pops up.
- Most conflict solutions can be applied to all conflicts of the same type. In this case, you can check the "Do this for all similar conflicts" checkbox.
- Choose the solution you want to apply.

Caveat: usually, the first solution that is presented is the one you will need. However, there can be other solutions for the conflict, and they may be better suited to your particular situation. Also, in some cases, it is handier to solve the conflict by hand regardless.

This is an overview of all error codes reported in the ASD:ModelBuilder Release 4 v9.2.7 Click on the error code for more details:

					RC5	RC6	RC7	RC8	RC9
RC10	RC11	RC12	RC13		RC15	RC16	RC17	RC18	RC19
RC20	RC21	RC22	RC23	RC24	RC25	RC26	RC27	RC28	RC29
	RC31	RC32	RC33	RC34	RC35	RC36		RC38	
	RC41	RC42	RC43	RC44		RC46		RC48	RC49
RC50	RC51	RC52		RC54	RC55		RC57	RC58	RC59
RC60	RC61	RC62	RC63	RC64	RC65	RC66	RC67	RC68	RC69
RC70	RC71	RC72	RC73	RC74	RC75		RC77	RC78	RC79
RC80			RC83	RC84		RC86	RC87	RC88	RC89
RC90		RC92	RC93	RC94	RC95	RC96	RC97	RC98	RC99
RC100	RC101	RC102	RC103	RC104	RC105	RC106	RC107		RC109
RC110	RC111	RC112		RC114		RC116		RC118	
	RC121	RC122		RC124	RC125	RC126	RC127		RC129
RC130	RC131	RC132	RC133	RC134	RC135	RC136	RC137	RC138	RC139
RC140		RC142	RC143	RC144	RC145	RC146	RC147	RC148	RC149
		RC152	RC153				RC157	RC158	RC159
RC160	RC161	RC162	RC163	RC164	RC165	RC166			
	RC171	RC172	RC173	RC174	RC175	RC176	RC177	RC178	RC179
RC180	RC181	RC182	RC183	RC184	RC185	RC186	RC187	RC188	RC189
RC190	RC191	RC192	RC193	RC194	RC195	RC196	RC197	RC198	RC199
RC200	RC201	RC202	RC203						

This is an overview of all error codes reported by the ASD Server Release 4 v9.2.7 Click on the error code for more details:

		CGE03	CGE04	CGE05	CGE06	CGE07		CGE09	CGE10
CGE11	CGE12	CGE13	CGE14		CGE16	CGE17	CGE18		CGE20
CGE21	CGE22								

Fixing IM-DM difference conflicts

As described in "Refactoring ASD Models", your design models need to match their surrounding interface models to a degree: for instance, all the events you use in a design model, must be defined in one of the interface models. The models can be edited independently, which can cause them to no longer match. The design model refers to the interface model interfaces and events *by name*. This means that when you change a name in an interface model, you also have to change it in the design model. Usually, you can use the Conflict Wizards to make corresponding changes to a design model.

Error code	Explanation	Fix
RC142	Your design model refers to an interface that is not present in the interface models. It can be that the interface was deleted from the interface model, but also that it was renamed. See "Refactoring ASD Models" for more details.	If the interface was deleted from the interface model, delete it from the design as well. If the interface was renamed, rename it in the design model too.
RC143	There is an interface in the interface model that is not in the design model. It can be that you deleted it from the design model, that you added it to the interface model, or that it was renamed. See "Refactoring ASD Models" for more details.	If the interface was renamed in the interface model, rename it in the design model too. If it was added to the interface model, add the interface. It is best to let the Conflict Wizard handle this, so that the events are copied as well.
RC144	There is an event which is present in the design model but not in the interface model. It can be that the event was deleted from the interface model, or that it was renamed. See "Refactoring ASD Models" for more details.	If the event was renamed in the interface model, rename it in the design model too. If it was deleted from the interface model, delete it in the design too. Note that this may cause rule cases to be deleted.
RC145	There is an event which is present in an interface model but not in the design model. It can be that it was added to the interface model, deleted from the design, or that it was renamed. See "Refactoring ASD Models" for more details.	If the event was renamed in the interface model, rename it in the design too. If it was added, add it to the design model. It is best to let the Conflict Wizard handle adding the event (copying it from the interface model), so that the parameters are copied automatically as well.
RC146	The notification interface in the design model is broadcasting while the notification interface in the interface model is not, or vice versa. The broadcast flag can have been changed in either model. See "Refactoring ASD Models" for more details.	Change the broadcast flag to match. It is best to let the Conflict Wizards handle this for you.
RC147	The parameters in the design model are different from the parameters in the interface model. Note that direction (in/out/inout) and type do not matter; these are not needed in the design model. The number of parameters, the names and the order of the parameters do matter. See "Refactoring ASD Models" for more details.	Change the parameters to match. Then change the arguments in the SBS to match the parameters. There are two Conflict Wizards for this conflict: one that only copies the parameters, and one that also changes the SBS arguments. **Caveat:** the latter wizard cannot cope with renamed parameters, since it will see those as deleted and added. Therefore it will delete and add arguments in that case. The wizard *is* capable of changing the order of arguments when the order of the parameters has changed, and also of deleting and adding arguments if parameters are deleted or added.
RC148	The reply type (valued/void) of a call event is different between the design model and the interface model. See "Refactoring ASD Models" for more details.	Change the reply type to match. Note that you may have to adjust your SBS.
RC149	You cannot use the same interface model as both an implemented model and used model. Usually, the ModelBuilder will prevent you from doing this.	Change the relative path of either the implemented service or the used service to point to a different interface model.

Fixing syntax related conflicts

These are the syntax related conflicts:

Error Code	Explanation	Fix
RC5	The name of the service violates the syntactical rules for names used in ASD modelling	Ensure that the indicated name complies with the "Guidelines for names and identifiers".
RC6	The name of the design violates the syntactical rules for names used in ASD modelling	
RC7	The name of the interface violates the syntactical rules for names used in ASD modelling	
RC8	The name of the event violates the syntactical rules for names used in ASD modelling	
RC9	The name of the parameter violates the syntactical rules for names used in ASD modelling	
RC10	The name of the main machine or sub machine violates the syntactical rules for names used in ASD modelling	
RC11	The name of the state violates the syntactical rules for names used in ASD modelling	
RC12	The name of the state variable violates the syntactical rules for names used in ASD modelling	
RC13	The name of the used service reference violates the syntactical rules for names used in ASD modelling	
RC110	The name of the tag violates the syntactical rules for names used in ASD modelling	
RC124	The component name in the definition of the specified service reference violates the rules for names used in ASD modelling	
RC130	The name of the construction parameter is invalid.	
RC163	The name of the data variable is invalid.	
RC171	The name of the rule case-local variable is invalid.	
RC176	The name of the enumeration value is invalid.	
RC179	The name of the type definition is invalid.	
RC15	The name of the specified namespace violates the rules for specifying names for namespaces	Ensure that the name of the namespace consists of names separated by dots, and every name consists of an alpha character or an underscore, followed by alphanumericals and underscores
RC112	The name of an ASD model can not be "asd.builtin.ITimer"	Ensure that the model name is not "asd.builtin.ITimer"
RC122	Model names should not be longer than 200 characters	Ensure that the model name is less than 200 characters
RC125	There is an empty Construction field in the specified service reference	Fill in the Construction field, or delete the service reference
RC126	There is a tag with no name	Specify a name for the tag
RC127	Currently a construction parameter can have only [in] as direction	Ensure that the direction of the construction parameter is [in]
RC129	The specified construction argument is not declared	Ensure that the construction argument, if not a literal, is declared as a construction parameter in the design model properties, or as a service reference
RC131	There is a syntax error in the construction field of the specified service reference	Ensure that the data specified in the construction field complies with the syntax for declaring construction parameters
RC132	There should be no value defined in the "Verification Cardinality" field since the used reference is used only for component injection	Ensure that the "Verification Cardinality" field is empty
RC153	Transfer interfaces can only have valued events, so they cannot have a VoidReply reply event.	Remove or rename the VoidReply event.
RC165	An argument can be one of: o. A rule case-local variable: simply the name of the local variable o. A data variable: a storage specifier (<< >> or ><) followed by the data variable name o. A literal value: any value enclosed in dollars ($). Note that dollars within the literal value must be escaped by using double dollars.	Put the argument expression in one of the forms mentioned. See also "Parameter Usage".
RC166	You used a storage specifier (<< >> or ><) but the name behind the storage specifier does not match any declared data variable.	If you intended to use a rule case-local variable, remove the storage specifier. Otherwise, correct the data variable name or declare the data variable in the Data Variables tab. See also "Data Variables" and "Parameter Usage".
RC188	The namespace prefix in the code generator settings is not valid.	A namespace prefix consists of identifiers separated by dots. Each identifier must start with an underscore or letter, followed by underscores, letters or digits.
RC196	The service name is not valid.	A service name consists of identifiers separated by dots. Each identifier must start with an underscore or letter, followed by underscores, letters or digits.
RC197	The user defined type should not start with a colon.	Ensure that the user defined type does not start with a colon.
RC198	A data invariant should be a list of datavariable.isValid() or datavariable.isInvalid() calls, separated by semi-colons.	Ensure the data invariant is according to the syntax.
RC199	A data variable referenced in the indicated data invariant cannot be found.	Adjust the data invariant or declare the data variable in the Data Variables table.
RC201	A data variable is referenced in the data invariant of a super state or the initial state of a submachine.	Remove the references to data variable in super states and initial states of submachines.
RC202	Using a double unary plus or minus is not allowed in guards, because they are often interpreted as an increment or decrement operator.	Remove the + + or - - from the guard.

Fixing name duplicates

The following table shows the messages for the situations in which a name of some items appears more than allowed:

Error Code	Explanation	Fix
RC16	Each interface in an ASD model must have a unique name	Ensure name uniqueness per indicated item
RC17	Each event, or reply event, in an interface, must have a unique name in the context of the respective interface	
RC18	Each parameter in an event must have a unique name	
RC19	Each state machine in a design model must have a unique name	
RC20	Each state must have a unique name in the state machine in which it is declared	
RC21	Each state variable must have a unique name in the context of the machine in which is declared	
RC22	Each used service reference, used service reference state variable, or construction parameter in a design model must have a unique name in the context of the design model	
RC23	Each event must have a name which is not used as a name for an interface	
RC111	Each tag in an ASD model must have a unique name	
RC164	Each data variable in an ASD model must have a unique name.	
RC177	Enumeration values must be unique across all enumeration type definitions in an ASD model.	
RC178	Enumeration values must be unique with respect to state variable names and service reference names.	
RC180	Type definitions in an ASD model must have a unique name.	
RC200	Giving a state variable and a data variable the same name likely causes confusion, so you'll probably want to give them unique names.	

Note: Model names, file names and interface names must be unique within the scope of the entire project. This is not automatically checked by the ASD:Suite, and need to be ensured by you. If the naming conventions are not met this could lead to verification errors, code generation errors or compilation errors.

Fixing interface related conflicts

The following table shows the specification conflicts related to usage and declaration of interfaces when building ASD models:

Error Code	Explanation	Fix
RC24	Each service should have at least one application interface	Specify at least one application interface in the interface model of the indicated service
RC25	Each interface must have at least one event	Specify at least one event for the respective interface
RC26	Each interface which has at least one valued call event, must have at least one reply event	Specify at least one reply event for the respective interface or make all events void
RC27	Each transfer event must be of type "valued"	Ensure that the specified transfer event is of type valued
RC31	Broadcasting interfaces are not allowed in the *Single-threaded* execution model	Turn off the Broadcast flag for the specified interface
RC32	There is no value in having a yoking value greater than the queue size of the design model	Change either the queue size or the yoking threshold. Note that the yoking treshold is present in the interface model, not in the design model
RC33	Yoking is only usable in the *Multi-threaded* execution model	Remove the yoking threshold for the specified notification event or use the *Multi-threaded* execution model instead of the *Single-threaded* one
RC51	There are no notification events specified as Observed for the specified broadcasting interface	Ensure that there is at least one notification event of the mentioned broadcasting interface specified as observed, or clear the broadcasting flag for the mentioned interface
RC52	It is not allowed to flag ITimer notification events as Singleton events	Ensure that the specified timer notification event is not flagged as Singleton event
RC118	The event queue must be at least of size 1	Ensure in the design model properties that the specified size for the event queue is greater or equal than 1
RC158	There are one or more call events with return type "void", but you do not have a VoidReply event in the corresponding reply events table. You need a VoidReply as answer to void call events.	Add a reply event named "VoidReply" to the reply events table.

Fixing argument, parameter or data variable related conflicts

The following table shows the specification conflicts related to arguments, parameters, or data variables used in building the ASD model:

Error Code	Explanation	Fix
RC28	Events on a Modelling Interface can not have parameters	Remove the parameters from the declaration of the respective event
RC29	Events on a Notification Interface can not have [out] or [inout] parameters	Remove the [out] or [inout] parameter from the declaration of the respective event
RC77	The number of arguments in a specified response must be the same as the number of parameters in the declaration of the event or return event used as response	Ensure that the list of arguments used in the response matches the list of parameters as in the declaration for the event or return event
RC78	The number of arguments in a trigger must be the same as the number of parameters in the declaration of the event or reply event used as trigger	Ensure that the list of arguments used in the trigger matches the list of parameters as in the declaration for the event or reply event
RC79	The Event column of the indicated rule case contains a trigger that has identical arguments to two or more [in] or [inout] parameters. This would make the value of the argument undetermined.	Change one of the argument names
RC80	The indicated rule case has an action with an [out] or [inout] argument that is also used for another argument. This is not allowed since it is not clear which value is actually written to the argument after the action is executed due to reference-sharing	Change one of the arguments
RC83	The storage specifier attached to an argument in the trigger does not match the direction of the parameter	Change storage specifier to conform to the parameter storage process described in "Parameter Usage".
RC84	The storage specifier attached to an argument in the action does not match the direction of the parameter	Change storage specifier to conform to the parameter storage process described in "Parameter Usage".
RC86	Literals should not be empty in an action (i.e. $$)	Fill in a non-empty literal or a valid argument.
RC104	A reply event can not have [out] or [inout] parameters	Remove the [out] or [inout] parameter from the declaration of the respective event
RC105	Data variables used in arguments must be preceded by a storage specifier (<<, >> or ><).	If you intended to use the data variable, add the correct storage specifier. If you accidentally named a rule case-local variable after a data variable, change the name. See also "Data Variables" and "Parameter Usage".
RC106	Literals should not be empty in a trigger (i.e. $$)	Fill in a non-empty literal or a valid argument.
RC107	Literal arguments on a trigger are only allowed for [out] parameters	Change the parameter direction or replace the literal argument with a data variable.
RC109	If non-cloning of parameters is selected, the execution type of the service should be *Single-threaded*.	Ensure that the execution type of the component is *Single-threaded*, or clear the "No Parameter Cloning" flag in the code generator settings.
RC172	Initialising a data variable at construction time and invalidating it at the end of every action sequence is a-symmetric and therefore not useful.	Change one or both of the Auto-Initialise or Auto-Invalidate fields of the data variable to "No". See also "Data Variables".
RC173	Specifying simply "$$" for the Data Type of a data variable is not allowed.	Clear the field. See also "Data Variables".
RC174	The data type of a data variable must be a target language type, enclosed in dollars (e.g. $std::string$ for a C++ string). Any dollars in between must be escaped by using double dollars.	Correct the type. See also "Data Variables".
RC183	The value of the variable mentioned in the conflict was used before any value was stored into it. Note that: • When a data variable is used on the [out] parameter of a trigger, its value is copied at the occurrence of the *reply event*, and not at the end of the rule case or action sequence. • When a rule case-local variable is used on the [out] parameter of a trigger, its value is copied at the end of the rule case. • Data variables that have the Auto-Invalidate field set to "Yes", are invalid at the start of an action sequence. • Data variables that have the Auto-Invalidate field set to "No", are assumed to be valid at the start of an action sequence. The model verifier will correct this assumption if necessary.	Make sure the variable is written to before it is read from. See also "Parameter Usage".
RC184	A variable is written to twice in a row without the value being read in between. This is usually not correct.	Ensure the variable is read between the writes, or that the variable is not written to twice. See also "Parameter Usage".
RC185	The built-in Initialise() and Invalidate() actions are only meant for use with data variables.	Remove the action or correct the name of the argument to a data variable name. See also "Data Variables".
RC186	Only rule case-local variables are allowed on transfer event [out] or [inout] parameters	Remove the storage specifier and do not use a data variable
RC187	Literal values (between dollars) are only allowed on outward-facing parameters. In the Actions column, this means that literals are only allowed on parameters with direction [in].	Use a data variable or rule case-local variable.
RC189	Usually you will not want an unused data variable in your model.	Remove the data variable or use it. Note that just calling Initialise() or Invalidate() on the variable is not considered "use".
RC203	There is a construction parameter in your design model that has type service(X) or service[](X) but there is no interface model with a model name X linked to your design model.	Change the type to the model name of a linked interface model. You can add a new Used Service (but not a service reference) if you need to link an extra interface model.

Fixing used service references related conflicts

The following table shows the messages that inform you that there are specification conflicts related to services used in a design model:

Error Code	Explanation	Fix
RC34	There are one or more specification conflicts in the specified interface model	Check and fix conflicts in the interface model
RC35	The specified interface model is not open/loaded	Add the interface model to the design model by performing the following steps: • Select Models in the Model Explorer. • Select the Add model in the context menu after pressing the right mouse key. • Select the missing interface model and press Open.
RC36	The design model name and the component name can not be the same in the Construction field of a service reference. This is to ensure that a component does not use itself	Change the design model name, or the component name in the Construction field of a service reference
RC38	The declaration of the indicated service reference is not valid. This can be because: • The name is not valid: Used Service Reference names should start with a letter, followed by letters, digits and/or underscores; • The name is already in use for something else (e.g. a keyword or another used service reference).	Ensure that the name of the indicated used service reference complies with the rules for naming used service references and that it is not used for naming any other item in your model
RC41	If you use the asd.builtin.ITimer service, you need to mark all interfaces as "used"	Make sure the "Used" checkbox of the "ITimer" interface is checked, or remove the ASD:Timer service as a used service
RC42	If you use the ASD:Timer service, you need to mark all interfaces as "used"	Make sure the "Used" checkbox of the "ITimerCB" interface is checked, or remove the ASD:Timer service as a used service
RC43	When using the single-threaded execution model, either none or all of the interfaces in the service must be marked as "used".	In the design model, make sure that the "Used" checkboxes of the interfaces of the service are all on or off. See also "Execution Model".
RC44	If the implemented service is *Single-threaded*, all used services must be *Single-threaded* as well. **Exceptions:** it is allowed to have a *Multi-threaded* used service if: • The used service does not have notification interfaces • The used service does not have modelling interfaces • The service reference uses all interfaces	Ensure that all used services use the *Single-threaded* execution model or they fall in one of the exceptional cases. See also "Execution Model".
RC46	The implementation of the ASD Timer requires that the models use the *Multi-threaded* execution model. See "Specifying the execution model" for details.	Ensure that all used services use the *Multi-threaded* execution model, or remove the ASD Timer service as a used service.
RC48	At most 128 instances can be put into a service reference	Lower the value that is specified in the "Cardinality" column
RC49	Each used service reference with unspecified cardinality (i. e. that have an asterisk (*) in the Cardinality column) must have a cardinality specified in the "Verification Cardinality" column	Fill in the "Verification Cardinality" column for the indicated used service reference
RC50	The number of instances for a service reference used in verification should be at least 1 and should not exceed the specified cardinality of the respective service reference	Ensure that the number in the "Verification Cardinality" column is greater than 1 and less than or equal to the cardinality of the service reference
RC133	The service reference is not used as a construction argument to another service reference and it has no used interfaces	Use the service reference as a construction argument to another service reference, or specify used interfaces for it, or remove the service reference
RC134	ITimer is not allowed as service type for a construction parameter	Check your code generator settings and remove asd.builtin.ITimer as service type of the indicated construction parameter
RC135	The argument to a ***use*** construction expression must be a construction parameter of a service() type declared in the code generator settings. It cannot be a service reference, a literal, or a construction parameter of a user-defined type.	Ensure that the indicated argument complies with the syntactical rules of arguments to a ***use*** construction expression
RC136	The service reference refers to a construction parameter of the wrong type	Ensure that the construction parameter and the service reference refer to the same interface model name
RC137	The used instances refered by the specified construction arguments depend on each other and create a cyclic construction dependency	Remove the cyclic dependency
RC138	A construction argument can not be of type asd.builtin. ITimer	Change the type in the declaration of the construction arguments.
RC139	The cardinality of the service reference does not match the size of the specified construction parameter	Ensure that the cardinality of the service reference does match the size of the specified construction parameter, or vice versa
RC140	A USR state variable cannot have the type of an unused service; at least one of the interfaces in the service must be used	Change the USR state variable constraint, or marks at least one of the interfaces in the service as "Used" by checking its "Used" checkbox. See also State Variables for Used Service References".
RC157	If you specify * as cardinality, you must also have a "use" expression in the Construction field to supply the desired number of instances at construction time.	Either specify a fixed cardinality or add a construction parameter to the design model properties and specify "use myConstructionParameter" in the Construction field.
RC159	You have two used service dependencies with the same name.	Change the name of one of the used service dependencies.
RC160	The use of singleton notification events is not supported for models that implement an interface model which has the Single-threaded Execution Model.	Singleton notification events are useless in the Single-threaded model, since the notification events are not processed until the thread of control returns to the component that receives them. Therefore there will be either zero or one event processed, which is usually not the intention when working with singleton events (where you typically want only zero or one in the queue, but multiple events processed if there is processing time). Make the event non-singleton.

RC161	ASD models have an automatically generated unique identifier. This identifier is not visible in the ModelBuilder. If you copy-paste an ASD model e.g. in Windows Explorer, then you end up with two ASD models with the same identifier. It is not supported to have a design model that refers to multiple interface models with the same identifier.	Open one of the interface models, choose File - Save As - New Model, and save the model under its own name. It will be overwritten and get a new unique identifier. In the future, don't copy models in Windows Explorer to create new ones but use File - Save As - New Model.
RC190	Cardinality of a service reference is out of range. The valid range is [1..128].	Update the cardinality of a service reference with a value from the valid range [1..128].
RC191	Construction parameter is not a vector and is used with an index.	Either change the construction parameter into a vector, or remove the index from the parameter in the construction field.
RC192	The index in an argument of a construction expression is out of range. The argument is a vector with a limited range, the index should be within this range.	Change the index in in the argument to a value within the valid range, or update the cardinality of the vector being passed.
RC193	There is a used service that has no service references to it	Either create a service reference or delete the used service
RC194	The Service field of the service reference is empty.	Select a service in the Service field.
RC195	The two indicated Used Services refer to the same interface model, have the same events and the same Observed, Singleton and Used flags.	Use the conflict wizard to merge the two services.

Fixing rule case related conflicts

The following table shows the specification conflicts related to specification of actions when building ASD models:

Error Code	Explanation	Fix
RC54	The indicated state can not be reached following a path from the initial state of the state machine	Ensure that there is at least one transition to the indicated state from a non-floating state
RC55	Every rule case has to have at least one action	Ensure that all rule cases have at least one action
RC57	A rule case can not have more than one abstract action, i. e. Illegal, Disabled, Blocked or NoOp	Ensure that the rule case has one and only one abstract action, or none at all
RC58	Each rule case which has actions different from Illegal, Disabled, and Blocked must have a target state	Ensure that the rule case has a target state, or it has Illegal, Disabled, or Blocked as action
RC59	Each rule case which has Illegal, Disabled or Blocked as action must not have a target state	Ensure that the rule case has no target state or does not have Illegal, Disabled, or Blocked as action
RC60	Only Blocked can be specified as action in the indicated rule case. See "State types in a design model" for the rules about allowed actions in various state types.	Press Shift+F8 to automatically fix the conflict or change the action to Blocked
RC61	Blocked can not be specified as action in the indicated rule case. See "State types in a design model" for the rules about allowed actions in various state types.	Press Shift+F8 to automatically fix the conflict or change the action to Illegal or any other valid action
RC62	A rule case can not have more than one valued call event in its actions, because after a valued call, the component needs to go to a synchronous return state to look at the return value	Ensure that there is at most one valued call event per rule case
RC63	A rule case can not have more than one reply event in its actions	Ensure that there is at most one reply event per rule case
RC64	A valued call event must be the last event in the actions of a rule case, because after a valued call, the component needs to go to a synchronous return state to look at the return value	Ensure that the valued call event is the last action in the rule case
RC65	A transfer reply event must be the last action in the actions of a rule case	Ensure that the transfer reply event is the last action in the actions of the rule case
RC66	Each rule case which has a transfer reply event as action must have the initial state of the sub machine as target state	Ensure that the rule case which has a transfer reply event as action has the initial state of the sub machine as target state
RC67	Each rule case that has the initial state of a sub machine as target state, must have a transfer reply event as last action	Ensure that all rule cases which have the initial state of a sub machine as target state, have a transfer reply event as last action
RC68	A rule case with a modelling event as trigger may not have Illegal as action	Press Shift+F8 to automatically fix the conflict or change the action to Disabled or any other valid response
RC69	A rule case triggered by modelling event has no effect when NoOp is specified as action, no state variable update is specified and there is no state change	Use "Disabled" to indicate that it is your intention that the modelling event should have no effect. Enter an Action and/or a State variable update and/or a different Target State in case the null-effect was a mistake.
RC70	The indicated reply event must belong to the same interface as the indicated trigger event	Ensure that the specified reply event belongs to the same interface as the indicated trigger
RC71	A "void" call event should not be followed by a reply event and/or a valued call event should not be followed by a void reply (i.e. "VoidReply")	Ensure that the valued_call_event-reply_event and void_call_event-void_reply pairs are correctly specified in rule cases
RC72	Transition to this state is preceded by a valued call event on one transition while it is preceded by a void call event on another transition. Consequently, it is unclear if this should be a "Synchronous return state" or a "Normal state". See "State types in a design model" for details about state types.	Ensure that all transitions to the state have either void or valued call events as the last event
RC73	It is not allowed to subscribe to a non broadcasting notification interface	Remove the respective Subscribe from the sequence of actions or make the notification interface broadcasting
RC74	It is not allowed to unsubscribe from a non broadcasting notification interface	Remove the respective Unsubscribe from the sequence of actions or make the notification interface broadcasting
RC75	You are attempting to Subscribe, or Unsubscribe, to an interface which is not declared as a used interface	Remove the Subscribe or Unsubscribe action or declare the respective interface as a used interface
RC101	No guard should be specified if the rule case has Blocked as action	Specify different action(s) or remove the guard
RC102	No state variable updates should be specified if the rule case has Blocked as action	Specify different action(s) or remove the state variable updates
RC103	A rule with a rule case in which Blocked is specified as action should have no other rule cases	Specify different action(s) or remove the other rule cases from the rule
RC114	Disabled is only allowed for rule cases having a modelling event as trigger	Specify different action(s)
RC116	No state variable updates should be specified if the rule case has Disabled as action	Specify different action(s) or remove the state variable updates
RC121	In a *Single-threaded* model it is not possible to have a reply event on a modelling event trigger because this always results in deadlock	Change the actions or the execution model

Fixing state variable and guard related conflicts

The following table shows the specification conflicts related to specification of state variables and/or guards when building ASD models:

Error Code	Explanation	Fix
RC87	The indicated state variable is not declared in conformance with the rules of defining a state variable in ASD modelling	Ensure that the specified state variable is declared correctly. See also "State Variables".
RC88	Used service reference state variables should have a cardinality specified in the Cardinality column	Ensure that there is a cardinality >=1 specified for the used service reference state variable
RC89	Only variables of type Used Service Reference can be declared with a Cardinality	Empty the Cardinality field or change the type of the variable to Used Service Reference
RC90	There is no constraint specified for the specified state variable	Ensure that there is a constraint specified for the respective Used Service Reference state variable in the "Constraint" column
RC92	The initial value specified for the indicated state variable is syntactically incorrect	Ensure that you enter a syntactically correct initial value. This depends on the type of variable: o Boolean: true or false, o Integer: a number o Used Service Reference: one or more used service references. **Note:** When you want a sequence of used service references, i.e. a collection of used service references, as initial value, use the "+" operator to concatenate the respective used service references.
RC93	Each state variable must have an initial value within the specified range	Specify an initial value within the specified range
RC94	There is more than one rule case without guard(s) for the specified trigger in the specified state	Ensure that there is no more than one rule case without guards for the specified trigger in the specified state
RC95	For the specified trigger in the specified state there is at least one rule case without guard and one rule case with guard. This also occurs if one rule case has "otherwise" as guard and there is at least one rule case of that rule that has no guards.	Fill in a guard on the guard-free rule case, typically "otherwise", or delete a rule case, in the indicated state, which has the indicated trigger
RC96	Design models need to be deterministic and therefore at most one rule case per rule can have the "otherwise" keyword as guard	Ensure that for a rule otherwise is specified only once
RC97	At least one rule case per rule has to not have "otherwise" as guard	Ensure that not all rule cases having the indicated trigger have "otherwise" as guard
RC98	The guard is syntactically incorrect	Ensure that the specified guard conforms to the specified rules. See also "Guards".
RC99	The state variable update is syntactically incorrect	Ensure that the specified state variable update conforms to the specified rules. See also "State Variable Updates".
RC100	Since in ASD multiple assignments are handled conform the simultaneous assignment semantics, there can be only one assignment to a specific state variable in a state variable update expression	Ensure that there are no multiple assignments to the same state variable in one state variable update expression
RC152	It is of no use to have a rule case with an empty guard and one with an "otherwise" guard, since an empty guard is equivalent to True, and therefore, the "otherwise" is always False. In other words, the "otherwise" rule case will never be executed.	Put something on the empty guard, or delete the "otherwise" rule case
RC162	The leftmost number in the range must be smaller than or equal to the rightmost number in the range, e.g. [0:2] is allowed but [2:0] is not.	Swap the numbers in the range.
RC175	You must give the list of values that your enumeration type can have.	Specify a comma-separated list of identifiers in the Constraint field. See also "Type Definitions".
RC181	Type definitions must have a type specified in the Type column.	Select a type. See also "Type Definitions".
RC182	The leftmost number in the range must be smaller than or equal to the rightmost number in the range, e.g. [0:2] is allowed but [2:0] is not.	Swap the numbers in the range. See also "Type Definitions".

Fixing code generation/verification conflicts

The following table shows the conflicts reported by the ASD server upon verification or code generation:

Error Code	Explanation	Fix
CGE03	For the indicated language(s) the following service reference constructs are not allowed: o State variables of used service reference type o Using a used service reference in a guard or state update expression o Using a used service reference expression in an action sequence	Make sure that used service references are not used for state variables, in guard or state update expressions or in actions.
CGE04	For the indicated language(s) it is not allowed to use service references as construction parameters.	Remove the construction parameter(s) of service reference type in the Model Properties dialog.
CGE05	For the indicated language(s) it is not allowed to use service reference arguments in the Construction field.	Remove the service reference argument(s) from the Construction field in the Service References table.
CGE06	For the indicated language(s) the cardinality of the used service must be defined at design time, it cannot be unbounded (*).	Update the cardinality of a service reference with a value from the valid range [1..128].
CGE07	For the indicated language(s) only the Single-threaded execution model is supported	Change the Execution model setting in the Model Properties dialog to Single-threaded (in the interface model(s)).
CGE09	For the indicated language(s) serialisation is not supported.	Uncheck the Serialisable option in the Model Properties dialog (in the interface model(s)).
CGE10	For the indicated language(s) serialisation is only supported in combination with the Single-threaded execution model.	Either uncheck the Serialisable option in the Model Properties dialog, or change the Execution model setting in the ModelProperties dialog to Multi-threaded (in the interface model(s)).
CGE11	For the indicated language(s) when the implemented service has the serialisetion optin set, all used services must also have the serialisation option set.	Either uncheck the Serialisable option in the Model Properties dialog for the implemented service, or check the Serialisable option in the ModelProperties dialog for all used services.
CGE12	For the indicated language(s) the package name in the Namespace prefix field is invalid.	The Namespace prefix can consist of multiple nested scopes, e.g. "com.mycompany.myproduct". In other words, the namespace is a (possibly empty) dot-separated list of identifiers.
CGE13	For the indicated language(s) the syntax of the Include/Import fields is not correct	The content of the "Include/import" fields in the Model Properties dialog should be zero or more include/import statements, one per line.
CGE14	For the indicated language(s) the syntax of the Include/Import fields is not correct	The content of the "Include/import" fields in the Model Properties dialog should be zero or more include/import statements, one per line.
CGE16	For the indicated language(s) the stub generator does not support the proxy option	Uncheck the option "Generate a proxy class for each interface" in the Generate Stub dialog.
CGE17	For the indicated language(s) the combination of a Multi-threaded implemented service with Single-threaded used services is not supported.	Either change the Execution model setting in Model Properties dialog of the implemented service to Single-threaded, or change the Execution model setting in Model Properties dialog of the used services to Multi-threaded.
CGE18	For the indicated language(s) the combination of a Single-threaded implemented service with Multi-threaded used services is not supported.	Either change the Execution model setting in Model Properties dialog of the implemented service to Multi-threaded, or change the Execution model setting in Model Properties dialog of the used services to Single-threaded.
CGE20	The namespace "asd.builtin" is reserved for ASD specific features such as the ITimer.	Make sure the combination of namespace prefix and namespace is not equal to "asd.builtin".
CGE21	Each interface model must have a unique modelname	Change the model name of either the implemented or used interface model. Note that the full modelname is constructed as the combination of the modelname and the namespace, and for code generation also namespace prefix and the property "use service name in fully qualified names" for the selected language.
CGE22	Each interface model must have a unique modelname	Change the model name of one the used interface models. Note that the full modelname is constructed as the combination of the modelname and the namespace, and for code generation also namespace prefix and the property "use service name in fully qualified names" for the selected language.

Verification

These are the steps to be followed when you want to verify an ASD model:

1. Prepare the model for verification
2. Start verification
3. Analyze and fix the reported errors

Note: If one of your checks ends up in "Queue full" see "Singleton Notification Events" and/or "Yoking Notification Events" for possible solutions.

Code Generation

To generate code and integrate the code into your software system follow these steps:

1. Prepare the ASD model for code generation
2. Generate code from an ASD model
3. You can generate *stub* code from in interface model to create foreign components that are fully integrated with the ASD components
4. Download the ASD Runtime for the language and version that matches your project

Preparing the ASD model for code generation

A conflict-free model is ready for code generation. Additionally, there are several settings you can use to tweak code generation and model verification to your requirements. For example, you might want to:

- change the component type,
- change the execution model,
- change the target language,
- change the code generator version,
- use namespaces
- specify construction parameters
- specify output path and tracing information
- import/include user defined types
- customize the code

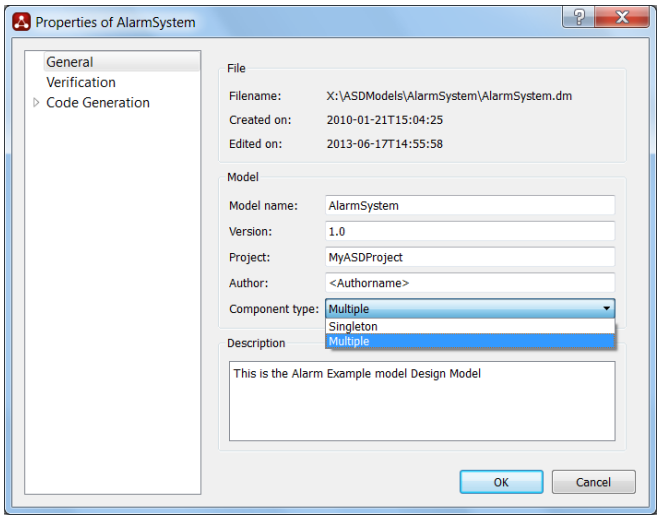
Specifying the component type

In ASD you can choose between two types of components: *Singleton* or *Multiple*.

When selecting *Singleton* for a component, a single instance of the respective component will be created and this instance will be accessed by all components that use the *Singleton* component. In case of *Multiple*, each component that uses the respective component will create and use its own instance(s) of the *Multiple* component.

To specify the desired component type you have to open the model properties dialog for your design model by selecting the model in the "Model Explorer", right-click and select the "Properties" item in the context menu.

The following dialog is shown:



The Properties dialog for a design model

In case you want two ASD components to share an instance, without using a Singleton component, you can pass a used component instance to an ASD component as a parameter. See "Construction parameters" for details about passing component instances.

Specifying the execution model

In ASD you can choose between two execution models for your components: *Multi-threaded* or *Single-threaded*.

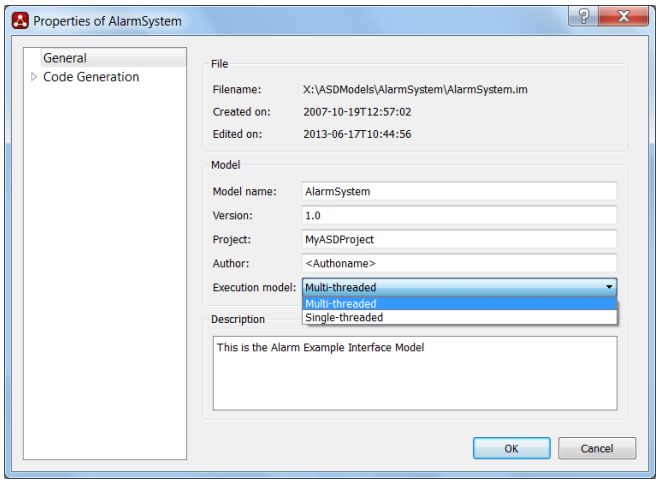
When selecting *Multi-threaded* for a component, this component will have its own separate thread for handling notifications, if any, from servers. In case no notifications are defined the separate thread is not created. In case of *Single-threaded*, this thread will not be created; only one execution thread will be active during the execution of a system of *Single-threaded* components. See the "ASD Runtime Guide" for details about the runtime execution semantics for the two execution models.

The following rules apply to a *Single-threaded* ASD component:

1. The component type of the design model which implements the interface of your component (see "Specifying the component type") is *Multiple*.
   Note: Specify *Multiple* as component type when you generate stub code for your component (see "Generating stub code from an ASD interface model").
2. No broadcasting notification interfaces are defined in your component.
3. All available interfaces are specified as used interfaces when you specify your component as used service in a design model.
4. A Multi-threaded component can be specified as used component only if it has no notification interfaces and no state change occurs spontaneously.
5. No yoking thresholds are specified for the defined notification events.
6. The ASD Timer is not a used service in the design model in which you use your component.
7. If a tree of components uses the *Single-threaded* execution model, then the entire tree can only be accessed by one thread at a time.

To specify the desired execution model you have to open the Properties dialog for your interface model by selecting the model in the "Model Explorer", right-click and select the "Properties" item in the context menu.

The following dialog is shown:



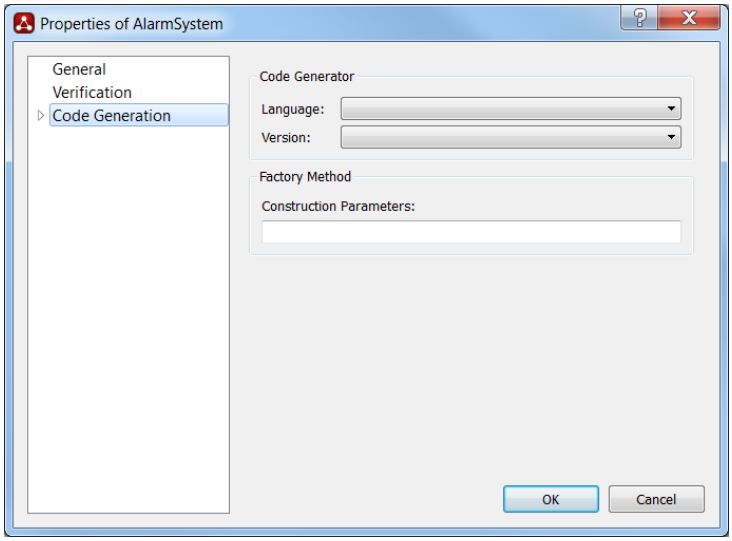
The Properties dialog for an interface model

Specifying a target language and the code generator version

Code generation language and version properties are now captured in the model properties section and are stored in the ASD model file. This allows you to specify the desired target language and code generator version for each model when information is missing or not according to your expectations. The target language and code generator version for the open model are also indicated in the status bar of the ASD:Suite. This information is required every time when you verify the model and generate code.

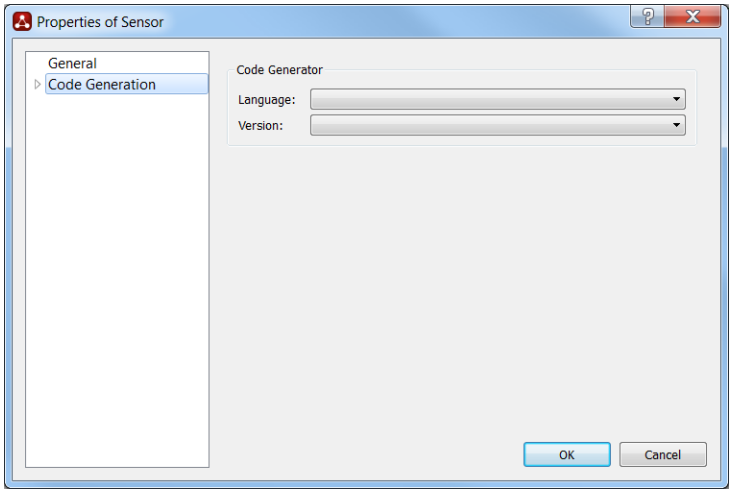
You can access the code generation "Properties" dialog of your model by selecting the interface model in the "Model Explorer", right-click, select the "Properties" item in the context menu and choose the "Code Generation" tab.

The following dialog appears on your screen if the ASD model is a design model:



The code generation "Properties" dialog for a design model

In case the ASD model is an interface model the code generation "Properties" dialog appears as follows:



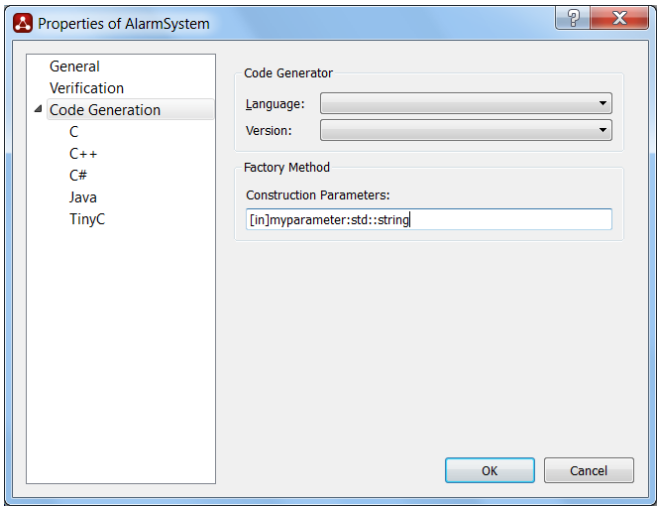
The code generation "Properties" dialog for an interface model

With the ASD:Edit command line tool you can set the code generation language and version for all ASD models in your project in one go.

Defining construction parameters

Components can have construction parameters - parameters that are passed to the component when it is instantiated. Construction parameters can be regular parameters like integers and strings, or they can be service parameters - instances of other components. Regular parameters can be passed on further to hand-written components. Service parameters can be used to pass used services to a component (this is known as dependency injection).

The construction parameters are defined in the "Construction parameters" field of the code generation settings:



Settings for code generation in an ASD design model

This is the syntax for construction parameter definition:

1. For parameters of user defined types: "[in]name:std::string"
   Note: You can use any type you like that the programming language allows. For most languages, you will need to include/import the type using the "Include/import (declaration)" field of the code generator settings. For more detail see "Referencing user defined types"
2. For a single injected service: "[in]siren: service(ISiren)"
3. For a vector of injected services: "[in]sensors: service[](ISensor)"

Note:

- When you want to define multiple construction parameters you specify them in a comma-separated list
- The border of the "Construction parameters" field is coloured in red if the syntax is incorrect

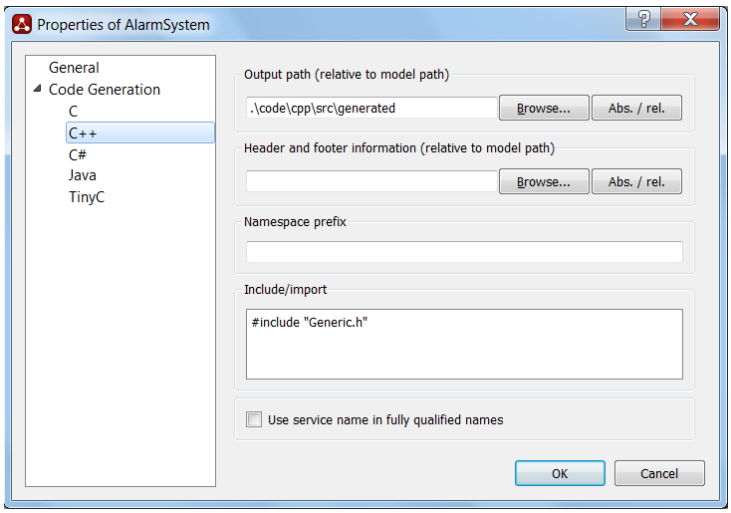
See "Construction Parameters" for details.

Specifying the output path and attribute code with tracing information

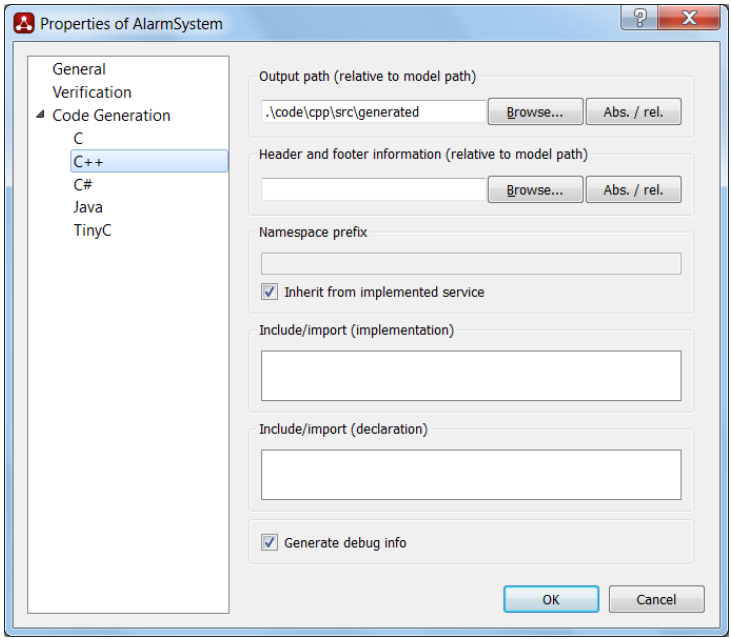
When you want to specify an output path for the generated files or you want to include tracing information in your generated code, fill in the respective data in the code generation "Properties" dialog of your ASD model.

You can access the code generation "Properties" dialog of your model by selecting the ASD model in the "Model Explorer", right-click, select the "Properties" in the context menu and choose your target language under the "Code Generation".

The following figures show the code generation "Properties" dialog for target language C++ in case the ASD model is an interface model, or a design model, respectively:



The code generation properties for target language C++ in an interface model



The code generation properties for target language C++ in a design model

Check-mark the "Generate debug info" checkbox to attribute the generated code with tracing information. The tracing information contains the component name, the state name and the trigger name. This information is reported every time a trigger function is entered, prefixed with "-->" and every time this trigger function is exited, prefixed with "<--". The information is passed to a language specific tracing mechanism:

- For C++, this is ASD_TRACE, a macro defined in the ASD Runtime header file trace.h. There is a default implementation using std::cout, but this can be customized by you.
- For C#, the generated code uses the .NET System.Diagnostics.Trace facility. This can also be customised by you within the limits of .NET. To enable tracing in a .NET application, the code must be compiled with the TRACE define set, i.e. -DTRACE and somewhere a listener must be registered to pass the information to you:

```
System.Diagnostics.Trace.Listeners.Add(new System.Diagnostics.ConsoleTraceListener);

System.Diagnostics.Trace.AutoFlush = true;
```

- For C the tracing is limited to a single string literal message. This message is somewhat customisable through redefining the pre-processor macro responsible for compiling the message. The default implementation gathers function, file and line number.
- For Java, by default the DiagnosticsDefaultTraceHandler is used. This class is part of the ASD Runtime for Java and contains a println to System.out. In case you want to customize the tracing you can override this class by a custom version.

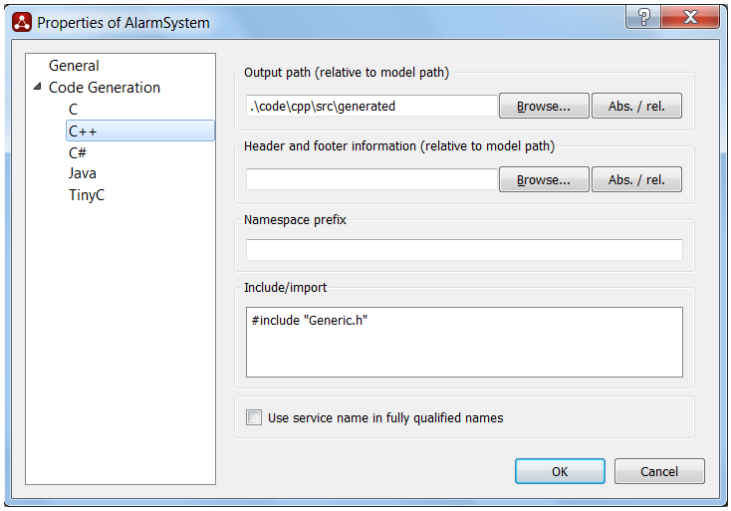
Ensuring correct referencing of user defined types

When you specified parameters of user defined types in your ASD component you need to ensure that the files where you defined the respective types are referenced in the generated code.

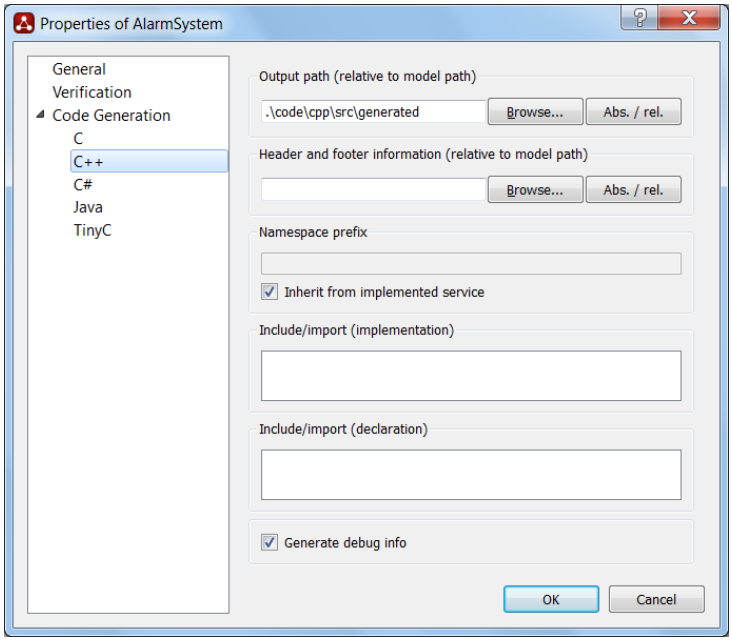
The ModelBuilder provides an "Include/import" field in the code generation "Properties" dialog of your ASD model to facilitate the specification of the required referencing statements. You can access the code generation "Properties" dialog of your model by selecting the ASD model in the "Model Explorer", right-click, select the "Properties" in the context menu and choose your target language under the "Code Generation".

> **Note:** Any data you fill in the "Include/import" field for any other target language than C, C++ or Java is going to be ignored during code generation.

The following figures show the code generation "Properties" dialog for target language C++ in case the ASD model is an interface model, or a design model, respectively:



The code generation properties for target language C++ in an interface model



The code generation properties for target language C++ in a design model

The content of the "Include/import" fields for C and C++ should be zero or more include statements, one per line. An include statement is one of the following:

- #include "FileName", or
- #include <FileName>

where, FileName is the name of the header file containing definitions of user defined types.

For Java the content of the "Include/import" fields should be zero or more import statements, one per line. An import statement looks like:

```
import com.verum.<DirName>.*;
```

where DirName is the name of the directory where the user defined classes are.

Note:

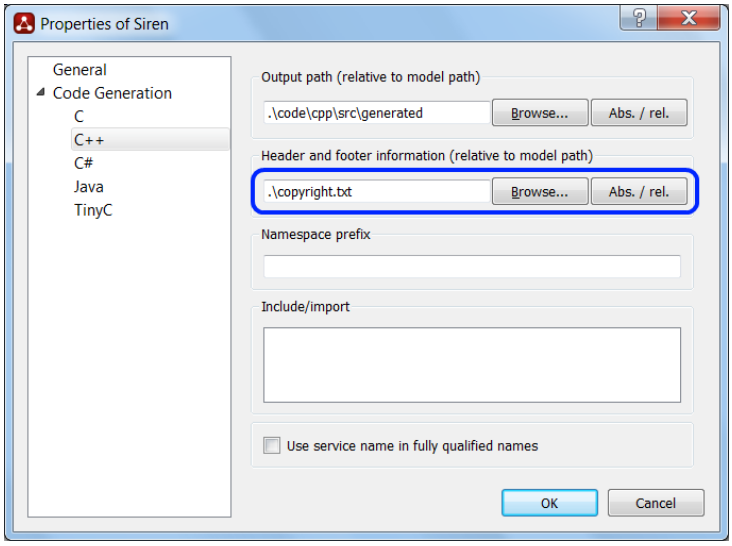
- FileName and DirName are strings containing only alphanumerical characters.
- The ASD:Suite performs a series of syntax checks on the content of the "Include/import" fields and will report errors if syntax is incorrect.
- Whenever user defined types are used in the component implementation, specify the include/import statements in the "Include/import (implementation)" field.
- Whenever user defined types are used in specifying component construction parameters, specify the include/import statements in the "Include/import (declaration)" field.

Specifying path to user provided text for code customization

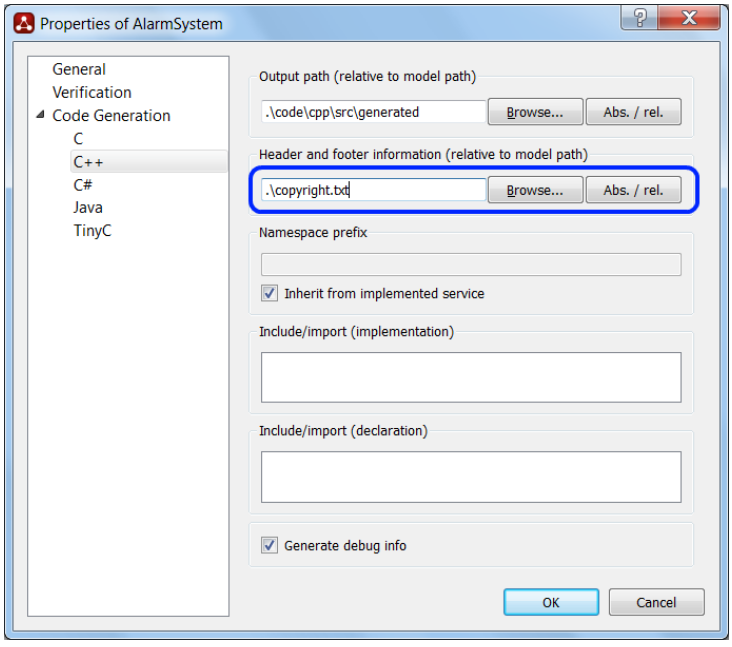
The ModelBuilder enables insertion of user provided text (like copyright text) in the generated source code. The text file containing this text is specified for each ASD model in the text field labeled "Include file" in the code generation "Properties" dialog for each target language.

You can access the code generation "Properties" dialog of your model by selecting the ASD model in the "Model Explorer", right-click, select "Properties" in the context menu and choose your target language under "Code Generation".

The following figures show the code generation "Properties" dialog for target language C++ in case the ASD model is an interface model, or a design model, respectively:



The code generation properties for target language C++ in an interface model



The code generation properties for target language C++ in a design model

The following list contains the rules for the text which you can specify as user provided text:

- The specified text file can contain a mixture of **directives** and plain text lines in any order. This enables it to serve as a "template" controlling the generated output source file contents.
  - Plain text lines are simply copied through to the generated output file without modification.
- Zero or more **<include>** directives can be specified in any order anywhere in the text file with the effect that contents of the specified text files are copied into the generated file.
  - Specify one **<include>** directive per line
- If the specified file in an **<include>** directive can not be opened for whatever reason when generating source code, the directive is completely ignored and has no effect
- Zero or one **<generate/>** directive can be specified designating the point at which the generated code is inserted into the output file. If omitted, the generated code is added to the output file after the last line in the specified text file. If multiple **<generate/>** directives are present in the file, only the first one is processed; the others are ignored as though they were not present.
  - Both DOS and UNIX style line-endings are allowed.

- Both the Copyright file and all include files must be 8-bit ASCII encoded.

Example:

- text file with no directives
- text file with an **<include>** directive
- text file with a **<generate/>** directive

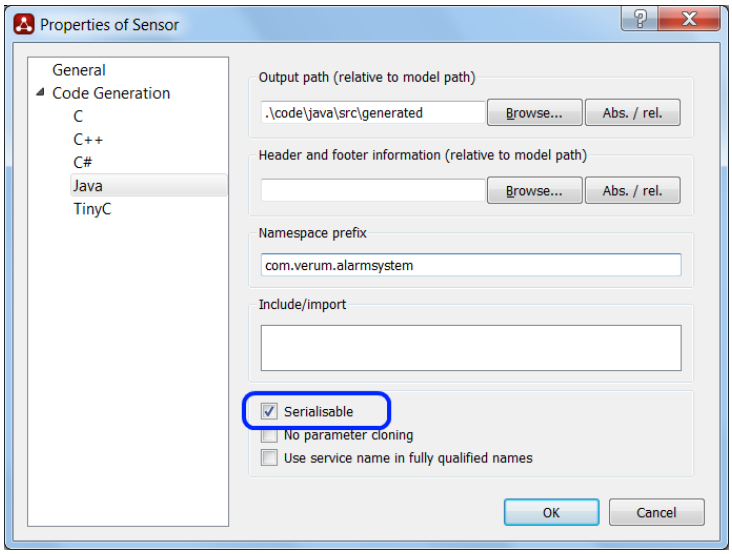
Serialising ASD components

Serialisation is a mechanism available in Java to store a Java object in a persistent form on disk, for example, to send over the network or for use in Remote Method Invocation.

In ASD, the serialisation is achieved via implementing Externalizable interface, wherein two methods are introduced: writeExternal and readExternal. These methods are visible in the Java generated code for the design model.

The reason to use the Externalisable over the Serialisable interface, is that Externalisable provides complete control to the user on the marshalling and un-marshalling process, however with the drawback of reduced flexibility. With regards to versioning of Serialisable objects, only the top level component is versioned.

In order to ensure serialisation you have to select the Serialisable check-box in the Model Properties dialog window under the "Code Generation->Java" tab. The following figure shows the Serialisable check-box in the interface model for the Sensor service:



The Serialisable check-box

Specifying the namespace (or package)

The namespace in which your interface or component is put by the code generator, depends on a number of things:

- The namespace that you type as part of the model name: instead of just naming a model e.g. Sensor, you can name it myAlarmSystem.Sensor. This puts the generated component or interface in a namespace "myAlarmSystem"
- The namespace prefix that you can specify in the code generator settings. Since e.g. Java package names tend to be both deeply nested and language-specific, it is often handy to specify most of the package name in the code generator settings.
- The setting of the "Use service names in fully qualified names" property that you can specify in the code generator settings of an interface model. This property allows different services to have the same interface names. When the the checkbox for this option is checked, all fully qualified names will contain the service name. For example the fully qualified name for an application interface call event the fully qualified name looks like:

  ```
  namespace.servicename.interfacename.calleventname
  ```

  This "Use service names in fully qualified names" property is by default set to true.
  Note: To retain backward compatibility of the generated code, this "use service names in fully qualified names" property is set to false during upgrade of older models.

You can access the code generation "Properties" dialog of your model by selecting the model in the "Model Explorer", right-click, select the "Properties" item in the context menu and choose the "Code Generation" tab.

The namespace can consist of multiple nested scopes, e.g. "com.mycompany.myproduct". In other words, the namespace is a (possibly empty) dot-separated list of identifiers.

When referencing services (e.g. in the construction field of the Service References tab, or in Construction Parameters) the reference should include the namespace (and prefix), if any.

Note: when generating code in Java, the namespace is translated into the directory-structure. For all other languages, the generated component and interface filenames (component and interface files) contain the namespace and namespace prefix.

Generating code from an ASD model

Note:

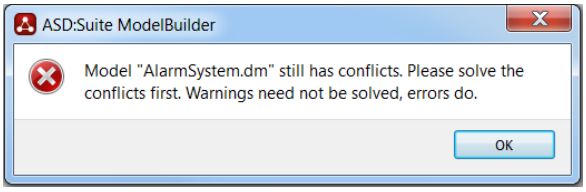
- Before generating code from a design model or an interface model with the ASD:Suite you might need to perform (some of the) items described in "Preparing the ASD model for code generation".
- When C is the target language, the ASD:Suite will use the value specified as "Size" for the event queue in constructing the event queue. **Note:** When you decide to decrease the value of the queue size for purposes of code generation you must ensure that the verification is still successful.

Code generation using the ModelBuilder can be done in one of the following ways:

1. Press F7 or Ctrl+F7, or
2. Select one of the following menu items: "Code Generation -> Generate Code" or "Code Generation -> Generate All Code", or
3. Select the design model or interface model in the "Model Explorer" window, right-click and select the "Generate Code" item in the context menu.

Note:

- Code generation language and version properties are now captured in the model properties section and are stored in the ASD model file. This allows you to specify the desired target language and code generator version for each model when information is missing or not according to your expectations. The target language and code generator version for the open model are also indicated in the status bar of the ModelBuilder.
- If you want to generate code for the selected ASD model using a different target language and/or code generator version than the ones specified in the model properties you can use the "Code Generation -> Generate Code With..." menu item or press Shift+F7 which opens the "Generate Code for <model-name>" dialog in which you have to specify the desired data.
- The following error message informs you that no source code can be generated from an ASD model with specification inconsistencies (conflicts):



Error message which prevents code generation from a model with conflicts

- The code will be generated in the specified directories. See "Specifying the output path and attribute code with tracing information" for details.
- The progress of the code generation can be followed in the "Output Window".
- Download the ASD Runtime and ensure correct referencing of user defined types before compilation and execution of the generated code.

verum

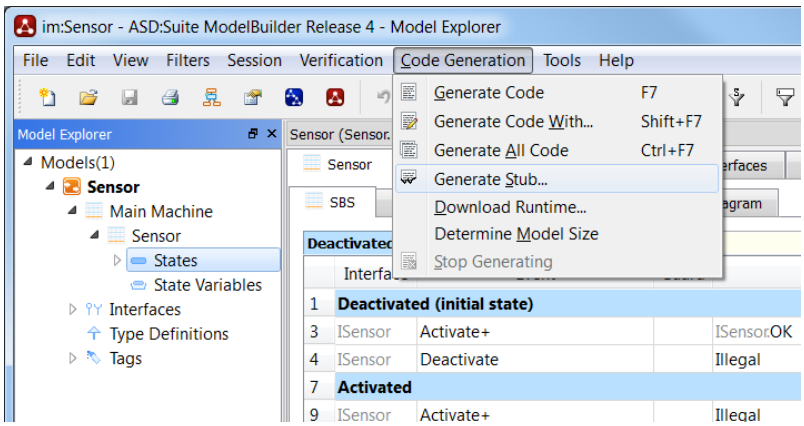
Home | Product | Technology | Resources | Training | Purchase | Company

Generating stub code from an ASD interface model

From an ASD interface model you can generate skeleton code which can serve as a starting point for implementing handwritten code for a client of an ASD component or for a Foreign component. The skeleton stub code is compile-able such that an executable can be created.

The main purpose of the generated stub code is to give you a head start in developing handwritten code, relieving you from the burden to write all the infrastructural code and clearly point out where you can add the custom user code.

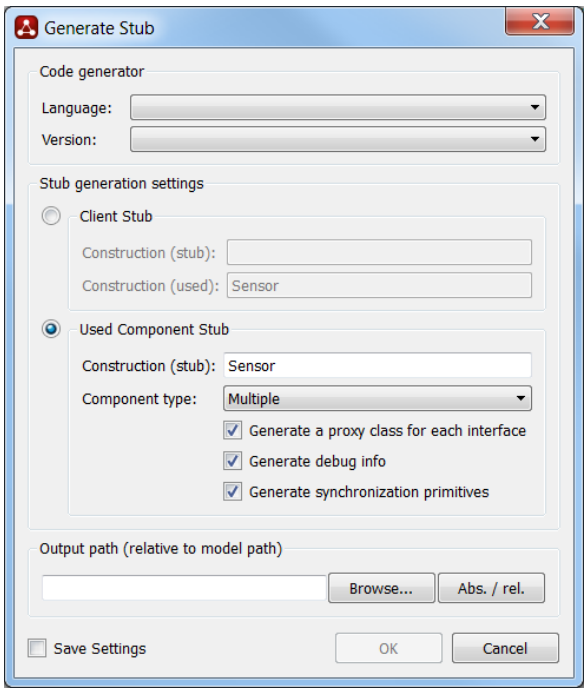
See the following figure for a graphical representation of the menu item used to initiate stub code generation:



Menu item to initiate stub code generation

Note: The stub code is generated only when the interface model is free of rule check errors (conflicts)

To complete the stub code generation, fill in the missing information in the following dialog and press OK:



The Generate Stub dialog

Note:

- For a Client Stub
  - the data in the "Construction (stub)" field denotes the name of the client and complies to the following syntax:

    myNamespace.MyComponentName(construction-parameters)

    , where the syntax for defining construction parameters is the same as the one presented in the "Defining construction parameters"section. The namespace is optional and may be left out.
  - the data in the "Construction (used)" field denotes the name of the ASD component and complies to the following syntax:

    myNamespace.MyComponentName(construction-parameters)

    , where the syntax for specifying construction arguments is the same as the one presented in the "Specifying construction arguments"section. The namespace is optional and may be left out.
- For a Used Component Stub
  - the data in the "Construction (stub)" field denotes both the name for the component and the signature for its GetInstance() method and complies to the following syntax:

myNamespace.MyComponentName(construction-parameters)

, where the syntax for defining construction parameters is the same as the one presented in the "Defining construction parameters"section. The namespace is optional and may be left out.

Example:
alarmdemo.MyAlarmSystem([in]housename:std::string, [in]siren: service(ISiren), [in]sensors: service[](ISensor))

results in a component named MyAlarmSystemComponent in the namespace alarmdemo with a GetInstance method that accepts (in this order) a string, a component that implements ISiren, and a vector of components that implement ISensor.

- you can specify (in the "Component type" field) the type of the component for which you generate stub code. You can choose between *Multiple* and *Singleton*. The default is *Multiple*.
- the checkboxes under the "Component type" field determine if trace statements, synchronization primitives, or a proxy classes (one per interface) are generated in the stub code. By default they are deselected.
  - **proxy classes** : turning this option on causes every interface to be generated in a separate proxy class. This is useful when handwritten components have many interfaces and events. It is particularly useful when several interfaces have events with the same name, reducing the probability of name clashes.
  - **debug info** : turning this option on causes trace statements to be inserted upon entry and exit of every method. This trace can provide to the developer useful information while debugging the system.
  - **synchronization primitives** : turning this option on causes all the methods to be thread-safe. This is particularly useful when making a foreign component which is accessible by multiple clients at the same time while data integrity within this foreign component must be guaranteed.
- The border of the "Construction (stub)" and "Construction (used)" fields turns red when the syntax is not correct.

To prevent that already existing handwritten files are overwritten accidentally, the following naming conventions are used for the various target languages and for the various stub code type for which skeleton code can be generated:

- For Client Stub code:
  - **C++** : <specified_output_path>\<specified_client_name>.h_tmpl,
    <specified_output_path>\<specified_client_name>.cpp_tmpl
  - **C#** : <specified_output_path>\<specified_client_name>.cs_tmpl
  - **C and TinyC** : <specified_output_path>\<specified_client_name>.h_tmpl,
    <specified_output_path>\<specified_client_name>.c_tmpl
  - **Java** : <specified_output_path>\
    <directory_structure_determined_by_specified_namespace>\
    <specified_client_name>.java_tmpl

  , where <specified_output_path> is the value filled in the "Output path" field and <specified_client_name> is the name of the component as filled in the "Construction (stub)" field.
- For Used Component Stub code:
  - **C++** : <specified_output_path>\<specified_component_name>Component.h_tmpl,
    <specified_output_path>\<specified_component_name>Component.cpp_tmpl
  - **C#** : <specified_output_path>\<specified_component_name>Component.cs_tmpl
  - **C and TinyC** : <specified_output_path>\<specified_component_name>Component.h,
    <specified_output_path>\<specified_component_name>ComponentImpl.h_tmpl,
    <specified_output_path>\<specified_component_name>Component.c_tmpl
  - **Java** : <specified_output_path>\
    <directory_structure_determined_by_specified_namespace>\
    <specified_component_name>Component.java_tmpl

  , where <specified_output_path> is the value filled in the "Output path" field and <specified_component_name> is the name of the component as filled in the "Construction (stub)" field.

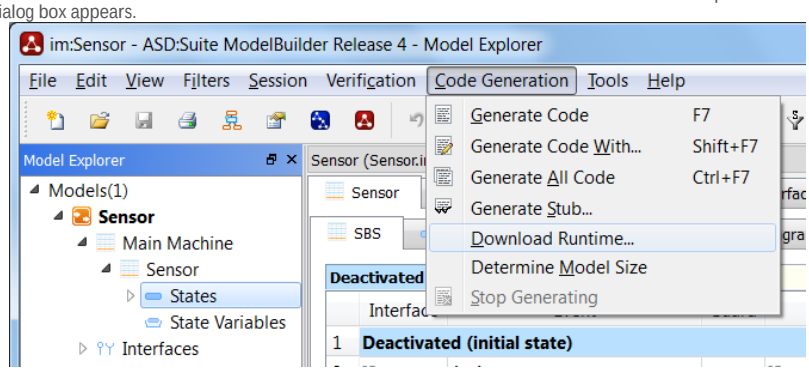
After the skeleton code is generated, rename the file(s) to the correct file name by removing the "_tmpl" postfix.

Downloading the ASD Runtime

The ASD Runtime is a software package distributed as part of the ASD:Suite, which enables source code generated with the ASD:Suite to be executed on a specific execution platform. Additionally, the ASD Runtime package implements the semantics required to ensure the compatibility between the generated code and the verified ASD models from which the code was generated.

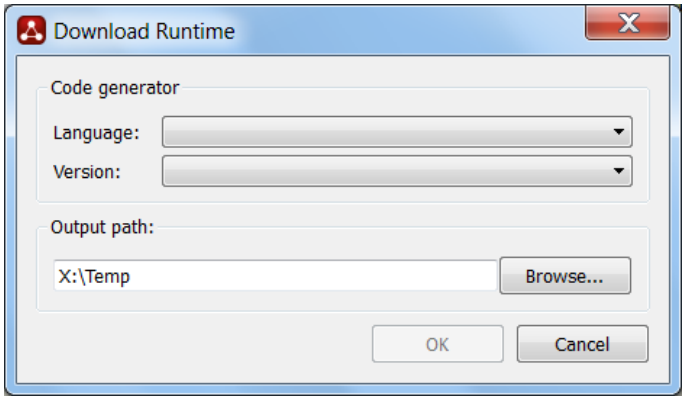
These are the steps to download the ASD Runtime for a specific target language using the ModelBuilder:

1. Select the "Code Generation -> Download Runtime..." menu item to initiate the ASD Runtime download process. A Download Runtime dialog box appears.



Menu item to download the ASD Runtime

- Choose the ASD Runtime language and version number from the dropdown lists.
- Select the output path where to store the ASD Runtime files.
- Select the OK button to begin the ASD Runtime download.



The "Download Runtime" dialog

When finished, a list of the ASD Runtime files that have been downloaded appears in the ASD:Suite "Output Window".

The download is complete when the "==== Finished successfully ====" message appears in the "Output Window".

See the "ASD Runtime User Guide" for details about the ASD Runtime.

Session Management

To use the ModelBuilder, you must be connected to the ASD:Server, and be registered as an authorized user. If you are not authorized, you are presented with the Login dialog. The only things you can do in this dialog are logging in, saving your models, and exiting the ModelBuilder.

- For making a connection, see "Logging In".
- For easily switching between different user or server settings, see "Saving Connection Settings".
- Optimising for long-latency connections is described in "Advanced Connection Settings".

When you start the ModelBuilder, you can already start to use it while the ModelBuilder automatically seeks connection with the ASD:Server. If making the connection fails, making the connection takes longer than a minute, you are presented with the Login dialog until you are connected and logged in.

Grace Period

Once you are logged in, you can view and edit models. If the connection is interrupted, you can still continue to work for a small period of time called the "Grace Period". The duration of the grace period is set by Verum on a per-customer basis. Note that your grace period may be zero. When the connection is interrupted, the ModelBuilder automatically tries to re-connect in the background. If the grace period expires without a connection having been established, the ModelBuilder will continue to re-connect, but you are presented with the Login dialog until a connection can be re-established.

Time-out Period

There is also a "Time-out Period" that causes the ModelBuilder to automatically log out after a period of inactivity. This is useful for users that have a pay-per-use license. Again this is a setting that Verum sets on a per-customer basis. Note that your time-out period could be set to 'infinite'. When the ASD:Suite ModelBuilder is logged out in this way, the Login dialog is presented and you can re-connect directly by clicking the OK button or by pressing <Enter>. Note that the server connection is kept, to facilitate quickly logging in again. You can see this in the status bar.

Authorized vs. Connected

In the status bar, you can see two statuses related to your session: one usually says "Authorized" and the other usually says "Connected". They are two separate statuses, because you can be authorized to use the ModelBuilder without being connected (e. g. in the minute after you start the ModelBuilder, or during the Grace Period), and also you can be logged out while still having a server connection (e.g. after the Time-out Period expires). In other words, the left field is your authorization status (Authorized/ Logging in/Logging out/Logged out), and the right field is the connection status (Connecting/Connected/Disconnecting/ Disconnected/Reconnecting).

To see more information about the current status, double-click the statuses in the status bar to see a dialog with more information.

Logging In

The Login dialog allows you to do three things: logging in, saving your models, and exiting the ModelBuilder. As described in "Session Management", you need to be logged in to use the ModelBuilder.

To log in, you need to provide the following information:

- User credentials: your user name, password and certificate file (.pem file)
- ASD Server settings: indicates which ASD:Server to connect to
- Optionally: HTTP tunnelling settings, for using an HTTP proxy to avoid using port 443

All settings are described below. **Note**: most settings are hidden under the "More >>" button in the dialog.

User Credentials

This identifies you to the ASD:Server:

- Certificate: the path to the the .pem file that every customer gets from Verum
- E-mail: your e-mail address
- Password: your password

You can choose to save the password, or to clear a saved password. Note that the password is only saved in the registry after a successful logon. This is because the server is used to encrypt the password for storage.

In case you have forgotten your password, just follow the link 'Help! I can't remember my password'. This will show you a webpage where you can request a password-reset.

ASD Server settings

This defines the ASD:Server to connect to. For the vast majority of users, this should be left to:

- Server: asd.verum.com, or for trial-users: trial.verum.com
- Port: 443

HTTP Tunnelling

If your network infrastructure does not allow you to use port 443, you can use a proxy server to connect to the ASD:Server.

- Enable HTTP Tunnelling: check this to use the HTTP proxy
- Proxy: the hostname of the proxy server
- Port: the TCP port of the proxy server, usually either 80 or 8080
- Username: optionally, the user name for your proxy server
- Password: optionally, the password for your proxy server

Again, you can save and clear the password.

Saving Connection Settings

If you regularly need to change user settings or HTTP tunnelling settings, it is handy to save and load them from the registry all at once.

To save login settings, use them to log into the ASD:Server. Once you are logged in, go to the Session Menu and choose "Save Settings". Type a name for your settings and click OK.

To use saved login settings, click the "Retrieve..." button in the Login dialog. You get a list of the names of all saved settings. Choose the settings you want and click OK. The settings are now filled in into the Login dialog. Click the "Log in" button to use the settings to log in.

Deleting login settings is possible from the Session menu as well.

Advanced Connection Settings

If you are geographically far removed from the ASD:Server or you happen to have a slow or low bandwidth internet connection, you may want to adjust some advanced settings to optimize your connection. In the Login dialog, use the "More >>" button to show the Advanced Settings section at the bottom. This section contains the following values:

- **Network timeout**:
  The time to wait for a connection attempt to succeed or a network read/write to complete. If this is too short, making a connection fails often, or uploading/downloading a file will fail often. If you experience this, increase the time-out value. If this is set too long, interrupted network connections will be detected only after a long time.
- **Operation timeout**:
  The time to wait for an application-level network operation to complete. Note that for AsdVerify.exe this comprises an entire model verification.
- **Heartbeat**:
  The ModelBuilder makes a call every so many seconds to keep the connection alive. This is because some proxy servers close the connection after a given amount of inactive time. If you are on a long-latency connection, these so-called heartbeat calls may start to take a considerable portion of time. In that case, increase the Heartbeat value to increase the period duration. Also, the heartbeat rate has an influence on when an interrupted network connection is detected. A slow heartbeat means that an interrupted connection goes unnoticed for a longer time.
- **Reconnect**:
  The number of seconds between two attempts to restore an interrupted connection. Setting this to a higher value causes the ModelBuilder to make less frequent re-connect attempts.

Note:
The ModelBuilder only notices that its connection is lost when it makes a call. Therefore, the Heartbeat value directly affects its ability to see if it still has a connection. Setting the Heartbeat to a very high value may cause the ModelBuilder to fail to notice an interrupted connection until you start to verify or generate code.

Command-line Tools

The ASD:Suite provides a number of tools you can run from the command line.

Overview:

- **AsdAuthorize.exe**: Use this for seeing the available and granted code generators for you, or to store your user credentials and server connection settings so you don't have to type them on the command line every time
- **AsdVerify.exe**: Use this to make sure all your models are verified correctly, or to check that generated code is generated from verified ASD models.
  - **AsdGenerate.exe**: Generates code from ASD models or downloads the ASD:Runtime.
  - **AsdRead.exe**: Query a model for its properties, like code generator version and language, implemented and used services etc.
  - **AsdEdit.exe**: Use this to set all kinds of properties, like code generator version and language, on multiple models at once.
- **AsdCompare.exe**: This is the command-line version of the ASD:Suite ModelCompare. It can detect semantic and non-semantic differences between two models.
  - **AsdConvert.exe**: This converts models to the current version of the ASD:Suite.

Note that these tools are also available for Linux and Solaris, albeit that the names of these tools are all in lower case and do not have the ".exe" extensions (i.e. "asdread" instead of "AsdRead.exe"). Also, the Linux installers create symlinks to these tools with shorter names to avoid you typing, e.g. "asdg" for "asdgenerate".

Of course, you can also call the graphical tools from the command-line. These do NOT support the Common Functionality described in the next section.

- **ASD:ModelBuilder**: called "ASD ModelBuilder.exe" on Windows, and "asdmb" or "asdmodelbuilder" on Linux.
- **ASD:ModelViewer**: called "ASD ModelViewer.exe" on Windows, and "asdmodelviewer" on Linux.
- **ASD:Compare**: called "CompareGui.exe" on Windows, and "asdcmp" or "asdcomparegui" on Linux.

Common functionality:

This section lists functionality common to all command-line tools.

Help

For all tools, type `--help` to see how a tool can be used. Next to that, tool version and revision information can be obtained with `--version` and `--revision`, respectively.

Files, directories, wildcards and recursion

Most tools accept multiple file names or directory names as input. It is possible to use wildcards in directory and file names, e.g.
`AsdEdit.exe -v 9.2.3 *.im`
This will apply the AsdEdit tool to all the interface models in the current directory.

The following wildcards can be used:

- ***** matches any sequence of zero or more characters except \
- **?** matches any single character
- **[]** matches any of the characters between the brackets

Next to wildcards, the tools also support recursing through subdirectories. To enable recursing, specify the `--recurse` option. **When recursing, the names on the command line are not file names but directory names**. Each given directory is visited recursively. To control which files are processed in those directories, specify a wildcard pattern using the `--name` option. The default pattern is `*.[id]m`, which matches all interface and design models.

For example, the following command processes all models in the current directory and all subdirectories:
`AsdEdit.exe -v 9.2.3 --recurse .`

And this example only processes the design models:
`AsdEdit.exe -v 9.2.3 --name "*.dm" --recurse .`

Note: Wildcard expansion for the command-line tools running on the Windows platform is performed by the command-line tools themselves. On the Linux platform the wildcard expansion is performed by the shell *before* passing on to the command-line tools. The net effect is the same, though.

Making a Connection

Most command-line tools need a connection with the ASD:Server to do its work. If you have already established connection with the ASD:Suite ModelBuilder and you saved your password, these settings will be used automatically. Otherwise, you can specify the connection settings on the command line as well. It is also possible to save your connection settings and password using ASD:Authorize (using the `--register` command).

For example:

```
AsdGenerate.exe -s "asd.verum.com" -p 443 -u "user@example.com" -c "C:\certificates\123456.
pem" --password "MyPassword1"
```

Note that if you omit the server password, you will be prompted for it.

If you need an HTTP proxy, add the following options:

```
--http-proxy "myproxy.com" --http-port 8080 --http-username "user" --http-password
"MyProxyPassword"
```

If the proxy does not need a username and password, it can be left empty.

Note that the command-line tools can also be asked to prompt for the proxy password (using the `--http-password-prompt` option). For details, see the --help option of any command-line tool.

Languages and Versions

Many tool commands need to be supplied with a code generator language and version (using `-l` and `-v` arguments, respectively). To check which languages and versions are granted to you, use ASD:Authorize. If a language and version is present inside the ASD model(s), you can omit the language and version arguments when generating code or verifying models. To set the language and version for all models, use ASD:Edit.

Tips

- It is recommended to start the command prompt using the "Start->All Programs->ASD Suite Release 4 V9.2.7->ASD Client Command Prompt" item.
- You can change the start-up folder for the ASD:Suite command-line tools started via the ASD Client Command Prompt by changing line 11 in the "ASDPrompt.bat" file which you can find in the folder specified during installation.
- It is recommended to add the full path to the folder where the ASD:Suite is installed to the PATH environment variable .
- To ensure that the latest version of the ASD:Suite command-line tools is used when you call them in the DOS command prompt, remove from the PATH environment variable all references to folders where other versions were installed.
- Even though you can specify the server connection settings as part of the command in the command prompt, it is recommended to run the ASD:Suite ModelBuilder once and save the connection settings, to store them as default values (or use ASD:Authorize to do the same from the command line).

ASD:Authorize

ASD:Authorize (AsdAuthorize.exe on Windows / asdauthorize on Linux), is a command-line tool for:

- Checking which code generator languages and versions are available.
- Checking which code generators you are authorized to use.
- Saving server connection options and user credentials to the registry, so you don't have to type them in every time.

The tool supports help, server connection options, and version and language as described in Command-line Tools.

Saving Server Connection Options and User Credentials

Most command-line tools need a server connection. To avoid having to type the server connection options on the command line every time, you can store them to the Windows Registry. You can do this simply by opening the ASD:ModelBuilder and making a connection; or by using this tool. The command is `--register`.

The exit code is 0 for succes and 1 for any failure.

Examples:

```
AsdAuthorize.exe --register -s asd.verum.com -p 443 -u somebody@example.com -c C:
\certificates\123456.pem --password "password1"
```

The command `--clean-registry` can be used to clean the passwords from the registry again.

Code Generator Versions and Languages

You can request the available languages, available versions, authorized languages, and authorized versions. Dependent on the `--verbosity` option, you get progress information and/or the resulting list of languages or versions.

When you request a list of languages, you can supply a version with the `-v` option to only display languages for that version. Conversely, when you request a list of versions, you can supply a language with the `-l` option to only display versions for that language.

The exit code is 0 for succes and 1 for any failure.

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Get a list of available languages:

```
AsdAuthorize.exe --available-languages
```

Get a list of available versions:

```
AsdAuthorize.exe --available-codegenerators
```

Get a list of languages you are authorized to use:

```
AsdAuthorize.exe --authorized-languages
```

Get a list of versions you are authorized to use:

```
AsdAuthorize.exe --authorized-codegenerators
```

Omit the progress information:

```
AsdAuthorize.exe --available-languages --verbosity 1
```

Only show languages for a specific version:

```
AsdAuthorize.exe --available-languages -v 9.2.3 --verbosity 1
```

ASD:Verify

ASD:Verify (AsdVerify.exe on Windows, asdverify on Linux), is a command-line tool for:

- Checking whether ASD models are verified. This can be useful in a build system to check that all ASD models in the archive are verified. This option does not require a server connection.
- Checking whether generated code is generated from a verified model. This option does not require a server connection.
- Verifying any ASD models for which the verification status is out-of-date. This is handy when you change an interface model and need to re-verify any dependent design models that surround it.

The tool supports help, server connection options, version and language, wildcards and recursion as described in Command-line Tools.

The exit code is 0 for succes, 1 for an error, and 2 if a model did not pass verification. When processing multiple models (using wildcards and/or recursion), processing continues with the next model if one model does not pass verification. You can change this behaviour with the `--stop-on-failure` option. In that case, processing will stop. Note that processing always stops on a non-verification error.

Checking whether models are verified

You can check whether a model is verified using the `--query-model` command. Note that this command does **not** require a server connection.
Depending on the `--verbosity` option, it can report progress information, overall verification status, and individual check statuses.
By default, the version and language stored in the ASD models is used, but you can supply your own as well. The verification status of a model will be reported as "outdated" if the version and language do not match the verified version and language.

Examples:

Query all models in a directory:

```
AsdVerify.exe --query-model -v 9.2.3 -l java *.[id]m
```

Check all design models in a directory tree starting at "asdmodels":

```
AsdVerify.exe --query-model -v 9.2.3 -l java --name *.dm --recurse ./asdmodels
```

Check all design models in a directory tree, only reporting overall verification status (not check statuses):

```
AsdVerify.exe --query-model --verbosity 2 -v 9.2.3 -l java --name *.dm --recurse ./asdmodels
```

Checking whether code was generated from a verified model

The `--query-code` command is useful for checking that your code is generated from a verified model. Note that this command does **not** require a server connection.
Depending on the `--verbosity` option, it can report progress information and overall verification status.
By default, the version and language stored in the ASD models is used, but you can supply your own as well. The verification status of a model will be reported as "outdated" if the version and language do not match the verified version and language.

Examples:

Query all C++ code files in a directory:

```
AsdVerify.exe --query-code --recurse --name *.cpp .
```

Verifying models

The `--verify` command displays the same results as the `--query-model` command. However, if the verification status of a model is not up-to-date, the model is verified first. This requires a server connection. Depending on the `--verbosity` option, the tool can report progress information, overall verification status, and individual check statuses.

The `--verify` command is the default command, so you can omit it from the command-line. By default, the version and language stored in the ASD models is used, but you can supply your own as well.

Note: the command-line tool only verifies a model if its verification status is out-of-date. Re-verifying a model when its status is not out-of-date is only possible using the ASD:ModelBuilder.

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Verify all models in a directory:

```
AsdVerify.exe -v 9.2.3 -l java *.[id]m
```

Verify all models in a directory, stopping the process if a model fails:

```
AsdVerify.exe -v 9.2.3 -l java --stop-on-failure *.[id]m
```

Verify all models in a directory tree, using the version and language stored in the models:

```
AsdVerify.exe --recurse ./asdmodels
```

Tip:
On Linux, if you installed ASD:Suite from a .deb or .rpm package, a symlink "asdv" is present which you can type instead of full "asdverify"

ASD:Generate

ASD:Generate (AsdGenerate.exe on Windows, asdgenerate on Linux), is a command-line tool for:

- Generating code from ASD models
- Downloading the ASD:Runtime

The tool supports help, server connection options, version and language, wildcards and recursion as described in Command-line Tools.

Generating code

Generating code is done with the `--code` command. This command is the default command, so you can omit it from the command-line. You can supply a version and language, and if you don't, the version and language stored inside the models will be used. Specifying an output directory is done with `--output`. If you specify `--all`, then for each design model, all surrounding interface models are processed as well.

The exit code is 0 for succes and 1 for any error. When processing multiple models (using wildcards and/or recursion), processing stops if one model fails.

Note: any existing files will not be overwritten if the downloaded files are identical. This avoids touching the files, allowing your build system to detect that they need not be re-compiled.

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Generate code for design model AlarmSystem.dm:

```
AsdGenerate.exe -v 9.2.3 -l java AlarmSystem.dm
```

Generate code for design model AlarmSystem.dm and all its interface models:

```
AsdGenerate.exe -v 9.2.3 -l csharp --all AlarmSystem.dm
```

Generate code for all models in the directory "asdmodels":

```
AsdGenerate.exe -v 9.2.3 -l cpp --recurse ./asdmodels
```

Generate code for all models in the directory "asdmodels", using the version and language stored in the ASD models:

```
AsdGenerate.exe --recurse ./asdmodels
```

Note:

- Before the generated code can be compiled and executed, the following steps need to be performed:
  - The ASD Runtime source must be downloaded from the ASD:Server, see instructions below or in "Downloading the ASD Runtime using the ASD:Suite ModelBuilder".
  - The files in which user defined parameter types are defined have to be made available during compilation. See "Ensuring correct referencing of user defined types" for details.

Downloading the ASD:Runtime

To download the ASD:Runtime, use the `--runtime` command. Supply a language, version, and an output directory.

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Download the ASD:Runtime for java:

```
AsdGenerate.exe --runtime -v 9.2.3 -l java --output ./code/runtime
```

Tip:
On Linux, if you installed ASD:Suite from a .deb or .rpm package, a symlink "asdg" is present which you can type instead of full "asdgenerate"

ASD:Read

The ASD:Read tool (AsdRead.exe on Windows, asdread on Linux) displays various properties of a model. You can use it to integrate ASD with other tools. The tool does not require a server connection. The tool can get the following properties from a model:

- Model properties, like model version, project, author, description etc.
- Code generation settings: the version, language
- Language-specific settings: namespace prefix, output path, debug information flag, etc.
- The paths to implemented and used models (in case of a design model).

For details, use `AsdRead.exe --help`.

The tool supports help as described in Command-line Tools.

Examples:

Get the author property of AlarmSystem.dm:

```
AsdRead.exe --author AlarmSystem.dm
```

Get the "Generate Debug Info" flag for C#. **Note**: language-specific properties require the `--language` option, which is *different* from the `-l` option (which gets the code generator language property).

```
AsdRead.exe --language csharp --debug AlarmSystem.dm
```

Tip:
On Linux, if you installed ASD:Suite from a .deb or .rpm package, a symlink "asdr" is present which you can type instead of full "asdread"

ASD:Edit

The ASD:Edit tool (AsdEdit.exe on Windows, asdedit on Linux) can be used for two purposes:

- To set various properties on multiple models at once
- To make the model name and main machine name equal to the file name in multiple models

The tool supports help, server connection options, wildcards and recursion as described in Command-line Tools.

Note: setting target language specific properties such as include files, output paths, debug flag etc requires the use of the `--language` option, which is *different* from the `-l` option (which sets the code generator language property).

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Set the author property of AlarmSystem.dm:

```
AsdEdit.exe --author "Lara Croft" AlarmSystem.dm
```

Set the code generation language and version on a directory of models recursively:

```
AsdEdit.exe -l csharp -v 9.2.3 --recurse ../asdmodels
```

Set the "Generate Debug Info" flag for C#. **Note**: language-specific properties require the `--language` option, which is *different* from the `-l` option (which sets the code generator language property).

```
AsdEdit.exe --language csharp --debug yes ../asdmodels
```

Tip:
On Linux, if you installed ASD:Suite from a .deb or .rpm package, a symlink "asde" is present which you can type instead of full "asdedit"

ASD:Compare

The ASD:Compare tool (AsdCompare.exe on Windows, asdcompare on Linux) indicates whether there are differences between two given ASD model files. The tool does not need a server connection. It accepts two file names and returns an exit code to indicate differences:

- Exit code 0: no differences found
- Exit code 1: error
- Exit code 2: differences found

Just like in the GUI Compare Tool, you can control which differences are reported (metadata, comments, or semantic differences). The `--relevance` flag controls this, see `AsdCompare.exe --help` for details.

Example:

```
AsdCompare "file1.dm" "file2.dm"
```

Note:
On Linux, do not confuse "asdcompare" with "asdcmp" or "asdcomparegui", the latter two start the graphical compare tool.

ASD:ModelConverter

The ASD:ModelConverter (AsdConvert.exe on Windows, asdconvert on Linux) is a command-line tool which upgrades one or more ASD model(s) built with previous major releases of the ASD:Suite. Note that the ASD:Suite ModelBuilder provides the same functionality under File-Upgrade Models (see "Upgrading ASD models"). On Linux, conversion is only supported for models made with ASD:Suite version 9.0.0 or higher. On Windows, conversion is supported for all versions.

The tool supports help, server connection options, wildcards and recursion as described in Command-line Tools.

The exit code is 0 for succes and 1 for any failure.

Examples:
The examples below assume a server connection is already made using the ModelBuilder. Otherwise, add your server connection options.

Upgrade design model AlarmSystem.dm:

```
AsdConvert.exe AlarmSystem.dm
```

Upgrade design model AlarmSystem.dm but save it as AlarmSystem-converted.dm:

```
AsdConvert.exe --output AlarmSystem-converted.dm AlarmSystem.dm
```

Upgrade a directory of models recursively:

```
AsdConvert.exe --recurse ../asdmodels
```

Upgrade a directory of models recursively, ignoring models for which conversion fails:

```
AsdConvert.exe --ignore-errors --recurse ../asdmodels
```

Tip: if you use wildcards and recursion, it can be useful to use the --ignore-errors flag to ignore any errors. That way, the converter will continue running over the remaining models if a model cannot be converted.

XZip license

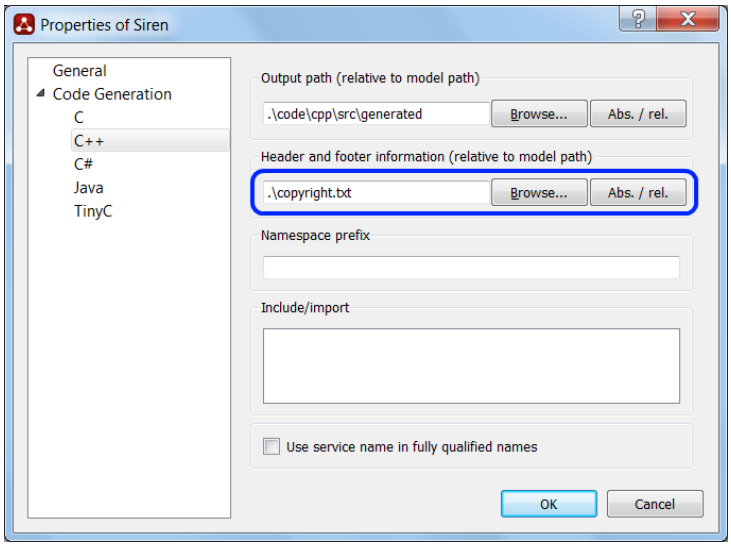
```
////////////////////////////////////////////////////////////////////////////
//
// Original authors' comments:
// ---------------------------
// This is version 2002-Feb-16 of the Info-ZIP copyright and license. The
// definitive version of this document should be available at
// ftp://ftp.info-zip.org/pub/infozip/license.html indefinitely.
//
// Copyright (c) 1990-2002 Info-ZIP.  All rights reserved.
//
// For the purposes of this copyright and license, "Info-ZIP" is defined as
// the following set of individuals:
//
//   Mark Adler, John Bush, Karl Davis, Harald Denker, Jean-Michel Dubois,
//   Jean-loup Gailly, Hunter Goatley, Ian Gorman, Chris Herborth, Dirk Haase,
//   Greg Hartwig, Robert Heath, Jonathan Hudson, Paul Kienitz,
//   David Kirschbaum, Johnny Lee, Onno van der Linden, Igor Mandrichenko,
//   Steve P. Miller, Sergio Monesi, Keith Owens, George Petrov, Greg Roelofs,
//   Kai Uwe Rommel, Steve Salisbury, Dave Smith, Christian Spieler,
//   Antoine Verheijen, Paul von Behren, Rich Wales, Mike White
//
// This software is provided "as is", without warranty of any kind, express
// or implied.  In no event shall Info-ZIP or its contributors be held liable
// for any direct, indirect, incidental, special or consequential damages
// arising out of the use of or inability to use this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it
// freely, subject to the following restrictions:
//
//    1. Redistributions of source code must retain the above copyright notice,
//       definition, disclaimer, and this list of conditions.
//
//    2. Redistributions in binary form (compiled executables) must reproduce
//       the above copyright notice, definition, disclaimer, and this list of
//       conditions in documentation and/or other materials provided with the
//       distribution. The sole exception to this condition is redistribution
//       of a standard UnZipSFX binary as part of a self-extracting archive;
//       that is permitted without inclusion of this license, as long as the
//       normal UnZipSFX banner has not been removed from the binary or disabled.
//
//    3. Altered versions--including, but not limited to, ports to new
//       operating systems, existing ports with new graphical interfaces, and
//       dynamic, shared, or static library versions--must be plainly marked
//       as such and must not be misrepresented as being the original source.
//       Such altered versions also must not be misrepresented as being
//       Info-ZIP releases--including, but not limited to, labeling of the
//       altered versions with the names "Info-ZIP" (or any variation thereof,
//       including, but not limited to, different capitalizations),
//       "Pocket UnZip", "WiZ" or "MacZip" without the explicit permission of
//       Info-ZIP.  Such altered versions are further prohibited from
//       misrepresentative use of the Zip-Bugs or Info-ZIP e-mail addresses or
//       of the Info-ZIP URL(s).
//
//    4. Info-ZIP retains the right to use the names "Info-ZIP", "Zip", "UnZip",
//       "UnZipSFX", "WiZ", "Pocket UnZip", "Pocket Zip", and "MacZip" for its
//       own source and binary releases.
//
////////////////////////////////////////////////////////////////////////////
```

Example of generated code customization when the specified text file has no directives

Let's consider the Siren interface model (known from the AlarmSystem example built and distributed by Verum) and the "copyright.txt" file with the following content:

```
//
// Copyright 2013 Verum Software Technologies BV
//
```

... and the C++ code generation properties for the Siren interface model as follows:



These is the C++ header file obtained after generating code from the Siren interface model:

```
// ////////////////////////////////////////////////////////////////////////////
// Generated for Project "" by Verum ASD:Suite Release 3, version 8.1.0.42234
// Generated from ASD Specification "Siren.im" last updated 2012-03-09T17:33:58
// ////////////////////////////////////////////////////////////////////////////
//
// Copyright 2013 Verum Software Technologies BV
//

#ifndef __ISIREN_INTERFACE_H__
#define __ISIREN_INTERFACE_H__

#include "passbyvalue.h"


#include <boost/shared_ptr.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>

class ISiren_API
{
public:
  enum PseudoStimulus {

    /// <summary>
    /// The VoidReply event indicates that the processing of the API call has been completed
    /// and the call can return when appropriate.
    /// </summary>
    VoidReply
  };
  virtual ~ISiren_API() {}

  /// <summary>
  /// Turn the siren on
  /// </summary>
  virtual void TurnOn() = 0;

  /// <summary>
  /// Turn the siren off
  /// </summary>
  virtual void TurnOff() = 0;
protected:
  ISiren_API() {}
private:
  ISiren_API& operator = (const ISiren_API& other);
  ISiren_API(const ISiren_API& other);
};
class ISirenInterface
{
public:
  virtual ~ISirenInterface() {}
  virtual void GetAPI(boost::shared_ptr<ISiren_API>*) = 0;
protected:
  ISirenInterface() {}
private:
  ISirenInterface& operator = (const ISirenInterface& other);
  ISirenInterface(const ISirenInterface& other);
};
#endif
```

Example of generated code customization when the specified text file has an "include" directive

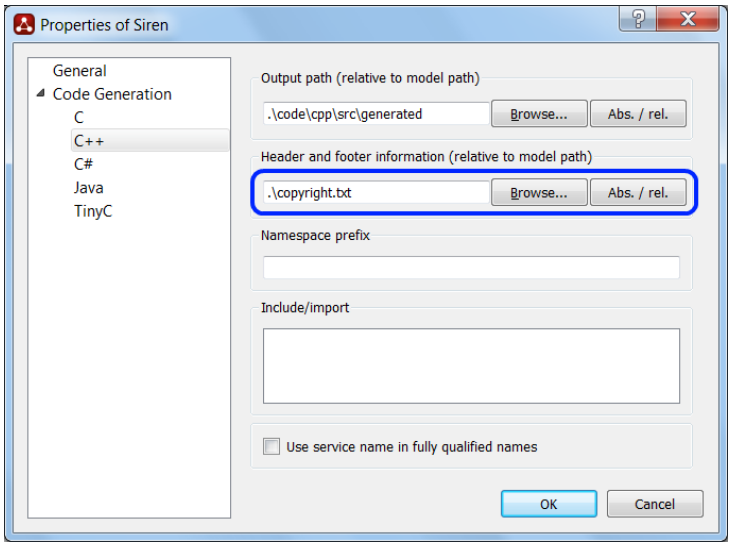
Let's consider the Siren interface model (known from the AlarmSystem example built and distributed by Verum) and the "copyright.txt" file with the following content:

```
//
// Copyright 2013 Verum Software Technologies BV
//
```
<include>some-other-text.txt</include>

where this is the content of the "some-other-text.txt" file:

```
//
// Just some other text
//
```

... and the C++ code generation properties for the Siren interface model as follows:



These is the C++ header file obtained after generating code from the Siren interface model:

```
// ////////////////////////////////////////////////////////////////////////////
// Generated for Project "" by Verum ASD:Suite Release 3, version 8.1.0.42234
// Generated from ASD Specification "Siren.im" last updated 2012-03-09T17:33:58
// ////////////////////////////////////////////////////////////////////////////
//
// Copyright 2013 Verum Software Technologies BV
//
//
// Just some other text
//

#ifndef __ISIREN_INTERFACE_H__
#define __ISIREN_INTERFACE_H__

#include "passbyvalue.h"


#include <boost/shared_ptr.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>

class ISiren_API
{
public:
  enum PseudoStimulus {

    /// <summary>
    /// The VoidReply event indicates that the processing of the API call has been completed
    /// and the call can return when appropriate.
    /// </summary>
    VoidReply
  };
  virtual ~ISiren_API() {}

  /// <summary>
  /// Turn the siren on
  /// </summary>
  virtual void TurnOn() = 0;

  /// <summary>
  /// Turn the siren off
  /// </summary>
  virtual void TurnOff() = 0;
protected:
  ISiren_API() {}
private:
  ISiren_API& operator = (const ISiren_API& other);
  ISiren_API(const ISiren_API& other);
};
class ISirenInterface
{
public:
  virtual ~ISirenInterface() {}
  virtual void GetAPI(boost::shared_ptr<ISiren_API>*) = 0;
protected:
  ISirenInterface() {}
```

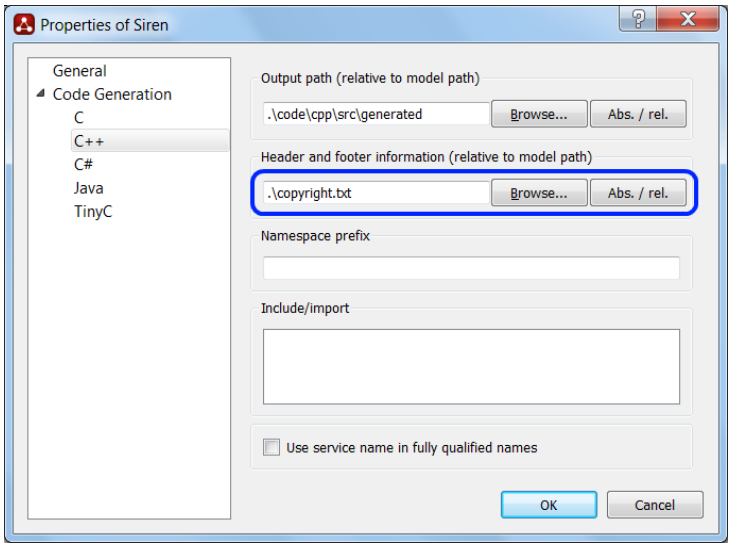
```
  private:
    ISirenInterface& operator = (const ISirenInterface& other);
    ISirenInterface(const ISirenInterface& other);
  };
  #endif
```

Example of generated code customization when the specified text file has a "generate" directive

Let's consider the Siren interface model (known from the AlarmSystem example built and distributed by Verum) and the "copyright.txt" file with the following content:

```
<generate/>
//
// Copyright 2013 Verum Software Technologies BV
//
```

... and the C++ code generation properties for the Siren interface model as follows:



These is the C++ header file obtained after generating code from the Siren interface model:

```
// ////////////////////////////////////////////////////////////////////////
// Generated for Project "" by Verum ASD:Suite Release 3, version 8.1.0.42234
// Generated from ASD Specification "Siren.im" last updated 2012-03-09T17:33:58
// ////////////////////////////////////////////////////////////////////////////

#ifndef __ISIREN_INTERFACE_H__
#define __ISIREN_INTERFACE_H__

#include "passbyvalue.h"


#include <boost/shared_ptr.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>

class ISiren_API
{
public:
  enum PseudoStimulus {

    /// <summary>
    /// The VoidReply event indicates that the processing of the API call has been completed
    /// and the call can return when appropriate.
    /// </summary>
    VoidReply
  };
  virtual ~ISiren_API() {}

  /// <summary>
  /// Turn the siren on
  /// </summary>
  virtual void TurnOn() = 0;

  /// <summary>
  /// Turn the siren off
  /// </summary>
  virtual void TurnOff() = 0;
protected:
  ISiren_API() {}
private:
  ISiren_API& operator = (const ISiren_API& other);
  ISiren_API(const ISiren_API& other);
};
class ISirenInterface
{
public:
  virtual ~ISirenInterface() {}
  virtual void GetAPI(boost::shared_ptr<ISiren_API>*) = 0;
protected:
  ISirenInterface() {}
private:
  ISirenInterface& operator = (const ISirenInterface& other);
  ISirenInterface(const ISirenInterface& other);
};
#endif

//
// Copyright 2013 Verum Software Technologies BV
//
```