ASD runtime semantics with code integration examples

ASD Runtime 9.2.3

TABLE OF CONTENTS

ASD runtime semantics with code integration examples

User Guide

ASD Runtime 9.2.3

This is a description of the ASD Runtime, a software package distributed as part of the ASD:Suite. The ASD Runtime 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.

In the "ASD Execution semantics" section you can find a description of how ASD components work at runtime, and in "Download and install" you can find guidelines to download the ASD Runtime.

In "Code integration guidelines" you can find descriptions about the content and role of the items in the ASD Runtime , together with guidelines for integrating the ASD generated code into your system.

In "Build integration guidelines" you can find descriptions about the content and role of the commandline tools, together with guidelines for integrating these tools into your build process.

ASD is licensed under EU Patent 1749264, US Patent 8370798 and Hong Kong Patent HK1104100

Interface models and Design models

ASD is a component-based technology whereby a system is defined in terms of ASD components and foreign components. A *component* is the common unit of architectural decomposition, specification, design, formal verification, code generation and runtime execution. An *ASD component* is a software component that is specified, designed, verified and implemented using ASD and is specified by:

1. An *ASD interface model* specifying a *service*: the externally visible behaviour of a component, and
2. An *ASD design model* specifying its inner working and how it interacts with other components.

An ASD component *implements* a service that is used by its *clients* and can *use* services that are implemented by its *servers*.

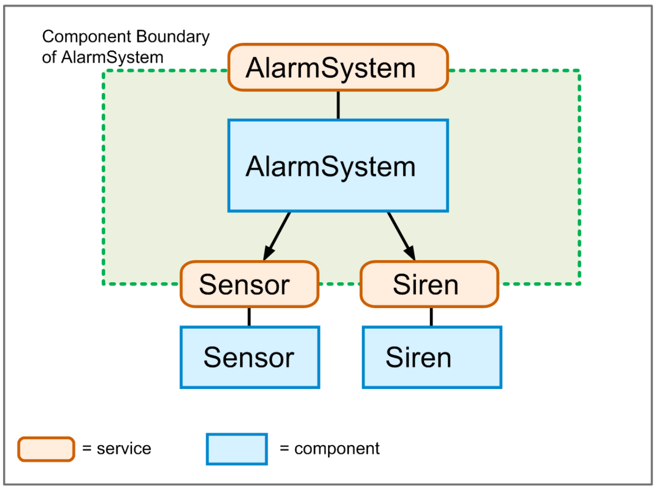
Collectively, the services between a component and its clients and servers form an imaginary border, called the *component boundary*. Information crosses this boundary in the form of events where one component *sends* the event (*action*) and another component *receives* the event (*trigger*):

- A *trigger* is either a call event coming from a client, a reply event coming from a server or a notification event coming from a server.
- An *action* is a call event that is sent to a server, a reply event sent to a client, or a notification event that is sent to a client.

Foreign components are hardware or software components of a system that are not developed using ASD. For example, third party components, legacy code or handwritten components representing those parts of a system that cannot be or are not generated from ASD designs. As they need to be used by ASD components, they must correctly interface and interact with them. The externally visible behaviour of each foreign component is specified in the form of an *ASD interface model*; foreign components do not have a corresponding ASD design model. These interface models are used for two purposes:

- Firstly, they are used when verifying an ASD component design. They represent allowable interactions of the ASD component with its environment.
- Secondly, they are used in code generation to create the correct interface header files in the specified target programming language.

The following figure depicts an ASD component *A* implementing a service *SA* and making use of components *B, C* and *D* via their respective services *SB, SC* and *SD*. Components *B, C* and *D* can be ASD components or foreign components.The component boundary comprises services *SA, SB, SC* and *SD*. The design of component *A* is specified in an ASD design model as will be the designs of any of the used components that are also ASD components. Each of the services is specified by an ASD interface model, irrespective as to whether or not the components implementing them are ASD components.



Context of an ASD component design

An *ASD interface model...*:

- ...describes the externally visible behaviour of a component and is as implementation-free as possible, meaning that the model defines *what* the component does under every circumstance but not *how* the component will do it. This allows the external behaviour to be specified independently of any specific implementation.
- ...is an abstraction of the component or system implementation that every compliant design is required to implement.
- ...is defined in terms of those events that pass between the component and its clients and contains two types of information:
  - method signatures specified in a manner consistent with the chosen target programming language, and
  - externally visible behaviour specified in the form of a *Sequence-Based Specification* (SBS).
  Additionally, an interface model can define one or more *modelling events* These are used to model the hidden internal behaviour of the design.
- ...is used, during code generation, to generate only interface declarations and not to generate executable logic.

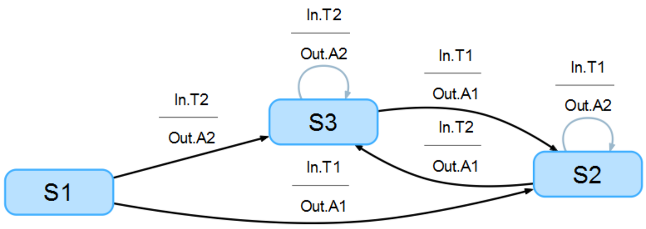
An *ASD design model ...*:

- ...describes the complete internal behaviour of a component, thereby defining one of the (many) possible implementations that faithfully comply with its implemented interface model. It is defined in terms of events that pass between the component, its clients and its servers only.
- ...contains behaviour in the form of one or more SBSs; information is imported from the set of interface models specifying the component boundary. To handle complex designs, a design model can be hierarchically decomposed into a main machine and one or more sub machines.
- ...is used, during code generation, to generate only executable logic and not to generate interface declarations.

SBS with rules and rule cases

Within ASD, both interface and design models are made 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.

An SBS is divided into equivalence classes, each of which corresponds to a state in a Mealy machine. A Mealy machine is a sequential machine in which the output depends on both the current state of the machine and the input.



The state diagram of a simple Mealy machine

Within each state, the SBS must define a *rule* for every possible trigger and each rule has one or more *rule cases* that define the resulting actions.

Rule cases can be *enabled* or *disabled* by using *guards* which are expressed as boolean expressions over *state variables*, i.e. when a guard is evaluated as false, the rule case is *disabled*; when the guard is evaluated as true, the rule case is *enabled*. A rule case without a guard is equivalent to a rule case with a constant guard "true". State variables are the means by which fine-grained history is captured and guards are the means by which fine-grained history-based control flow decisions are made.

In addition to its guard, each rule case specifies

- a sequence of actions to be executed sequentially,
- zero or more simultaneous assignments to state variables and
- the next state

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments
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*
8	**Activated_Idle**						
10	IAlarmSystem	SwitchOn+		Illegal		-	*Illegal - alarm already activated*
11	IAlarmSystem	SwitchOff		IAlarmSystem.VoidReply		Deactivating	*Busy deactivating*
12	Internal	[AlarmTripped]		IAlarmSystem_NI.Tripped		Activated_AlarmMode	*Sensor has tripped the alarm*

Screenshot of an SBS fragment from an ASD Interface Model

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

Operational semantics

The ASD Runtime supports two types of execution models: The *Multi-threaded* execution model and the *Single-threaded* execution model.

In the *Multi-threaded* execution model, each ASD component that uses a service with a notification interface gets a Deferred Procedure Call (DPC) thread that empties the component queue. Furthermore, the ASD Runtime uses Operating System (OS) synchronization primitives for Multi-threaded applications. See "Notification events" and "Thread context switching and monitor semantics in practice" for details.

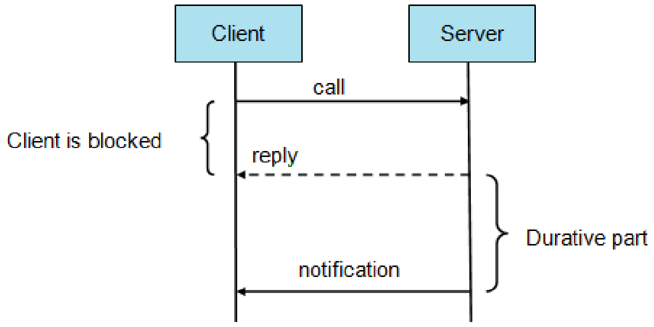
In the *Single-threaded* execution model, no threads are instantiated in the ASD components and execution of the events takes place synchronously. See "The *Single-threaded* execution model" for details.

Terms of use | Privacy Policy

Durative and non-durative actions

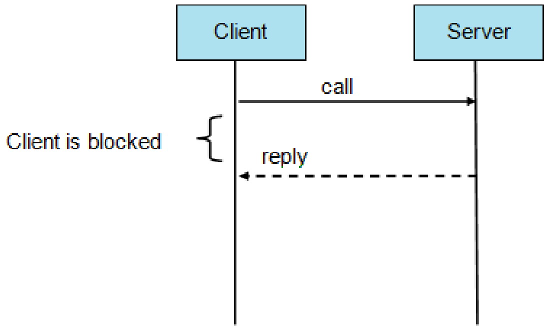
An architecture built using ASD components follows the client-server pattern, where clients can synchronously invoke one or more servers, and servers can asynchronously invoke notifications to clients.

A durative action is where a client synchronously invokes an operation on a server and the result (a notification) comes later (i. e. asynchronously). Until the operation returns, the client remains blocked and it cannot invoke other operations at the same or other servers. After the method has returned to the client, the server ***is processing*** the durative part and eventually informs the client asynchronously through the notification.



A durative action

A non-durative action, on the other hand, is where the client synchronously invokes an operation on the server and remains blocked until the server ***has completely processed*** the request.



A non-durative action

Notification events

Notification events usually carry the result of a durative action but they can also inform the client about spontaneous actions on the server side.

In the *Multi-threaded* execution model the notification events are executed in a decoupled way: the server synchronously posts the notification event in a queue of the client after which the DPC server thread of the client will pick it up and execute it. DPC stands for Deferred Procedure Call.

In the *Single-threaded* execution model the server synchronously posts the notification event in a queue of the client and it informs the client that notifications are available for processing, after which the execution thread processes the respective notification(s). See "The *Single-threaded* execution model" for details.

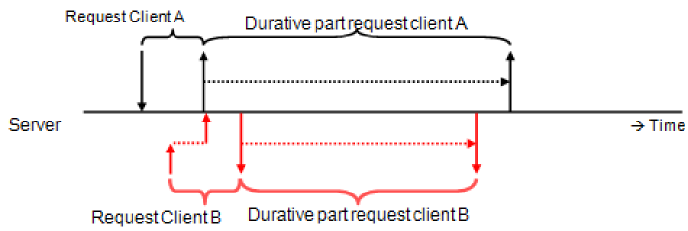
Run-To-Completion semantics

Within ASD, the run-to-completion semantics refers to the execution at runtime of one rule case. It has the following meaning:

- All actions are completely processed before the transition to the next state occurs
- All state variable updates are performed before the transition to the next state occurs

Monitor semantics

Within ASD, it is assumed that the components have monitor semantics, i.e. only one client has access to the component at any time (components are non-reentrant). The following figure provides a graphical representation of what is meant by monitor semantics within ASD:



Monitor semantics within ASD

In the above picture, it is important to note that the order of invoking notifications by a server is independent of the order of invoking methods at the server. The request of client B is processed later by the server, but the durative part is completed earlier as client B is invoked by the corresponding notification earlier than client A. Additionally you can observe that the processing of client B is only started after synchronously replying to the request of client A.

Timers

ASD components can make use of the ASD Timer service by instantiating as many Timers as they need. Timers are commonly used within mechatronics and communication systems for guarding against failures. ASD timer components are supplied as part of the ASD Runtime package. This is done to make them independent of execution platforms and to provide the timer cancel guarantee.

> **Note:** The ASD Timer service can be used only in design models having *Multi-threaded* as execution model.

When using a timer to guard a mechanical movement, for example, a component will be expecting either a notification event signifying the movement has ended or a timer event signifying that the notification event did not arrive within the expected time. If the movement-ended notification event is received first, the design will cancel the timer. In many runtime environments the request to cancel a timer request can race with the expiration of the respective timer. This racing results in a component receiving timer expired signals from (just) cancelled timers. Unless the runtime environment provides mechanisms for dealing with such situations, this adds unnecessary complexity in the design, often causing errors.

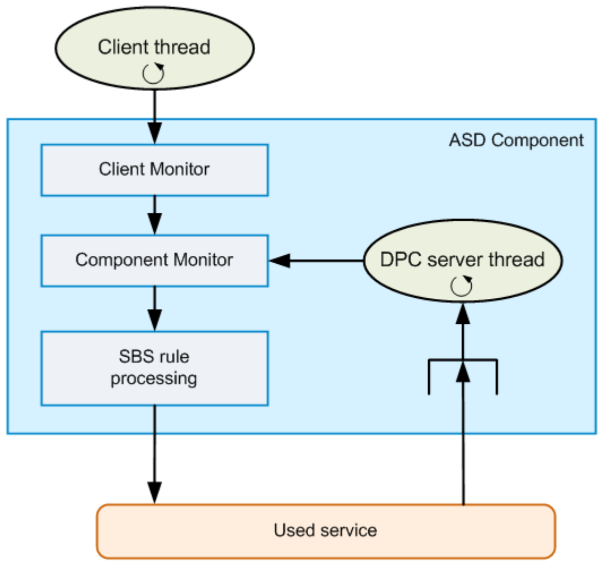
An ASD component which uses the ASD timer guarantees that if a timer has been cancelled before the rule corresponding to the timer expiry event has been executed, the timer event will never be seen by the component and the corresponding rule will not be executed.

Thread context switching and monitor semantics in practice

Within ASD, rules are executed atomically. When a component receives a trigger in a state, the guards of every rule case belonging to the rule are evaluated and the enabled rule case selected. If there is more than one enabled rule case (interface models only), the choice between them is non-deterministic. The actions are executed sequentially and when completed, the state variable updates (if any) are performed using simultaneous assignment semantics and a transition is made to the next state.

An ASD component synchronously executes client triggers using the clients' thread context. In contrast, notification events that it receives from its servers are decoupled via a FIFO ordered *queue* and are executed by a *DPC server thread* belonging to the component. This is done so that a server is never blocked when invoking a notification event implemented by its clients, thus removing a common cause of deadlocks emerging during system composition. Both the queue and the DPC server thread are generated automatically as part of the component if any of its servers has a notification interface. An ASD component must therefore be thread-safe in the face of competing requests for service by multiple client threads and its own DPC server thread.

An ASD component behaves like a pair of nested monitors in order to serialise access by competing client threads and its own DPC server thread. The following figure presents two monitors set up as nested monitors:



A pair of nested monitors

In ASD the outer monitor is called the *client monitor* and serves to serialise client access to the component. The inner monitor is called the *component monitor* and serves to serialise access between a client thread and the component's DPC server thread, thereby implementing the atomicity property of rule execution in both the formal models and the resulting code. The DPC server thread always takes precedence over the client thread when competing for the component monitor and this is guaranteed by explicit synchronisation, making it unaffected by thread priorities or scheduling policies of the underlying OS. Once inside the component monitor, the DPC server thread sequentially processes all notification events in the queue by invoking the corresponding triggers on the component before exiting the component monitor.

ASD uses the synchronisation primitives of the underlying target platform OS to implement the client and component monitors. As such, ASD imposes no ordering on the sequence in which pending client requests are serviced by a component. When one client request is processed to completion, the ASD Runtime instructs the underlying OS to select one of the waiting Client threads and schedule it for execution. The selected client thread and thus the effect of thread priorities is determined by the underlying OS and not by ASD.

When a client thread invokes a trigger it must enter both the client monitor and the component monitor and atomically execute the corresponding rule. When the rule has been completely executed, the client thread exits the component monitor and must pass the *client barrier* before it can exit the client monitor. If the client barrier is closed, the client thread is blocked from further execution inside the client monitor (but outside the component monitor to ensure that the DPC server thread can access the component); if the client barrier is open, the client thread exits the client monitor and continues execution without waiting.

The client barrier is closed when the client thread enters the client monitor and before it starts executing any rule; it is opened when some rule case executes a set of actions containing a matching reply to the client trigger. If the rule corresponding to the client trigger contains this reply, the client barrier will be open when the rule execution is completed and the client thread will continue without waiting. Otherwise, the client barrier remains closed and the client thread remains blocked inside the client monitor until the client interface is subsequently opened as a result of some (asynchronous) notification event originating from a server. In such case, the client thread can continue to execute the moment the DPC server thread has processed the corresponding reply event that has opened the client barrier. Such continuation fully depends on the scheduling algorithm within the operating system.

The *Single-threaded* execution model

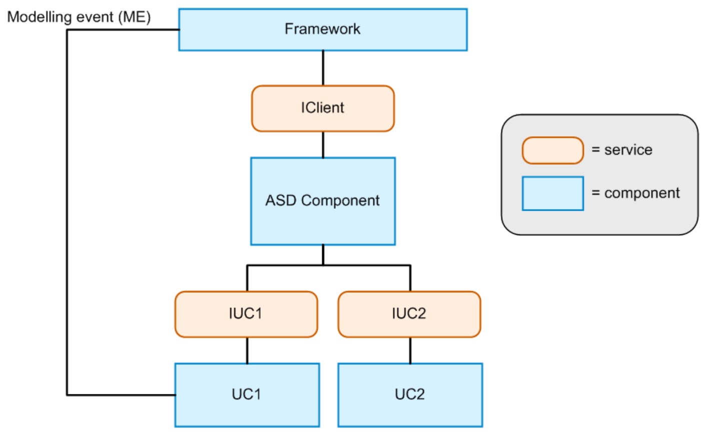
The *Single-threaded* execution model is useful in a thread constrained environment or when simply a large number of components are instantiated.

Generally speaking, the main feature of the *Single-threaded* execution model is that no DPC server thread is instantiated to process notification events. Notification events are still decoupled by a queue, but they are processed in a different manner:

1. When a client thread would normally leave the component monitor (as a result of a matching reply event), the client thread is now re-used to process notification events;
2. Spontaneous notification events from a used server that are not the result of a client request are processed by an explicit request from the used server.

The semantics of the *Single-threaded* execution model

The basic behaviour of *Single-threaded* execution model is demonstrated in the following composition of a client ASD component and two used components (UC1 and UC2).



Component architecture where the ASD *Single-threaded* execution model is recommended

The following list reflects the processing flow of a trigger with a notification event as associated action:

1. The **Framework** invokes the *API.E1* trigger on the **ASD Component**, which, in its turn, invokes **UC1** via *UC1:API.E1* (see rule case number 3 in **ASD_Component_SBS**).
2. *UC1:API.E1* starts a durative action, the completion of which is notified by a later event from the framework. It also sends a notification event *UC1:NI.NE1* (see rule case number 3 in **UC1_SBS**).
3. The return from *UC1:API.E1* signals the end of the component's list of actions (in this case there is only a single action in the list) and so the component performs any state variable updates and transits to *State_2*. This obeys the atomic rule execution required by the ASD operational semantics.
4. Before the **ASD Component** sends a reply to the **Framework**, the **ASD Proxy**, a proxy for the **ASD Component,** checks the component's queue, the **ASD Queue**, and empties it by sequentially executing the rules corresponding to every notification event in the queue; in this example there is one such notification event, namely *UC1:NI.NE1*. The rule corresponding to this notification event is the rule in the component's *State_2* (see rule case number 11 in **ASD_Component_SBS**). These actions are all performed under the thread context of the **ASD Component**'s client, namely the **Framework**.
5. When all queued notification events are processed to completion, the **ASD Component** sends a return to the **Framework**, leaving itself in *State_3*.
6. Subsequently, the **Framework** invokes some bottom edge interface on **UC1** to signal the completion of the durative action it started earlier triggered by *UCI:API.E1* (see rule case number 3 in **UC1_SBS**). The corresponding rule is executed under whichever thread context is used by the **Framework** for this call. The response of **UC1** is to send the *UC1:NI.NE2* notification event to the **ASD Component**; this is queued in the **ASD Queue** as usual.
7. Before returning to the framework, **UC1** is required to notify the **ASD Proxy** that it has reached the end of its rule and thus any pending notification events in the **ASD Queue** must be processed. In the diagram below, this is the red event called *UC1:ASD.processCBs*.

   Note: the *processCBs* method it is not to be specified as an event in the ASD model, it is inserted by the code generator at the right place in the generated code.

   The **ASD Proxy**empties the **ASD Queue**and processes each event to completion, in this case the single event *UC1:NI.NE2*.The rule for this event is processed by the **ASD Component**in its *State_3*.
8. The **Client** invokes **UC2** via *UC2:API.E1* when processing trigger *UC1:NI.NE2*.
9. **UC2** responds to an *UC2:API.E1* by posting two notification events, namely *UC2:NI.NE1* and *UC2:NI.NE2*. These are placed into the **ASD Queue**.
10. The used component **UC2** sends a reply to the **ASD Component** completing the actions of the rule for *UC1:NI.NE2*. The **ASD Component** in its turn completes any state variable updates, transits to *State_4* and sends a reply to the **ASD Proxy**.
11. The **ASD Proxy** continues to empty the **ASD Queue**, in this example processing *UC2:NI.NE1* and *UC2:NI.NE2* and eventually sends a return to **UC1** (see the red return event at the bottom of the sequence diagram). *UC2:NI.NE1* is processed as expected in *State_3*, *UC2:NI.NE2* is processed in *State_5* and the **ASD Component** transits to *State_1*.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**State_1 (initial state)**					
4	API	E1		UC1:API.E1; API.VoidReply		State_2
9	**State_2**					
12	API	E1		API.VoidReply		State_2
13	UC1:NI	NE1		NoOp		State_3
17	**State_3**					
20	API	E1		API.VoidReply		State_3
22	UC1:NI	NE2		UC2:API.E1		State_4
25	**State_4**					
28	API	E1		API.VoidReply		State_4
31	UC2:NI	NE1		NoOp		State_5
33	**State_5**					
36	API	E1		API.VoidReply		State_5
40	UC2:NI	NE2		NoOp		State_1

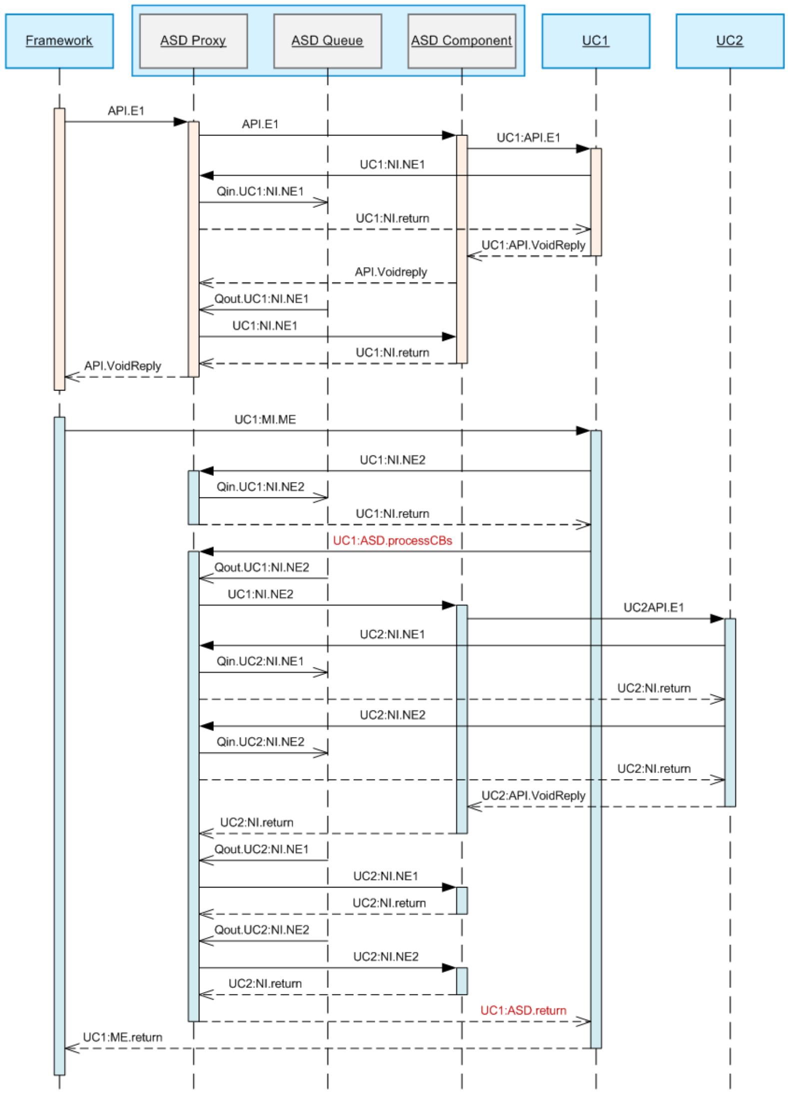
(part of) SBS of the design model for ASD Component

	nterface	Event	Guard	Actions	State Variable Updates	Target State
1	**State_1 (initial state)**					
3	API	E1		API.VoidReply; NI.NE1		State_2
5	**State_2**					
8	MI	ME		NI.NE2		State_1

(part of) SBS of the interface model for used component UC1

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**State_1 (initial state)**					
3	API	E1		API.VoidReply; NI.NE1; NI.NE2		State_1

SBS of the interface model for used component UC2



Sequence diagram for the *Single-threaded* execution model

In the *Single-threaded* execution model, the used component must cooperate in its behaviour with the proxy of the ASD Component at runtime in order to achieve the necessary runtime behaviour. In particular, after executing the behaviour corresponding to the modelling event rule case (i.e. one corresponding to a notification event), it must give the ASD Runtime the opportunity to sequentially process any queued notification events before it returns to the Framework.

If the used component is an ASD component this cooperative behaviour can be guaranteed in the generated code. Otherwise, if the used component is a foreign component, this cooperative behaviour has to be implemented by hand in the used component.

Limitations of the *Single-threaded* execution model

Although the Single-threaded execution model has the advantage of reduced resource consumption, the usage of this model is more restricted than the Multi-threaded model:

1. See "*Single-threaded* vs *Multi-threaded* - Instantiated resources" for limitations in using Single-threaded components in combination with Multi-threaded components.
2. A Single-threaded component can only be accessed via a single thread.
3. ASD Timers cannot be used in the Single-threaded context because their implementation relies on the Multi-threaded execution model, i.e. they rely on having DPC threads.
4. A client call event may not rely on a used service notification event for generating the reply since this causes a deadlock in a Single-threaded context (the single thread is waiting for another thread to post a notification event, but by definition no other thread is allowed to do that).
5. Broadcast notification interfaces are not supported.
6. Not using all interfaces of used Single-threaded services is not supported. A design model using a Single-threaded service must use **all** of its interfaces.
7. Yoking is not supported. Yoking has no meaning in a Single-threaded context.
8. Singleton notification events are not supported. Singleton notification events have no meaning in a Single-threaded context.

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

Single-threaded vs. *Multi-threaded* - Instantiated resources

An ASD component can implement a *Single-threaded* interface model or a *Multi-threaded* one. See "Specifying the execution model" for details. Similarly it's used components can either all implement *Single-threaded* interface models or all implement *Multi-threaded* interface models. See the following table for resource instantiation depending on the various combinations:

	The ASD component implements a Single-threaded interface model	**The ASD component implements a Multi-threaded interface model**
All used components implement *Single-threaded* interface models	**Allowed** No DPC thread created No mutexes/condition variables created	**Not allowed**
All used components implement *Multi-threaded* interface models	**Allowed** only with certain conditions e.g. no notification events No DPC thread created No mutexes/condition variables created Note: Errors might be reported if some of the Multi-threaded models do not conform to the expected rules.	**Allowed** DPC thread created Mutexes/condition variables created
There is at least one used component which implements a *Single-threaded* interface model and there is at least one used component which implements a *Multi-threaded* interface model	**Allowed** only with certain conditions e.g. no notification events No DPC thread created No mutexes/condition variables created Note: Errors might be reported if some of the Multi-threaded models do not conform to the expected rules.	**Not allowed**
No used components	**Allowed** No DPC thread created No mutexes/condition variables created	**Allowed** No DPC thread created Mutexes/condition variables created

Basically, the execution model property of the implemented service determines the use of Multi-threading primitives. The execution model property of the used service determines the creation of the DPC thread. See "Specifying the execution model" for details about how to see and/or change the execution model property.

The ASD Runtime software package

The ASD Runtime is a software package which enables ASD:Suite generated code to run on various software development platforms. The ASD Runtime is provided as source code in a target programming language, therefore it has to be compiled by you in your software project, in the same way as you compile any other source module written (or generated) in the respective programming language.

The ASD Runtime software package is available for download from the ASD Server. The ASD Runtime is language specific, i.e. there is one package for each target programming language supported by the ASD:Suite. The number and type of files in the ASD Runtime software package vary according to the selected language.

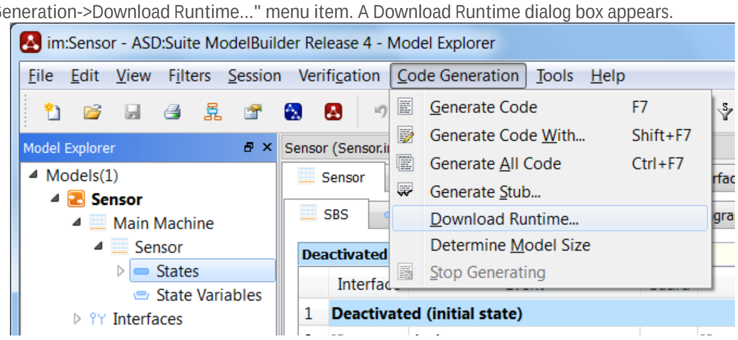
The used version of the ASD Runtime must match the selected code generator version, otherwise compilation errors will occur in the generated code. A new ASD Runtime is released with each code generator version.

The ASD Runtime should not be modified/altered by you. If you do so the guarantees provided by ASD are invalidated.

ASD Runtime download using the ASD:Suite ModelBuilder

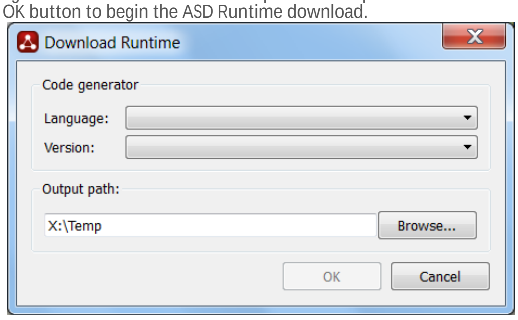
The following describes the steps you should take to download the ASD Runtime for a specific programming language using the ASD:Suite ModelBuilder:

1. Select the "Code Generation->Download Runtime..." menu item. A Download Runtime dialog box appears.



Menu item to download the ASD Runtime

2. Choose the code generator language and version number from the provided 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 window

When finished, a list of the ASD Runtime files that have been downloaded will appear in the "Output Window" of the ASD:Suite ModelBuilder. The download is complete when the "==== Finished successfully ====" message appears.

verum

Home Product Technology Resources Training Purchase Company

ASD Runtime download using the ASD:Suite Commandline Client

You can download the ASD Runtime using the ASD:Suite Commandline Client following these steps:

1. Open a DOS command window. This can be done by opening the Windows Start Menu and then selecting the "ASD Suite Release <release_number> V<version_number> --> ASD Client Command Prompt" option from the program list.
2. Type in the following command at the DOS prompt:

```
AsdGenerate --runtime -v <code_generator_version> -l <language> -o <output-dir>
```

- Replace `<code_generator_version>` with the version of the ASD Runtime you wish to download
- Replace `<language>` with 'cpp', 'csharp', 'java', 'c' or 'tinyc'.
- Replace `<output-dir>` with the path where you wish the ASD Runtime files to be downloaded.

Note: there are additional settings that you may need to supply, like user credentials. For a list of all ASD Runtime download options, type in the following command at the DOS prompt: `AsdGenerate -h`

For example:

```
AsdGenerate --runtime -v 9.2.3 -l cpp -o X:\code\runtime
```

downloads the files of the C++ ASD Runtime version 9.2.3 into the "X:\code\runtime" directory.

How to integrate the generated code in your development environment

When you have generated code from ASD models, you can start integrating the code into your application.

Read the following sections for code integration related information for the various languages supported by ASD and the ASD: Suite.

verum

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

The ASD Runtime for C++

The following files make up the ASD Runtime package for C++

File Name	Description
asdDefConfig.h	Header file containing configuration of ASD Runtime
asdInterfaces.h	A set of ASD dedicated interfaces, currently for *Single-threaded* execution model
asdSingleThreaded.h	Interface and implementation of *Single-threaded* execution model
asdMultiThreaded.h	Interface of *Multi-threaded* execution model
asdMultiThreaded.cpp	Implementation of *Multi-threaded* execution model
asdDiagnostics.h	Interface to the diagnostics tracing and illegal handlers
asdDiagnostics.cpp	Implementation of the default diagnostics tracing and illegal handlers
asdDataVariable.h	Utility for data variables
asdPassByValue.h	Utility to guarantee pass by value semantics for parameter passing
asdTransfer.h	Utility for parameter passing
asdUsedServiceRef.h	Utility for used service reference operators

The C++ ASD Runtime files support the generated ASD C++ code and provide an interface to handwritten code. The sections of the C++ ASD Runtime files that support ASD generated code have a specific format and may NOT be edited and the respective files should NOT be included in any handwritten code. These sections in the files are prefaced by "namespace asd_<#>" where <#> is a specific build version number. Example: namespace asd_31362.

Parameter definition and parameter passing in C++

The declaration of parameters and how they are passed on as arguments in an ASD design model is described in the user manual. This section describes the language specific details for C++. Foreign code that interfaces with the ASD generated code must adhere to the semantics as defined here.

There are no type checks for parameters. It is your job to ensure that the specified types are correct. For example, all parameters that are used with a certain data variable must have the same type. Errors caused by wrong type specifications appear as compilation problems when you attempt to compile the generated code in your development environment.

Parameter passing in C++

For [in] parameters in C++ the following holds:

- An [in] parameter can be of any C++ type or user defined type. Note that the designer must ensure that such user defined types are accessible (to the compiler) using the import/include field.
- An [in] parameter is always passed by const-reference for all events.
- Where the type evaluates to a pointer, the pointer is the underlying "value type" passed by const-reference. In such case, the lifecycle management of the referenced values is not managed by ASD.
- An [in] parameter can not be declared "volatile".
- The [in] parameters of notification events are copied to avoid common data-racing issues that can occur in Multi-threaded environments, and therefore their types must be copyable. Since copying is performed using the copy constructor, a public copy constructor with usual semantics must be available.

For [out] and [inout] parameters in C++ the following holds:

- An [out] or [inout] parameter can be of any C++ type or user defined type. Note that the designer must ensure that such user defined types are accessible (to the compiler) using the import/include field.
- All [out] and [inout] parameters are passed by reference.
- All [inout] arguments must be initialised (i.e. they must have a pre-existing value).
- An [out] parameter has true out-semantics and pre-existing values are not available anymore. For foreign code it is assumed that no partial updates have been performed.
- ASD requires that the first use of an [out] parameter of an application call event used as trigger is (also) an [out] parameter of an application call event used in a sequence of actions. Subsequent uses of the respective parameter can be as [in], [out] or [inout] parameter of an application call event used in a sequence of actions.

For the data variables the following holds:

- Because data variables are being read and written to, the data variable **type** cannot be const.
- Data variables are being initialised to zero during construction of the component.
- The type must be copyable.
- The type must implement a public assignment ("=") operator with the expected, usual semantics; other semantics or those with side effects are not guaranteed to compile, link or execute correctly.
- The type must have a public default constructor
- The type must have a public destructor.

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

Component integration in C++

This section describes how handwritten client components and/or handwritten used components can be made to work with the ASD generated components. The best way to go about this is to make use of the stub generator that is present in the ASD: Suite ModelBuilder. The paragraphs below describe how to use the stub generator effectively to quickly generate code for respecively the handwritten client component(s) and handwritten used component(s). In addition this section also described how component creation works with construction parameters

Stub generation for calling components

To generate code for a handwritten **client** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C++ in this case) as well as the version.
4. Select the "Client Stub" radio button and fill in the following options:
   - Fill in the name of the **stub** component to be generated in the construction parameter field. This name is the name of the **calling** component that is created for you including all queue plumbing, notification registration, etc.
   - Fill in the name of the **used** component that is to be referred in the construction parameter field. This name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
     Tip: Load the design model that uses the interface model for which you want to generate a stub. Select the interface model and right-click it; the ModelBuilder has now filled in the name of the used component for you as it matches the name of the construction parameter as found in the service references tab.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the client component for C++ can be found here.

Stub generation for used components

To generate code for a handwritten **used** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C++ in this case) as well as the version.
4. Select the "Used Component Stub" radio button and fill in the following options:
   - Fill in the name of the component to be generated in the construction parameter field. Note that this name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
   - Select the component type in the dropdown menu.
   - Select "Generate debug info" if you want trace statement to be included in your stub code.
   - Select "Generate synchronization primitives" if you want your functions to contain mutexes in order to be thread-safe. This is typically not needed in the Single-threaded model. In the Multi-threaded this can be needed when the stub has multiple clients and contains, for example, data that requires thread-safe access.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the used component for C++ can be found here.

Component creation using component construction parameters

The typical use-cases for designing ASD components using component construction parameters are systems in which it is undesirable or impossible at design time to specify which implementations will fulfil the dependencies or when the design relies on multi-client interfaces on non-singleton components, i.e. construction-time diversity or shared multiple:

- **Construction-time diversity** can be used when the choice between different implementations of components (or sub-systems) depends on system configuration and can be resolved before ASD components are instantiated. For example, the implementation of a protocol handler will be instantiated to match system configuration settings specifying which communication protocol should be used or which camera driver implementation is selected to match the actual camera hardware detected.
- **Multiple shared instances** of an ASD component can exist in a programs' address space if it has component type Multiple. When two different component instances need to use the same instance of a used service that used service has a multi-client interface. Component construction parameters should be used to bind the different client components to the shared used service instance when the used service implementation cannot be made Singleton.

Note: See "Specifying the component type" for details about component types, Singleton or Multiple.

When you define construction parameters in the Model Properties of a design model, they end up in the signature of the GetInstance () method. You can read more about construction parameters in "Defining construction parameters".

Construction parameter	GetInstance parameter
[in] myparameter: std::string	const asd::value< std::string >::type& myparameter
[in] myparameter: service(Sensor)	const boost::shared_ptr<SensorInterface>& myparameter
[in] myparameter: service[](Sensor)	const std::vector<boost::shared_ptr<SensorInterface>>& myparameter

Component instance life-time is controlled by the number of references to the instance. Each dependent component instance has a reference to each injected dependency instance. The handwritten client code responsible for building the instance hierarchy using construction parameters also holds a reference to each instance. Control over instance life-time is effectively passed to components to which the instance was injected when the builder releases its reference. The builder maintains in control of instance life-time when the builder keeps its reference.

The code generated for "Singleton" and "Multiple" ASD components using construction parameters is identical. The handwritten

client building the instance hierarchy should call <component>::ReleaseInstance() on each singleton component it instantiates when the singleton components are generated following the regular ASD construction mechanism.

Thus, the builder is responsible for the life-time of all singleton instances in the instance hierarchy.

Justification: <component>::ReleaseInstance() is not called on dependencies of components using construction parameters. A component using construction parameters knows only the dependency interfaces. Consequently it cannot call the ReleaseInstance () associated with a specific implementation of that interface.

Note that the need for use of the singleton pattern is greatly reduced when construction parameters are used to facilitate the use of multi-client interfaces.

verum

Home Product Technology Resources Training Purchase Company

OS support - supported compilers and boost versions for C++

For C++ the underlying OS support is ensured by the usage of boost primitives www.boost.org.

These are the supported / not-supported (boost-version,compiler) pairs:

✓ - supported, ✗ - not supported

Boost version	gcc 4.3.6	gcc 4.4.6	gcc 4.5.3	gcc 4.6.1	Visual Studio 2008	Visual Studio 2010
boost_1_42_0	✓	✓	✓	✓	✓	✗
boost_1_43_0	✓	✓	✓	✓	✓	✓
boost_1_44_0	✓	✓	✓	✓	✓	✓
boost_1_45_0	✓	✓	✓	✓	✓	✓
boost_1_46_1	✓	✓	✓	✓	✓	✓
boost_1_47_0	✓	✓	✓	✓	✓	✓

Note: Due to a change in the cygwin library set, applications based on C++ code generated with the ASD:Suite and compiled and built with gcc 4.6.1 do not execute under cygwin 1.7.18-1 (or newer).

Trace outputs in C++: content and customization

The ASD Runtime for C++ collects the following information with the generated trace statements :

- component identifier
- state identifier
- interface identifier
- trigger identifier
- function
- file
- line

This information is gathered together by an instance of the asd::diagnostics::info struct which is passed to the overridable trace handler. The pre-processor is used to collect function, file and line information when available.

Depending on the environment (OS, compiler, libraries) used to deploy the generated code, thread identifier and/or call stack, etc. could be additionally traced by supplying a custom trace handler. The default trace handler (asd::diagnostics::handler) uses the "cout" standard library facility to write this information to standard out.

The following code snippet is an example of trace handler customization/redefinition and set-up:

```
#include "diagnostics.h"
#include <string.h>
#include <iostream>


using namespace std;


static void my_tracehandler (constasd::diagnostics::info& i)
{
    using namespaceasd::diagnostics;
    std::cout << (i.type == info::enter ? "-->": i.type == info::exit ? "<--" : "illegal")
          << " " << i.component // contains the component name
          << " " << i.state // contains the current state of the component
          << " " << i.channel // contains the interface channel name
          << " " << i.stimulus // contains the stimulus name
          << " " << i.file // contains the file name
          << " " << i.member // contains the member function of the class
          << " " << i.line // contains the line number in the file
          << std::endl;
}

int main ()
{
    asd::diagnostics::handler old_handler;

    // Install my own handler
    old_handler = asd::diagnostics::set_trace(my_tracehandler);

    // Do stuff
    // ...

    // Re-install old handler
    asd::diagnostics::set_trace(old_handler);

    return 0;
}
```

System failure in C++: content and customization

A generated ASD component can end up in an illegal state when the implementation of one of its handwritten foreign components that is not compliant with its interface specification sends events to ASD components. For example, a user interface triggers an API call that is not expected (i.e. specified as illegal in the corresponding state in the interface model) or a handwritten component used by an ASD component triggers a notification that is not expected (i.e. specified as illegal in the corresponding state in thein the design model).

Because the ASD components might be in an unexpected (illegal) state, the correctness of further execution is no longer guaranteed and default behaviour in ASD is then termination of the application. In ASD an illegal handler is responsible for terminating of the application: in C++ this is done by a call to std::abort().

The redefinition/customization of the illegal handler and its set up is similar to customizing and setting up a custom trace handler.
Note: the function used to install the illegal trace handler is: asd::diagnostics::set_illegal.

The ASD Runtime for C#

The following files make up the ASD Runtime package for C#

File Name	Description
Channels.cs	A set of ASD dedicated interfaces, currently for *Single-threaded* execution model
Singlethreaded.cs	Interface and implementation of *Single-threaded* execution model
MultiThreaded.cs	Interface and implementation of *Multi-threaded* execution model
Diagnostics.cs	Interface and default implementation of the diagnostics tracing and illegal handlers
DataVariable.cs	Utility for data variables
PassbyValue.cs	Utility to guarantee pass by value semantics for parameter passing
Ucv.cs	Utility for used service reference operators

The C# ASD Runtime files support the generated ASD C# code and provide an interface to handwritten code. The sections of the C# ASD Runtime files that support ASD generated code have a specific format and may NOT be edited and the respective files should NOT be included in any handwritten code. These sections in the files are prefaced by "namespace asd_<#>" where <#> is a specific build version number. Example: namespace asd_31362.

Parameter definition and parameter passing in C#

The declaration of parameters and how they are passed on as arguments in an ASD design model is described in the user manual. This section describes the language specific details for C#. Foreign code that interfaces with the ASD generated code must adhere to the semantics as defined here.

The type of a parameter can be any C# value base type or reference type. There are no type checks for parameters. It is your job to ensure that the specified types are correct. For example, all parameters that are used with a certain data variable must have the same type. Errors caused by wrong type specifications appear as compilation problems when you attempt to compile the generated code in your development environment. In case you specified parameters with a user defined type, you have to ensure that the respective type definition is accessible from within the generated code.

Parameter passing in C#

For [in] parameters the following holds:

- All value type parameters are passed by value. All reference type parameters are passed by reference.
- All parameters are immutable. All ASD generated code complies with this assumption. You are required to ensure that all user components also comply.
- The parameters of notification events are copied to avoid common data-racing issues that can occur in Multi-threaded environments. Their types must therefore be copyable.

For [out] and [inout] parameters the following holds:

- All [out] parameters are passed as C# "out" parameters and all [inout] parameters are passed as C# "ref" parameters.
- The ASD:Suite requires that the first use of an [out] parameter of an application call event used as trigger is as an [out] parameter of an application call event used in a sequence of actions. Subsequent uses of the respective parameter can be as [out] or [inout] parameter of an application call event used in a sequence of actions.
- If the first use of an undecorated (i.e. no storage specifier attached) parameter name within a rule case is as an [out] parameter of an application call event used in a sequence of actions, a local variable of the same name is introduced.
- The ASD:Suite ensures that all [out] parameters are guaranteed to have a type-specific default value assigned to them (using the C# default operator) when this is not done in the ASD model. Assigning "default" to references does not create a default constructed object of the type but instead assigns "null" to the reference.
- The ASD:Suite assumes that "null" can be passed as any in, out or inout parameter and generates code accordingly. All handwritten components interfacing with ASD components must also assume this.
- All [inout] arguments must be initialised (i.e. they must have a pre-existing value).

For the data variables and parameters of a notification event, the following holds:

- By default all parameters and data variables be of a value type or user defined type which is cloneable and supports the C# extension mechanism. Storing and retrieving reference types to and from the data variables is performed by cloning the objects. The value types are copied by value. The value retrieved from a data variable which has never been assigned a value will be the type's C# default value. For all reference types this is "null"; for all value types it is 0.
- The default behaviour can be overridden with the "No Parameter Cloning" option. When this option is enabled, data variables and parameters in the event queue will be assigned by reference instead of the default cloning behaviour.
- The type of an data variable or an in parameter of a notification event must be a value type or a cloneable reference type. unless the no parameter cloning option is enabled. A cloneable reference type is one which has all of the following properties:
  - It must inherit from the "Icloneable" interface.
  - It must provide an implementation for the "Clone" method. The generated C# code uses the C# extension mechanism to provide a "PassByValue" implementation for all value types and reference types. For reference types, this is implemented in terms of calling the "Clone" method. For value types, this simply returns the value.

The ASD generated C# code will not dereference "null" references in the following cases:

- When using an [in] parameter in an application call event, or notification event, used as trigger.
- When using an [in] parameter in an application call event, or notification event, used in a sequence of actions.
- When receiving an out value from an application call event used in a sequence of actions.
- When storing to a data variable.
- When retrieving from a data variable which has not been assigned a value and is therefore a "null" reference.

Component integration in C#

This section describes how handwritten client components and/or handwritten used components can be made to work with the ASD generated components. The best way to go about this is to make use of the stub generator that is present in the ASD: Suite ModelBuilder. The paragraphs below describe how to use the stub generator effectively to quickly generate code for respecively the handwritten client component(s) and handwritten used component(s). In addition this section also described how component creation works with construction parameters

Stub generation for calling components

To generate code for a handwritten **client** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C# in this case) as well as the version.
4. Select the "Client Stub" radio button and fill in the following options:
   - Fill in the name of the **stub** component to be generated in the construction parameter field. This name is the name of the **calling** component that is created for you including all queue plumbing, notification registration, etc.
   - Fill in the name of the **used** component that is to be referred in the construction parameter field. This name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
     Tip: Load the design model that uses the interface model for which you want to generate a stub. Select the interface model and right-click it; the ModelBuilder has now filled in the name of the used component for you as it matches the name of the construction parameter as found in the service references tab.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the client component for C# can be found here.

Stub generation for used components

To generate code for a handwritten **used** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C# in this case) as well as the version.
4. Select the "Used Component Stub" radio button and fill in the following options:
   - Fill in the name of the component to be generated in the construction parameter field. Note that this name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
   - Select the component type in the dropdown menu.
   - Select "Generate debug info" if you want trace statement to be included in your stub code.
   - Select "Generate synchronization primitives" if you want your functions to contain mutexes in order to be thread-safe. This is typically not needed in the Single-threaded model. In the Multi-threaded this can be needed when the stub has multiple clients and contains, for example, data that requires thread-safe access.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the used component for C# can be found here.

Component creation using component construction parameters

The typical use-cases for designing ASD components using component construction parameters are systems in which it is undesirable or impossible at design time to specify which implementations will fulfil the dependencies or when the design relies on multi-client interfaces on non-singleton components, i.e. construction-time diversity or shared multiple:

- **Construction-time diversity** can be used when the choice between different implementations of components (or sub-systems) depends on system configuration and can be resolved before ASD components are instantiated. For example, the implementation of a protocol handler will be instantiated to match system configuration settings specifying which communication protocol should be used or which camera driver implementation is selected to match the actual camera hardware detected.
- **Multiple shared instances** of an ASD component can exist in a programs' address space if it has component type Multiple. When two different component instances need to use the same instance of a used service that used service has a multi-client interface. Component construction parameters should be used to bind the different client components to the shared used service instance when the used service implementation cannot be made Singleton.

Note: See "Specifying the component type" for details about component types, Singleton or Multiple.

When you define construction parameters in the Model Properties of a design model, they end up in the signature of the GetInstance () method. You can read more about construction parameters in "Defining construction parameters".

Construction parameter	GetInstance parameter
[in] myparameter: string	string myparameter
[in] myparameter: service(Sensor)	SensorInterface myparameter
[in] myparameter: service[](Sensor)	List<SensorInterface> myparameter

Component instance life-time is controlled by the number of references to the instance. Each dependent component instance has a reference to each injected dependency instance. The handwritten client code responsible for building the instance hierarchy using construction parameters also holds a reference to each instance. Control over instance life-time is effectively passed to components to which the instance was injected when the builder releases its reference. The builder maintains in control of instance life-time when the builder keeps its reference.

The code generated for "Singleton" and "Multiple" ASD components using construction parameters is identical. The handwritten

client building the instance hierarchy should call <component>::ReleaseInstance() on each singleton component it instantiates when the singleton components are generated following the regular ASD construction mechanism.

Thus, the builder is responsible for the life-time of all singleton instances in the instance hierarchy.

Justification: <component>::ReleaseInstance() is not called on dependencies of components using construction parameters. A component using construction parameters knows only the dependency interfaces. Consequently it cannot call the ReleaseInstance () associated with a specific implementation of that interface.

Note that the need for use of the singleton pattern is greatly reduced when construction parameters are used to facilitate the use of multi-client interfaces.

Supported compilers and execution platforms for C#

The C# source code generated using the ASD:Suite is ECMA (European Computer Manufacturers Association) compliant C#.

The supported compilers must be able to compile and build C# code conforming to the C# 3.0 language specification.

Therefore, the following compilers are supported: Microsoft Visual Studio 2008 and 2010.

Note: Build and execution of C# code generated using the ASD:Suite requires a Windows .NET 2.0 compatible runtime.

Trace outputs in C#: content and customization

In C#, the default .NET System.Diagnostics.Trace facility is used to allow redirection of trace statements to file, standard out, or any other output device.

The default implementation retrieves the name of the calling function, the file and the line number of where the trace statement is executed by making use of the .NET library.

The following code snippet is an example of trace handler customization:

```
public void traceHandler (diagnostics.info i)
{
    if (i.type == asd.diagnostics.type.illegal)
    {
       MessageBox.Show(
           "Program termination due to illegal event:\n"
            + "\n"
            + " EVENT:\t\t\"" + i.channel + "." + i.stimulus + "\"\n"
            + " STATE:\t\t\"" + i.state + "\"\n"
            + " COMPONENT:\t\""+ i.component + "\"\n"
            + "\n"
            + "Has the design been verified using the ASD:Suite?",
           "ASSERTION FAILURE",
           MessageBoxButtons.OK,
           MessageBoxIcon.Error
        );
       Application.Exit();
    }
    else
    {
        logWindow
        .showInfo ((i.type == asd.diagnostics.type. enter ? "-->"
        : "<--")
        + " "
        + i.component
        + (i.state == "" ? " Constructor"
        : "." + i.state + "." + i.channel + "." + i.stimulus)
        + Environment.NewLine);
    }
}
```

where logWindow is a Form defined as follows:

```
private class LogWindow : Form
{
    private TextBox textBox = null;

    public LogWindow(String title, int width, int height)
    {
       this.Text = title;
       this.ClientSize = new System.Drawing.Size(width, height);
       this.textBox = new TextBox();
       this.textBox.Multiline = true;
       this.textBox.ScrollBars = ScrollBars.Vertical;
       this.textBox.Dock = DockStyle.Fill;
       this.textBox.ReadOnly = true;
       this.Controls.Add (this.textBox);
       this.Show();
    }

    public void showInfo(String data)
    {
        textBox.AppendText(data);
    }
}
```

Use the following construction to install the customised trace handler:

```
asd.diagnostics.set_trace(this.traceHandler);
```

System failure in C#: content and customization

A generated ASD component can end up in an illegal state when the implementation of one of its handwritten foreign components that is not compliant with its interface specification sends events to ASD components. For example, a user interface triggers an API call that is not expected (i.e. specified as illegal in the corresponding state in the interface model) or a handwritten component used by an ASD component triggers a notification that is not expected (i.e. specified as illegal in the corresponding state in thein the design model).

Because the ASD components might be in an unexpected (illegal) state, the correctness of further execution is no longer guaranteed and default behaviour in ASD is then termination of the application. In ASD an illegal handler is responsible for terminating of the application: in C# this is done by a call to System.Diagnostics.Process.GetCurrentProcess().Kill().

The redefinition/customization of the illegal handler and its set up is similar to customizing and setting up a custom trace handler.
Note: the function used to install the illegal trace handler is: asd.diagnostics.set_illegal().

The ASD Runtime for Java

The following files in the ASD Runtime package for Java may be used by the customer in the handwritten code. These files may not be changed by the customer. They are located in the Java runtime package under namespaces like com.verum.asd.runtime.*, where * can be "channels", "diagnostics" and "parameters".

File Name	Package	Description
SingleThreaded.java ISingleThreaded.java	channels	A set of ASD dedicated interfaces for the *Single-threaded* execution model
Diagnostics.java DiagnosticsDefaultTraceHandler.java DiagnosticsInfo.java DiagnosticsTraceListener.java	diagnostics	Interface and default implementation of the diagnostics tracing and illegal handlers
BooleanHolder.java ByteHolder.java CharHolder.java DoubleHolder.java FloatHolder.java Holder.java IntHolder.java LongHolder.java ObjectHolder.java ShortHolder.java	parameters	Utility classes for in, out and in/out parameter passing and data variables

The following files in the ASD Runtime package for Java are for ASD internal use only. These files may NOT be changed or included by the customer. They are located in the Java runtime package under namespace "com.verum.asd.runtime_<revisionID>", where "revisionID" is a number.

Package File Name	Description
SingleThreadedContext.java SingleThreadedContextNoDpc.java	Interface and implementation of *Single-threaded* execution model
MultiThreadedContext.java MultiThreadedContextNoDpc.java MultiThreadedDpc.java IFunctor.java	Interface and implementation of *Multi-threaded* execution model
ITransfer.java PassByValue.java	Utility to guarantee pass by value semantics for parameter passing
VariableUCV.java FixedUCV.java NullUCV.java	Support for used service reference operators

Home Product Technology Resources Training Purchase Company

Parameter definition and parameter passing in Java

The declaration of parameters and how they are passed on as arguments in an ASD design model is described in the user manual. This section describes the language specific details for Java. Foreign code that interfaces with the ASD generated code must adhere to the semantics as defined here.

The type of a parameter can be any Java value base type or reference type. There are no type checks for parameters. It is your job to ensure that the specified types are correct. For example, all parameters that are used with a certain data variable must have the same type. Errors caused by wrong type specifications appear as compilation problems when you attempt to compile the generated code in your development environment. In case you specified parameters with a user defined type, you have to ensure that the respective type definition is accessible from within the generated code.

Parameter passing in Java

For [in] parameters the following holds:

- The type of an [in] parameter can be any valid Java primitive type (boolean, char, int, long, byte, short, float, double) or a class. All primitive type [in] parameters are passed by value and all [in] parameters that have a class as type are passed by reference. Note that array types "int[]", "MyType[]", etc. are not supported. Use java.util.ArrayList instead.

For [out] and [inout] parameters the following holds:

- All [out] parameters have "out" semantics and pre-existing values can not be assumed or used. The ASD:Suite uses patterns that assume that pre-existing values of [out] parameters can not be assumed by you as a user of the ASD:Suite. Specifically, it is assumed that handwritten components always avoid partial updates (e.g. assigning values to some but not all data members of an output object on the assumption that the others have valid values) and never use pre-existing values as input.
- The ASD:Suite requires that the first use of an [out] parameter of an application call event used as trigger is as an [out] parameter of an application call event used in a sequence of actions. Subsequent uses of the respective parameter can be as [in], [out] or [inout] parameter of an application call event used in a sequence of actions.
- If the first use of an undecorated (i.e. no storage specifier attached) parameter name within a rule case is as an [out] parameter of an application call event used in a sequence of actions, a local variable of the same name is introduced.
- All [inout] arguments must be initialised (i.e. they must have a pre-existing value).
- To model [out] and [inout] parameters, the ASD Java runtime contains the classes BooleanHolder, CharHolder, IntHolder, LongHolder, ByteHolder, ShortHolder, FloatHolder and DoubleHolder to model primitive type [out]/[inout] parameters. The generic type ObjectHolder<T> is used to model [out]/[inout] parameters that have a class as type. Each of the "Holder" classes has a "get" and a "set" method for getting/setting the value of the [out]/[inout] parameter.

For the data variables and parameters of a notification event, the following holds:

- All [in] parameters of a notification event must be of a primitive type or have a "clonable" (see below) class as type.
- The parameters that are transferred to and/or from data variables should be either of a primitive type or have a "cloneable" (see below) class as type. Furthermore, these types should have a default (empty) constructor that is used to initialise the data variables at construction time.
- There are no type checks for parameters. It is your job to ensure that the specified types are correct. For example, all parameters that are used with a certain data variable should have the same type. Errors caused by wrong type specifications appear as compilation problems when you attempt to compile the generated code in your development environment.
- The copying of parameters is performed using the standard Java "clone" method. A clonable type should implement the standard Java interface "Cloneable". For a user-defined type MyType, Verum recommends to implement its clone method by means of a protected copy constructor:

```
public class MyType implements Cloneable {
     protected MyType(MyType myType) {
     // Code performing a (preferably deep) copy
     }
     public MyType clone() {
     return new MyType(this);
     }
}
```

- Note: Use the "No Parameter Cloning" option in the Code Generator section of the Model Properties dialog to be able to pass parameters by reference.

Terms of use | Privacy Policy

Component integration in Java

This section describes how handwritten client components and/or handwritten used components can be made to work with the ASD generated components. The best way to go about this is to make use of the stub generator that is present in the ASD: Suite ModelBuilder. The paragraphs below describe how to use the stub generator effectively to quickly generate code for respecively the handwritten client component(s) and handwritten used component(s). In addition this section also described how component creation works with construction parameters

Stub generation for calling components

To generate code for a handwritten **client** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (Java in this case) as well as the version.
4. Select the "Client Stub" radio button and fill in the following options:
   - Fill in the name of the **stub** component to be generated in the construction parameter field. This name is the name of the **calling** component that is created for you including all queue plumbing, notification registration, etc.
   - Fill in the name of the **used** component that is to be referred in the construction parameter field. This name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service. **Tip**: Load the design model that uses the interface model for which you want to generate a stub. Select the interface model and right-click it; the ModelBuilder has now filled in the name of the used component for you as it matches the name of the construction parameter as found in the service references tab.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the client component for Java can be found here.

Stub generation for used components

To generate code for a handwritten **used** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (Java in this case) as well as the version.
4. Select the "Used Component Stub" radio button and fill in the following options:
   - Fill in the name of the component to be generated in the construction parameter field. Note that this name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
   - Select the component type in the dropdown menu.
   - Select "Generate debug info" if you want trace statement to be included in your stub code.
   - Select "Generate synchronization primitives" if you want your functions to contain mutexes in order to be thread-safe. This is typically not needed in the Single-threaded model. In the Multi-threaded this can be needed when the stub has multiple clients and contains, for example, data that requires thread-safe access.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the used component for Java can be found here.

Component creation using component construction parameters

The typical use-cases for designing ASD components using component construction parameters are systems in which it is undesirable or impossible at design time to specify which implementations will fulfil the dependencies or when the design relies on multi-client interfaces on non-singleton components, i.e. construction-time diversity or shared multiple:

- **Construction-time diversity** can be used when the choice between different implementations of components (or sub-systems) depends on system configuration and can be resolved before ASD components are instantiated. For example, the implementation of a protocol handler will be instantiated to match system configuration settings specifying which communication protocol should be used or which camera driver implementation is selected to match the actual camera hardware detected.
- **Multiple shared instances** of an ASD component can exist in a programs' address space if it has component type Multiple. When two different component instances need to use the same instance of a used service that used service has a multi-client interface. Component construction parameters should be used to bind the different client components to the shared used service instance when the used service implementation cannot be made Singleton.

Note: See "Specifying the component type" for details about component types, Singleton or Multiple.

When you define construction parameters in the Model Properties of a design model, they end up in the signature of the GetInstance () method. You can read more about construction parameters in "Defining construction parameters".

Construction parameter	GetInstance parameter
[in] myparameter: String	String myparameter
[in] myparameter: service(Sensor)	final SensorInterface myparameter
[in] myparameter: service[](Sensor)	final java.util.List<SensorInterface> myparameter

Component instance life-time is controlled by the number of references to the instance. Each dependent component instance has a reference to each injected dependency instance. The handwritten client code responsible for building the instance hierarchy using construction parameters also holds a reference to each instance. Control over instance life-time is effectively passed to components to which the instance was injected when the builder releases its reference. The builder maintains in control of instance life-time when the builder keeps its reference.

The code generated for "Singleton" and "Multiple" ASD components using construction parameters is identical. The handwritten

client building the instance hierarchy should call <component>.releaseInstance() on each singleton component it instantiates when the singleton components are generated following the regular ASD construction mechanism.

Thus, the builder is responsible for the life-time of all singleton instances in the instance hierarchy.

Justification: <component>.releaseInstance() is not called on dependencies of components using construction parameters. A component using construction parameters knows only the dependency interfaces. Consequently it cannot call the ReleaseInstance () associated with a specific implementation of that interface.

Note that the need for use of the singleton pattern is greatly reduced when construction parameters are used to facilitate the use of multi-client interfaces.

Supported compilers and execution platforms for Java

The Java source code generated using the ASD:Suite is JDK 6.0 compatible.

The Eclipse development environment or the NetBeans IDE can be used for building and running the generated Java source code.

The ASD:Suite supports JRE 1.6 and only needs Java SE (Standard Edition).

Trace outputs in Java: content and customization

In Java by default the DiagnosticsDefaultTraceHandle is used. This class is part of the ASD Runtime for Java and contains a println to System.out.

To obtain a different trace output you can write a class that implements the DiagnosticsTraceListener class and write your own traceHandler method.

The following code snippet is an example of traceHandler customization:

```
public void traceHandler(DiagnosticsInfo i) {
    if (i.getType() == DiagnosticsInfo.DirType.illegal) {
        JOptionPane.showMessageDialog(contentPane,
            "Program termination due to illegal event:\n\n"
            + "<html><table>" + "<tr>"
            + "<td>EVENT:</td><td>\"" + i.getChannel() + "."
            + i.getStimulus() + "\"</td>" + "</tr>" + "<tr>"
            + "<td>STATE:</td><td>\"" + i.getState()
            + "\"</td>" + "</tr>" + "<tr>"
            + "<td>COMPONENT:</td><td>\"" + i.getComponent()
            + "\"</td>" + "</tr>" + "</table><html>" + "\n\n"
            + "Has the design been verified using the ASD:Suite?",
            "ASSERTION FAILURE", JOptionPane.ERROR_MESSAGE);
        System.exit(1);
    } else {
       logWindow
        .showInfo((i.getType() == DiagnosticsInfo.DirType.enter ? "-->"
            : "<--")
            + " "
            + i.getComponent()
            + (i.getChannel() == "" ? " Constructor" : ".")
            + i.getState() + "." + i.getChannel() + "."
            + i.getStimulus()) + "\n");
    }
}
```

where logWindow is a frame defined as follows:

```
class LogWindow extends JFrame {
  private static final long serialVersionUID = 1L;
  private JTextArea textArea = null;
  private JScrollPane pane = null;

  public LogWindow(String title, int width, int height) {
    super(title);
    setSize(width, height);
    textArea = new JTextArea();
    textArea.setEditable(false);
    pane = new JScrollPane(textArea);
    getContentPane().add(pane);
    setVisible(true);
  }

  public void showInfo(String data) {
    textArea.append(data);
    textArea.setCaretPosition(textArea.getText().length());
    this.getContentPane().validate();
  }
}
```

Note: use the following construction to install the customised trace handler:

```
Diagnostics.setTraceHandler(this);
```

Home Product Technology Resources Training Purchase Company

System failure in Java: content and customization

System failure leads to the termination of the application. In ASD an illegal handler is responsible for terminating the the application in an application specific correct and safe way. If it fails to do so, it will be by returning or by throwing an exception.

Because the ASD components might by in an unexpected (illegal) state, the correctness of further execution is no longer guaranteed. Generated ASD components might end up in an illegal state when handwritten clients or handwritten foreign components send events to ASD components that are not compliant with the interface specification.

This default behaviour can be adapted to your needs with the following requirement on the illegal handler: it should not return to the caller.

In Java the process is halted by an assert(false).

The redefinition/customization of the illegal handler and its set up is similar to customizing and setting up a custom trace handler.
Note: the function used to install the illegal trace handler is
Diagnostics.*setIllegalHandler*().

C versus TinyC

The generated C code essentially follows the same structure as the truly object-oriented programming languages, such as C++, C#, and Java: the generated C code implements the interfaces using v-tables and each method call of an object has as a first argument a pointer to a struct reflecting the "this" pointer for a component instance.

However, in extreme resource constrained environments multiple component instances are hardly used. By using singleton components the internal data of a component can become a local variable of the C source file that implements this component. Consequently, the "this" pointer is no longer needed as all local variables have become "static" within this C source file. Further, if no dynamic bindings are used 'glueing' the components together meaning that all bindings are known at design time (including the bindings for the callbacks), then the linker can be used to resolve and make the bindings.

The implementation of these two optimisations has led to the introduction of the TinyC code generator which is aimed at extreme resource constrained targets such as 8051 microcontrollers. These optimisations are at the expense of restricting the following features in TinyC:

- TinyC does not support the Multi-threaded execution model; it only supports the Single-threaded execution model. Therefore, all limitations of the Single-threaded model apply whenever TinyC is used.
- TinyC does not support the component type being set to multiple. It only supports the component type set to singleton.
- TinyC does not support the use of state variables of type used service reference.
- TinyC does not support the use of service reference construction parameters.
- TinyC does not support the use of service references with cardinality > 1.

All features as present in ASD are fully supported in C.

Note that the generated C code cannot be mixed with the generated TinyC code. From an ASD point of view, C and TinyC are regarded as two different languages. Also note that both C and TinyC differ from the other supported programming languages in one aspect: the generated code is completely static from a memory management point of view. They both comply with MISRA C coding standard.

verum

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

The ASD Runtime for C and TinyC

The ASD Runtime for C and TinyC is the same as there are no essential differences between the two languages. It provides functionality that is shared and used by all generated ASD components such as threading, various utility functions and so on. The ASD Runtime provides this functionality through a number of interfaces that is described in more detail in the Doxygen documentation.

The ASD Runtime is divided in four modules:

1. ASD basic types module (asd);
2. ASD Rte module (rte);
3. ASD Diagnostics module (diagnostics);
4. ASD OSAL module (OSAL);

The ASD basic types module defines general types and functions for ASD. ASD components only use types that are defined in this module and no C built-in types are used. Exceptions to this rule are parameter types. User specific parameter types are defined in header files specified in the model. All types in ASD must have an associated initialiser macro. These macros are used for the initialisation of an ASD component. Not providing an initialiser for a user defined type will result in a compiler error.

The Rte module implements the actual functionality of the runtime environment. This module should not be changed in any way. The reason for this is that it is closely aligned with the execution semantics of ASD on which the (formal) verification is based. Changing the behaviour of the Rte module can break the equivalence between the (formal) verification of the ASD models on one hand and the execution of the generated code on the other. NOTE THAT CORRECT BEHAVIOUR IS NOT GUARANTEED ANYMORE ONCE THE RTE MODULE HAS BEEN CHANGED IN ANY WAY.

The Diagnostic module implements a simple diagnostic facility in ASD. An ASD design model has a property to include trace statements into the generated code. When this trace property is enabled, the code generator will emit entry and exit macros in the generated code. The implementation of these macros can be changed for logging purposes only. Although the macros could change the behaviour of the components because they are inserted into the ASD components, any side effects in the ASD components are not allowed. An example of user-defined implementation is to have empty macro definitions or logging events to a specific logger module.

The ASD OSAL (operating system abstraction layer) module implements a wrapper to operating system calls. The runtime mainly uses threading, synchronization, and time functionality from the operating system. The mapping to a specific operating system is implemented in this module. When support is needed for a new operating system, this module needs to be re-implemented according to the specification in this document. Again no side effects are allowed in the implementation of this module.

The ASD Runtime contains files that are specific for the Multi-threaded execution model and the Single-threaded execution model. It also contains files that are common to both execution models. The following files make up the ASD Runtime package for both C and TinyC:

File Name	Module	Description
asdDefConfig.h	asd	Configuration ASD Runtime
asd.h	asd	General ASD types
asdImpl.h	asd	Interface of ASD types
asd.c	asd	Implementation of ASD types
asdDbg.h	diagnostics	Diagnostics module
asdDbgImpl.c	diagnostics	Implementation of Diagnostics module
asdDbgImpl.h	diagnostics	Interface of Diagnostics module
asdOSAL.h	OSAL	Operating system wrapper
asdOSALImpl.c	OSAL	Implementation of operating system wrapper
asdOSALImpl.h	OSAL	Interface of operating system wrapper
asdRte.h	rte	Interface of ASD Runtime (Common)
asdRteBuffer.c	rte	Implementation of ASD Runtime (Multi-threaded)
asdRteBufFifo.c	rte	Implementation of ASD Runtime (Common)
asdRteBroadcast.c	rte	Implementation of ASD Runtime (Multi-threaded)
asdRteContextMT.c	rte	Implementation of ASD Runtime (Multi-threaded)
asdRteDpcMT.c	rte	Implementation of ASD Runtime (Multi-threaded)

asdRteUcv.c	rte	Implementation of ASD Runtime (Common)
asdRteISThreadInterface.h	rte	Interface of ASD Runtime (Single-threaded)
asdRteContextSThread.c	rte	Implementation of ASD Runtime (Single-threaded)

Parameter definition and parameter passing in C and TinyC

The declaration of parameters and how they are passed on as arguments in an ASD design model is described in the user manual. This section describes the language specific details for C and TinyC. It also explains how to include data into your (hand-written) components and how to ensure that this data is then properly initialised.

Parameter passing in C and TinyC

For [in] parameters in C and TinyC the following holds:

- An [in] parameter can be of any C type or user defined type. Note that the designer must ensure that such user defined types are accessible (to the compiler) using the import/include field.
- An [in] parameter is always passed by value for all events. This implies that a copy of the argument is made. The C assignment statement is used to perform the copy-action. The C assignment statement can make shallow copies, but making deep copies - also copying referenced values - is not supported.
- An [in] parameter in the generated C are exactly as they have been specified in the ASD interface models. It is possible to declare them as "const" in the ASD interface models. For example:

  ```
  writeBuf ([in] buf:const char*, [in] c:const char) : void
  ```

  In such case, the "const" is passed on the C/TinyC code generator and used accordingly. Note that there is no checking on correct usage of const parameters in ASD which could lead to compiler errors.
- Pointers are allowed as [in] parameters, but the lifecycle management of the referenced values is not be managed by ASD.

For [out] and [inout] parameters in C and TinyC the following holds:

- An [out] or [inout] parameter can be of any C type or user defined type. Note that the designer must ensure that such user defined types are accessible (to the compiler) using the import/include field.
- An [out] parameter has true out-semantics and pre-existing values are not available anymore. For handwritten components it is assumed that no partial updates have been performed.
- All [out] and [inout] parameters are passed via pointer dereferencing. The only way to pass [out] or [inout] parameters in C is to pass a pointer to a variable instead of passing the actual variable. Therefore the specified type "T" of an [out] or [inout] parameter will become "T*" in the generated C code.
- The parameter type is - besides the addition of the pointer indirection - untouched. It is the responsibility of the designer that parameter types are not const and the associated assignment to an [out] or [inout] parameter will not result in a compiler error.

For the data variables the following holds:

- Because data variables are being read and written to, the data variable **type** cannot be const.
- Data variables are being initialised to zero during construction of the component.
- Storing and retrieving the data variables takes place via the C assignment operator using the semantics as defined above.

How to include data into your components

It is possible to store data in your (handwritten) components, and it is best to use the stub-generator to generate the infrastructure of the used component, after which it is trivial to include additional data.

The <component>ComponentImpl.h header file where <component> is the name of the component contains a separate structure called <component>Data acting as a container for the component's data. This structure has a corresponding initialiser macro called <COMPONENT>DATA_INITIALIZER where <COMPONENT> is the name of the component is in capital case. When adding data, you must add the corresponding type and member to the aforementioned structure including the respective initialiser to the aforementioned macro. When the data is of a user defined type, you have to provide the initialiser macro declaration in the user specific include file, following the aforementioned naming convention. Omitting this initialiser macro can lead to compiler errors on the component instantiation and initialisation.

Initially, it is assumed that such handwritten component contains no data, and this is achieved by commenting out the following compiler directive: <COMPONENT>_HASIMPLSTRUCT where <COMPONENT> is the name of the component is in capital case (also in the aforementioned header file). So to use data, it is necessary to uncomment this compiler directive.

An example of how to include data for C can be found here, whereas for TinyC it can be found here.

Component integration in C and TinyC

This section describes how handwritten client components and/or handwritten used components can be made to work with the ASD generated components. The best way to go about this is to make use of the stub generator that is present in the ASD: Suite ModelBuilder. The paragraphs below describe how to use the stub generator effectively to quickly generate code for respecively the handwritten client component(s) and handwritten used component(s). In addition this section also described how component creation works with construction parameters

Stub generation for calling components

To generate code for a handwritten **client** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C or TinyC in this case) as well as the version.
4. Select the "Client Stub" radio button and fill in the following options:
   - Fill in the name of the **stub** component to be generated in the construction parameter field. This name is the name of the **calling** component that is created for you including all queue plumbing, notification registration, etc.
   - Fill in the name of the **used** component that is to be referred in the construction parameter field. This name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
     Tip: Load the design model that uses the interface model for which you want to generate a stub. Select the interface model and right-click it; the ModelBuilder has now filled in the name of the used component for you as it matches the name of the construction parameter as found in the service references tab.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the client component for C can be found here, whereas for TinyC it can be found here.

Stub generation for used components

To generate code for a handwritten **used** component, the following steps should be carried out:

1. Open the respective ASD interface model in the ASD:Suite ModelBuilder.
2. Start stub generator by right-clicking the interface model in the Model Explorer window or by selecting it in the Code Generator menu.
3. Select the language (C or TinyC in this case) as well as the version.
4. Select the "Used Component Stub" radio button and fill in the following options:
   - Fill in the name of the component to be generated in the construction parameter field. Note that this name should be the same as the name of the construction parameter in those ASD design models where this ASD interface model is referred to as a used service.
   - Select the component type in the dropdown menu. Note that for TinyC only the type "singleton" is supported.
   - Make sure that the option "Generate proxy class for each interface" is deselected since this is not supported for C nor TinyC.
   - Select "Generate debug info" if you want trace statement to be included in your stub code.
   - Select "Generate synchronization primitives" if you want your functions to contain mutexes in order to be thread-safe. This is typically not needed in the Single-threaded model. In the Multi-threaded this can be needed when the stub has multiple clients and contains, for example, data that requires thread-safe access.
5. Select the output path where the generated stub should go to.
6. If desired, select "Save Settings" to remember these settings for a next time that stub code needs to be generated.
7. Finally, press "OK" to start the stub generation after which the files are generated. You need to remove the "_tmpl" from the file extension after which the files are ready for use. When you already have generated stub code before, you can use a merge tool to compare and merge the generated stub code.

The generated stub code for the used component for C can be found here, whereas for TinyC it can be found here.

Component creation using component construction parameters

The typical use-cases for designing ASD components using component construction parameters are systems in which it is undesirable or impossible at design time to specify which implementations will fulfil the dependencies or when the design relies on multi-client interfaces on non-singleton components, i.e. construction-time diversity or shared multiple:

- **Construction-time diversity** can be used when the choice between different implementations of components (or sub-systems) depends on system configuration and can be resolved before ASD components are instantiated. For example, the implementation of a protocol handler will be instantiated to match system configuration settings specifying which communication protocol should be used or which camera driver implementation is selected to match the actual camera hardware detected.
- **Multiple shared instances** of an ASD component can exist in a programs' address space if it has component type Multiple. When two different component instances need to use the same instance of a used service that used service has a multi-client interface. Component construction parameters should be used to bind the different client components to the shared used service instance when the used service implementation cannot be made Singleton.

Note: See "Specifying the component type" for details about component types, Singleton or Multiple.

When you define construction parameters in the Model Properties of a design model, they end up in the signature of the GetInstance () method. You can read more about construction parameters in "Defining construction parameters".

Construction parameter	GetInstance parameter
[in] myparameter: int	int myparameter
[in] myparameter: service(Sensor)	SensorInterface_Intf* myparameter
[in] myparameter: service[](Sensor)	SensorInterface_Intf* myparameter[], asdUint myparameter_array_size

Component instance life-time is controlled by the number of references to the instance. Each dependent component instance has a reference to each injected dependency instance. The handwritten client code responsible for building the instance hierarchy using construction parameters also holds a reference to each instance. Control over instance life-time is effectively passed to components to which the instance was injected when the builder releases its reference. The builder maintains in control of instance life-time when the builder keeps its reference.

The code generated for "Singleton" and "Multiple" ASD components using construction parameters is identical. The handwritten client building the instance hierarchy should call <component>::ReleaseInstance() on each singleton component it instantiates when the singleton components are generated following the regular ASD construction mechanism.

Thus, the builder is responsible for the life-time of all singleton instances in the instance hierarchy.

Justification: <component>::ReleaseInstance() is not called on dependencies of components using construction parameters. A component using construction parameters knows only the dependency interfaces. Consequently it cannot call the ReleaseInstance () associated with a specific implementation of that interface.

Note that the need for use of the singleton pattern is greatly reduced when construction parameters are used to facilitate the use of multi-client interfaces.

Supported compilers and execution platforms for C and TinyC

These are the officially supported compilers for the C/TinyC source code generated using the ASD:Suite:

- Microsoft Visual Studio 2008
- gcc 4.3.2

Other compilers and/or versions of compilers are known to work as well but they are not part of the release suite that is used to validate the generated code in combination with the ASD Runtime.

Custom OSAL support for C and TinyC

When needed, you can create a platform specific **OSAL** (Operating System Abstraction Layer). The OSAL module of the ASD Runtime acts as a wrapper to the operating system calls and mainly consists of threading, synchronization, and timing functionality. An implementation for POSIX is provided by default. There is also a Nullos implementation that is used for platforms without an operating system.

After downloading the ASD runtime you can copy the provided POSIX or Nullos implementation files to a separate directory and make the platform specific changes in the OSAL as needed for your operating system. The existing OSAL model containing the POSIX and Nullos implementation should not be misused to change the modelled (runtime) behaviour. This breaks the equivalence in behaviour between what is verified on one hand with the execution of the generated code using the ASD Runtime on the other.

Note: Any OSAL implementation need to adhere to the interface description that is specified in the Doxygen documentation.

Trace outputs in C and TinyC: content and customization

The Diagnostic module of the ASD runtime provides macros that are inserted in the generated code if the debug flag is turned on.

Currently there are three macros involved: ASDDBG_ENTRYTRACE, ASDDBG_EXITTRACE and ASDDBG_ASSERT. The default implementation gathers function, file and line number.

The implementation of these macros can be changed in the platform specific OSAL.

System failure in C and TinyC: content and customization

A generated ASD component can end up in an illegal state when the implementation of one of its handwritten foreign components that is not compliant with its interface specification sends events to ASD components. For example, a user interface triggers an API call that is not expected (i.e. specified as illegal in the corresponding state in the interface model) or a handwritten component used by an ASD component triggers a notification that is not expected (i.e. specified as illegal in the corresponding state in thein the design model).

Because the ASD components might be in an unexpected (illegal) state, the correctness of further execution is no longer guaranteed and default behaviour in ASD is then termination of the application. In ASD an illegal handler is responsible for terminating of the application: in both C and TinyC this is done by a call to asd_illegal() (asd.h). The asd_illegal() itself cannot be re-implemented by a platform specific version. The asd_illegal() function is never allowed to return such that execution can continue.

This default behaviour can be adapted to your needs with the following requirement on the illegal handler: it should **never** return to the caller.

Carry out the following steps to create your own illegal handler:

1. Copy the file "asdDefConfig.h" to a new file named "asdConfig.h".
2. Edit the file "asdConfig.h" and add the compile directive:
   ```
   #define ASDIMPL_ILLEGAL(msg)
   ```
3. Provide the corresponding body for the directive, such as, for example:
   ```
   #define ASDIMPL_ILLEGAL(msg) {for(;;);}
   ```
4. Re-compile with the option "-DASD_HAVE_CONFIG_H".

The user can also register an illegal callback function via the asd_register_illegalCB() function. This function is called by the default asd_illegal() implementation such that the client is informed about the erroneous situation and can take action like resetting the system, informing the user, or make sure that hardware components are set in a secure condition.

RAM and ROM footprint optimizations

In some cases it can be possible that the generated code does not fit into RAM and/or ROM. This section presents some suggestions to reduce the footprint even further in the generated C and TinyC code:

1. **Switch off trace statements.**
   By default the ASD components are generated with debug information enabled. In the generated C and TinyC code this implies that macro's are present and inserted resulting in trace output being generated during run-time execution. These macro's can be switched by either simply not generating them anymore or by disabling them using a compile-time directive. Carry out the following steps to disable this compile-time directive:
   - Copy the file "asdDefConfig.h" to a new file named "asdConfig.h".
   - Edit the file "asdConfig.h" and comment out the compile directive:
     `#define ASD_ENABLE_DEBUG`
   - Re-compile with the option "-DASD_HAVE_CONFIG_H".

2. **Switch off assert on re-entrancy check.**
   An application interface function can only exit the component after a return event (void reply or valued return) has occurred. In case of the Multi-threaded execution model, the application interface thread can suspend until the dpc thread issues a return event. In case of the Single-threaded execution model, the return event is always issued on the same execution thread. This behaviour is checked by the model checker. Therefore, a return event has always been issued before the execution thread finishes the application interface function call. By default, the ASD Runtime will assert on this missing return event (i.e. another application interface function tries to re-enter the component whilst the previous function has not returned yet). This assert is to aid users to check correct behaviour while developing/testing their applications. This assert can be optimized out by enabling a compile-time directive. Carry out the following steps to enable this compile-time directive:
   - Copy the file "asdDefConfig.h" to a new file named "asdConfig.h".
   - Edit the file "asdConfig.h" and uncomment the compile directive:
     `#define ASDIMPL_NO_ILLEGAL_ON_RETURNEVENT`
   - Re-compile with the option "-DASD_HAVE_CONFIG_H".

3. **Reduce the queue size.**
   The queue size is set to 7 by default and is allocated statically due to static memory management in C and TinyC. Often, especially in the Single-threaded execution model, a queue size of 2 or even 1 is sufficient. Carry out the following steps to reduce the queue size:
   - Open the respective design model in the ASD:Suite ModelBuilder.
   - Go to the model properties of the model (by pressing alt-F7).
   - Select the verification properties.
   - Now the notification event queue size is visible and often has the default value of 7. Enter a value that seems suitable and press "OK".
   - First re-verify the model to ensure that there is no queue full (by pressing ctrl-F5). If there is a verification error reporting a queue-full, then apparently you need to increase the queue size somewhat until the verification error reporting the queue-full is gone. Reversely, if there is no verification error reporting a queue-full, one might consider reducing the size even further.
   - Once verification is succesful, re-generate the code (by pressing F7 or ctrl-F7).
     Warning: During execution the (static) queue can overflow if verification was performed with yoking enabled.

   Tip: a queue size of 1 also saves on code. It can pay off to change your models such that a queue size of 1 is sufficient.

4. **Avoid pointers.** One should be careful with using pointers on 8-bit microcontrollers. This results in pointer chasing which has negative impact on code size.

5. **Apply proper abstractions and component responsibilities.**
   The above are relatively simple and straightforward steps to reduce RAM and/or ROM footprint. However, the way the ASD models are constructed can also have an impact on RAM and/or ROM size. Typically, the better the responsibilities are distributed amongst the (ASD) components the more benefical this can be for RAM and/or ROM usage. Further, applying proper abstractions in your interfaces can also help in 'normalizing' your ASD models and hence avoiding code duplication. For example, consider three events called "ev1", "ev2", and "ev3" that are made available on an interface that some ASD component is using. When the response to these three events is the same (or can be made the same), then this leads to code duplication and therefore higher ROM usage. This can be eliminated by folding these three events onto a single new one that then eliminates this code duplication. Often, a (positive) side effect is that responsibilities have been divided better resulting in more maintainable ASD models.

How to integrate the commandline tools in your development environment

Code integration into your application, supported by your build environment, can start after you have built your models, verified them, and generated code from them.

Read the following sections for build integration related information supported by the ASD commandline tools.

Commandline tools

The commandline tools that are typically used for integration into your build environment are AsdVerify and AsdGenerate.

The AsdVerify can be used to verify models in a batch; it is optimised such that if the models are not (semantically) changed, the actual verification is not performed and only the verification status is reported. Using the verbosity levels, it is possible to report the results of each individual check as well as the overall status.

The AsdVerify can also be used to both query models and code. In case the query is performed on models, it can report the results of each individual check and/or the overall verification status. In case the query is performed on code, it can report if the code was indeed generated from a verified model and has not been modified by hand.

The AsdGenerate can be used to generate source code from models in a batch; like the AsdVerify it is optimised such that if the models are not (semantically) changed, the original source code file is not overwritten (if present). This is especially useful in build environments where make-like tools are used to check dependencies and to build only what is absolutely necessary. Another reason to simply generate all code (which will not overwrite unchanged files) from within a make-like build system is that there is no dependency generator from models to code as this requires round-tripping to the ASD hosted services; to do this in a reliable way is not possible.

Note that for the actual verification and source code generation, the AsdVerify and the AsdGenerate will require a connection to the ASD hosted services. The possibility to query models and/or code does not require a connection.

verum

Integration

The commandline utilities can be used in an automated build environment to build your applications. This section describes some topics to keep in mind when integrating these utilities into your environment.

Firstly, when using the commandline utilities to verify and/or to generate code, then the build system must have access to the internet in order to reach the verification engines and the code generators as these are hosted.

Secondly, when using **make** as the utility to compile and link your applications, then one should know that make is designed to minimize the build time by only compiling and/or linking what is absolutely necessary. It uses the principle of targets and dependencies to realise this. The commandline utilities, such as AsdVerify and AsdGenerate are designed and built to support this as much as possible; for example, AsdGenerate does not overwrite the source file(s) if there are no changes (in other words, the timestamps of the respective source file(s) do not change and therefore unnecessary compilation is prevented). Consequently, one has to be extremely careful in constructing the makefile and the dependencies between the various targets. A dependency statement as shown below will not work in a makefile:

```
serverComponent.c: server.dm
  AsdGenerate -g -l cpp -v 9.2.3 server.dm
```

When the ASD design model is 'touched', its timestamp will change in the file system causing the makefile to trigger the dependency as shown above. However, as the AsdGenerate now notices that the source has not changed, it will **not** overwrite the source file. As a result, nothing happens and a subsequent call to make will result in triggering this dependency again.

Finally, also note that undoing changes by hand (instead of using the undo functionality) can have similar strange effects in your build environment; especially when it is based on make. For example, manually changing a reply as follows: VoidReply to Illegal to VoidReply implies that the save-button gets enabled and therefore the timestamp of the model is changed when you press the save-button. However, the generated code contains a timestamp of a version of a model from which the code has been generated. If the model is not changed syntactically nor semantically but only the timestamp has changed (by an edit-action as described above), then the code will not be overwritten as the generated code is seen as identical (therefore one could argue that the generated code is still generated from a correct version of the model despite the fact that the code has a different timestamp of the model it was generated from).

Tip: use the undo function (control-Z) as much as possible since the enabling and disabling of the save-button is attached to the undo functionality.

An example of a makefile can be found here. It is based on a simple project with two design models and three interface models. The models and the (generated) code are in the same directory with the exception of the runtime code which is stored in a separate sub-directory.

Makefile

```
# ----------------------------------------------------------------------------
# "default", "all", "clean" "verify", and "generate" are targets and will
# always be executed even if a file named accordingly is present
# ----------------------------------------------------------------------------
.PHONY: default all clean verify generate


# ----------------------------------------------------------------------------
# ASD Runtime
# ----------------------------------------------------------------------------
ASD_RT_PATH = ./runtime


# ----------------------------------------------------------------------------
# Tools and their settings
# ----------------------------------------------------------------------------
CC = gcc
CFLAGS = -g -Wall -I/usr/include -I$(ASD_RT_PATH)
LDFLAGS =
AR = ar
ARFLAGS = -cqv


# ----------------------------------------------------------------------------
# ASD Tools and their settings
# ----------------------------------------------------------------------------
ASDPATH = "/cygdrive/c/Program Files (x86)/Verum/ASD Suite Release 4/V9.2.7/"
ASDGENERATE = $(ASDPATH)AsdGenerate
ASDVERIFY = $(ASDPATH)AsdVerify
ASDVERIFYFLAGS = --no-mvr --stop-on-failure
ASDLANGVERSION = -ltinyc -v 9.2.3


# ----------------------------------------------------------------------------
# Generated headers and sources
# ----------------------------------------------------------------------------
GEN_OBJS = $(patsubst %.c, %.o, $(wildcard *Component.c))
GEN_SRCS = $(wildcard *Component.c)
GEN_HEADERS = $(wildcard *Component.h) $(wildcard *ComponentImpl.h) $(wildcard *Interface.h)
GEN_FILES = $(GEN_SRCS) $(GEN_HEADERS)


# ----------------------------------------------------------------------------
# ASD interface models and design models
# ----------------------------------------------------------------------------
ASD_DMS = $(wildcard *.dm)
ASD_IMS = $(wildcard *.im)
ASD_MODELS = $(ASD_IMS) $(ASD_DMS)


# ----------------------------------------------------------------------------
# Main target to be built (including handwritten code)
# ----------------------------------------------------------------------------
MAIN_OBJS = main.o callingclient.o
MAIN_SRC = main.c callingclient.c
MAIN_HEADERS = mytypes.h callingclient.h
MAIN_TARGET = main


# ----------------------------------------------------------------------------
# Dependencies
# ----------------------------------------------------------------------------

# ----------------------------------------------------------------------------
# Default target to built. When called without arguments, make always builds
# first target it finds. In this case, it starts building main
#
# When command 'make all' is given, it also does verification, code generation
# followed by compiling and linking the main target
# ----------------------------------------------------------------------------
default: $(MAIN_TARGET)
all: verify generate default


# ----------------------------------------------------------------------------
# Dependencies of ASD Runtime
# ----------------------------------------------------------------------------
ASD_RT_PATH = ./runtime
ASD_RT_TARGET = $(ASD_RT_PATH)/libasd_rt.a
include $(ASD_RT_PATH)/asd_runtime.mk

# ----------------------------------------------------------------------------
# Dependencies of generated code
# ----------------------------------------------------------------------------
$(GEN_OBJS): $(GEN_SRCS) $(GEN_HEADERS)


# ----------------------------------------------------------------------------
# Dependencies of main
# ----------------------------------------------------------------------------
$(MAIN_OBJS): $(MAIN_SRC) $(MAIN_HEADERS)
$(MAIN_TARGET): $(MAIN_OBJS) $(GEN_OBJS) $(ASD_RT_TARGET)
  $(CC) $(LDFLAGS) $(MAIN_OBJS) $(GEN_OBJS) $(ASD_RT_TARGET) -o $@


# ----------------------------------------------------------------------------
# Dependency describing how to generate object file from source
# ----------------------------------------------------------------------------
%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@


# ----------------------------------------------------------------------------
# Command to verify ASD interface and design models
# ----------------------------------------------------------------------------
verify:
  $(ASDVERIFY) --verify $(ASDLANGVERSION) $(ASDVERIFYFLAGS) --recurse --name \*.im .
  $(ASDVERIFY) --verify $(ASDLANGVERSION) $(ASDVERIFYFLAGS) --recurse --name \*.dm .


# ----------------------------------------------------------------------------
# Command to generate code from ASD interface and design models.
# Note that the generator ONLY overwrites the code IF the code differs from
# the one stored currently in the filesystem.
# ----------------------------------------------------------------------------
generate:
```

```
  $(ASDGENERATE) -g $(ASDLANGVERSION) --recurse --name \*.im .
  $(ASDGENERATE) -g $(ASDLANGVERSION) --all --recurse --name \*.dm .


# ---------------------------------------------------------------------------
# Command to conditionally generate code from ASD interface and design models
# It first checks if code has been generated (if not, it is thrown away),
# followed by checking the verification status of the model. If the model is
# correct, code is generated and otherwise the make process is stopped
#
# Note that the section below is based on shell statements which is simply to
# make it more readable and understandable what happens. Make supports similar
# statements
# ---------------------------------------------------------------------------
generate_if:
  for GEN_CODE in $(GEN_FILES); do \
   $(ASDVERIFY) --query-code $$GEN_CODE; \
   STATUS=$$?; \
   if [ $$STATUS = 0 ]; \
   then \
    echo File $$GEN_CODE : OK; \
   else \
    if [ $$STATUS = 2 ]; \
    then \
     echo File $$GEN_CODE : Out of date and is removed; \
     rm -rf $$GEN_CODE; \
    else \
     echo File $$GEN_CODE : Error and stop; \
     exit 1; \
    fi; \
   fi; \
  done; \
  for MODEL in $(ASD_MODELS); do \
   $(ASDVERIFY) --query-model $$MODEL; \
   STATUS=$$?; \
   if [ $$STATUS = 0 ]; \
   then \
    echo Model $$MODEL : OK. Regenerating code; \
    $(ASDGENERATE) -g $(ASDLANGVERSION) $$MODEL; \
   else \
    if [ $$STATUS = 2 ]; \
    then \
     echo Model $$MODEL : Verification status failed. Reverifying model; \
     $(ASDVERIFY) --verify $(ASDLANGVERSION) $(ASDVERIFYFLAGS) $$MODEL; \
     $(ASDGENERATE) -g $(ASDLANGVERSION) $$MODEL; \
    else \
     echo Model $$MODEL : Error and stop; \
     exit 1; \
    fi; \
   fi; \
  done


# ---------------------------------------------------------------------------
# Command to clean
# ---------------------------------------------------------------------------
clean:
  rm -f *.o $(MAIN_TARGET)
  rm -f $(ASD_RT_PATH)/*.o $(ASD_RT_TARGET)


# This does NOT work. If you touch the .dm without actually changing it,
# then source is generated but since code is identical, it will not be
# overwritten and the make does not continue compile the source, linking it, etc.
#serverComponent.c: server.dm
#   $(ASDGENERATE) -g $(ASDLANGVERSION) $<
```

Asd_runtime.mk as included in Makefile above

```
# ---------------------------------------------------------------------------
# ASD Runtime
# ---------------------------------------------------------------------------
ASD_RT_OBJS = $(patsubst %.c, %.o, $(wildcard $(ASD_RT_PATH)/asd*.c))
ASD_RT_SRC = $(wildcard $(ASD_RT_PATH)/asd*.c)
ASD_RT_HEADERS = $(wildcard $(ASD_RT_PATH)/asd*.h)


# ---------------------------------------------------------------------------
# Dependencies in ASD runtime
# ---------------------------------------------------------------------------
$(ASD_RT_OBJS): $(ASD_RT_SRC) $(ASD_RT_HEADERS)
$(ASD_RT_TARGET): $(ASD_RT_OBJS)
  $(AR) $(ARFLAGS) $@ $(ASD_RT_OBJS)
```