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

ASD:Suite Visual Verification Guide

ASD:Suite Release 4 v9.2.7

TABLE OF CONTENTS

ASD:Suite Visual Verification

User Guide

ASD:Suite Release 4 v9.2.7

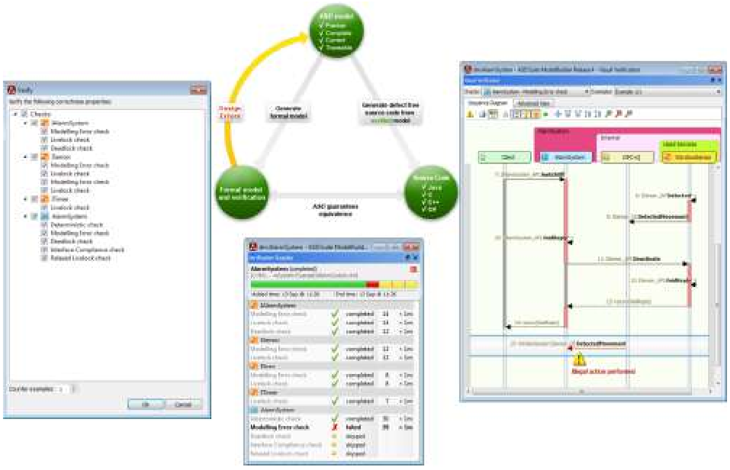
ASD:Suite verification is model verification using the ASD:Suite. Using the ASD:Suite you prove the correctness and completeness of your design or interface models. Readers of this guide should be familiar with Sequence Based Specifications (SBS) and should know how to create ASD models using the ASD:Suite ModelBuilder. This information can be found in the "ASD:Suite User Manual".

This guide is a reference for ASD:Suite users on how to verify ASD models using the ASD:Suite, how to interpret the results, and how to fix any problems found.

The main verification concepts are outlined in "ASD:Suite verification concepts".

The performed checks are introduced in "What is verified?".

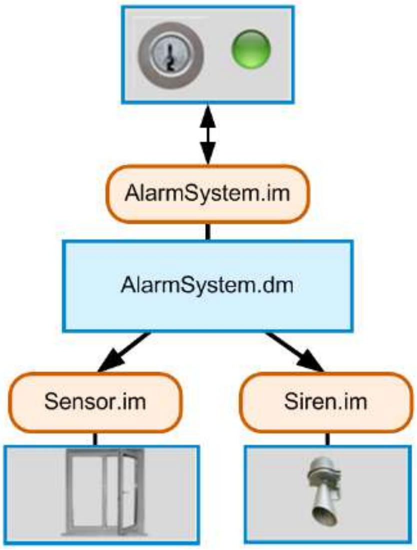
In the "Guidelines for verification" section you find guidelines for using the Visual Verification feature of the ASD:Suite ModelBuilder, while in "Guidelines for reducing verification time" you find guidelines to reduce the verification time.



ASD:Suite verification concepts

The ASD:Suite is used to detect errors in interface models and design models. Removing these errors ensures that the generated code works and that the resulting system has certain desirable properties, such as deadlock-freedom. See "What is verified?" for an explanation of the properties that are checked .

The following figure shows the AlarmSystem component, which implements the AlarmSystem service and uses the Sensor and Siren services. The blue rectangle represents an instance of the AlarmSystem component specified by the ASD Design Model called AlarmSystem.dm. The grey rectangles represent one or more instances of components implementing the Siren and the Sensor attached to the system. The orange oval shapes represent services each of which is specified by an ASD Interface Model.

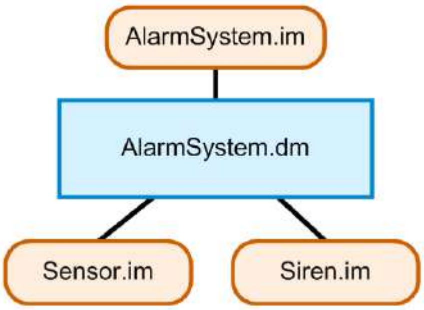


An Alarm system

Two types of verification can be performed:

- Verification of interface models;
- Verification of design models, including the referenced interface models.

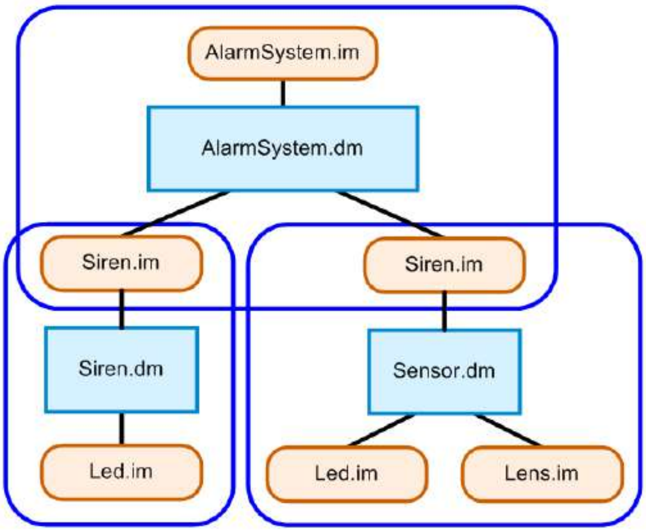
The following figure shows the verification scope for the AlarmSystem design:



Verification scope for a design model

Verifying a design model on its own together with its referenced interface models ensures that the behaviour specification is complete and the respective component behaves correctly in its environment according to the specified behaviour. This approach, called *compositional verification,* makes the ASD:Suite scalable with respect to large software systems; verifying each component individually ensures correctness of the entire system.

The following figure shows the compositional verification with emphasis on the verification scope for each component:



Compositional verification

When all models in a system are verified, the resulting complete system is guaranteed to have the correctness properties which are described in "What is verified?".

Assumptions

The following list contains the assumptions on which verification using the ASD:Suite is based:

- *Foreign components specified as used components behave according to their service specifications*. When verifying a component using the ASD:Suite only knowledge about the used services is required.
- *Foreign components which are clients of the verified component observe the "non-blocking notification" rule*. The foreign component is a client of the verified component and is responsible for correct handling of the received notifications such that they do not cause blocking at runtime.

Limitations

ASD:Suite verification focuses on verifying properties that are hard for humans to verify. These properties mainly concern ordering of events, asynchronous behaviour, deadlock and livelock. See "What is verified?" for a brief description of these properties. The list below gives an indication on what is ***not*** checked:

- *Anything not specified in a model:* Only what is specified in the model is checked. This means that system wide deadlocks caused by resource allocation contention or similar causes are not detected by verification using the ASD:Suite.
- *Dependency on data:* Verification using the ASD:Suite is data independent. This is caused by the fact that only sequence of events are checked and the data carried by the parameters specified for events defined in the ASD model is not considered. Data correctness is usually very domain-specific and is very time-consuming to check. Given the all-purpose nature of the ASD:Suite, data correctness checks are therefore not done.

  The only exception are the data variables, that are used to pass on data within an ASD component, across rule-cases. They are checked for having a proper value at the moment when they are read.
- *Timing properties*: In so-called "hard" real-time systems, timing properties are also correctness properties; a real-time system that fails to meet its deadlines is not correct. The ASD:Suite cannot be used to verify the correctness of timing properties in real-time systems. Arrival rates and service times can be approximated in interface models by notification event yoking, leaving to you the responsibility of reflecting the real runtime behaviour. See "Yoking Notification Events" for details about yoking.

What is verified?

Components developed with the ASD:Suite ModelBuilder can be verified to ensure that:

- Interface and design models are *complete* and *well formed*.

  This ensures that every possible scenario is considered and covered in the models and that all models are free from errors such as range errors, livelocks and deadlocks, that design models are deterministic and that data variables are used properly in the design model.

- The design complies with the service specifications of components it uses.

  This means that the design never sends any trigger to a used component unless that used component is in such a state that the trigger is legal and that the design is willing to accept every notification and reply event received from the used component whenever, according to its service specification, the used component is allowed to send it.

- The design together with all of its used components complies with the implemented service specification.

  This means that all behaviour required by the service specification is implemented by the design and all behaviour implemented by the design is allowed by the service specification.

Deadlock-freedom

A *deadlock* is a situation where none of the components in a system can make progress; nothing can happen and the system simply does not respond. This commonly occurs when two components each require an action from the other before they can perform any further action themselves. Another common cause is when a component is waiting for some external event which fails to occur.

The following Sequence Based Specifications (SBS) show a simple example of a deadlock: on rule case 13 in the design model SBS, the Alarm system sends a 'Deactivate' event to the Sensor, after which it waits in the 'Deactivating' state for the Sensor to send the 'Deactivated' notification. However in the Sensor interface model, the Sensor never sends the 'Deactivated' notification in reaction to the 'Deactivate' event in rule case 10 in the interface model SBS. Now both Alarmsystem design and Sensor interface are waiting for each other and cannot proceed, hence the deadlock situation.

Alarm system design model:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
9	**Activated_Idle**					
13	IAlarmSystem_API	SwitchOff		IAlarmSystem_API.VoidReply; WindowSensor:ISensor_API.Deactivate		Deactivating
14	WindowSensor:ISensor_CB	DetectedMovement		IAlarmSystem_CB.Tripped; Timer:ITimer:CreateTimer(5)		Activated_Tripped
17	**Deactivating**					
22	WindowSensor:ISensor_CB	DetectedMovement		NoOp		Deactivating
23	WindowSensor:ISensor_CB	Deactivated		IAlarmSystem_CB.SwitchedOff		NotActivated

Sensor interface model:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
7	**Activated**					
10	ISensor_API	Deactivate		ISensor_API.VoidReply		Deactivated
11	ISensor_INT	Detected		ISensor_CB.DetectedMovement		Triggered

SBS showing a deadlock in a system

In general, deadlocks can be hard to find, because the entire system needs to be reviewed to discover them and freedom from deadlocks is a property of the system as a whole. For example, component A might be waiting for B which is waiting for C while C is waiting for A. ASD ensures that this never happens in the following way:

- Each component by itself can be verified as being deadlock free;
- Components built using the ModelBuilder can only be composed in ways that have been proven not to cause deadlock.

Note: The ASD:Suite can only verify what it knows; therefore, e.g. handwritten code or resource allocation schemes can still cause deadlocks.

Livelock-freedom

A component is said to be *livelocked* when it is permanently busy with internal behaviour but ceases to serve its clients. For example, due to a design error such that the design is constantly interacting with its used components and starving the client; or due to the arrival rate of unconstrained external events such that processing them starves the client. As seen from the outside of a component, this appears very similar to deadlock. The difference is that a deadlocked component does nothing at all whereas a livelocked component might be performing lots of actions, but none of them are visible to the component's clients.

The most common cause of a livelock in an ASD Interface Model is the incorrect use of modelling events. The simplest case of a livelock is the specification of NoOp as action to a modelling event together with a transition to a state where the only possible operation is a transition to the originating state on a rule case having a modelling event as trigger and NoOp as action. It considers the modelled event to occur so often that the outside world no longer gets the chance to interact with the system. The following figure shows an example of such a livelock.

	Interface	Event	Guard	**Actions**	State Variable Updates	Target State
19	**Triggered**					
22	ISensor_API	Deactivate		ISensor_API.VoidReply		Deactivating
23	ISensor_INT	Detected		NoOp		Triggered

Livelock in interface model

The ASD:ModelBuilder will detect this and will raise a conflict for this type of livelock.

The most common cause of a livelock in an ASD Design Model is a looping interaction with a used component. In the example shown in the following figure the design triggers an action in a used component that results in a synchronous reply event being sent back to the design, which in turn triggers the same action in the used component again.

	Interface	Event	Guard	Actions	**State Variable Updates**	Target State
1	**NotActivated (initial state)**					
4	IAlarmSystem	SwitchOn+		Sensor:ISensor.Activate+		Activating
11	**Activating (synchronous return state)**					
16	Sensor:ISensor	OK		IAlarmSystem.Ok		Activated_Idle
17	Sensor:ISensor	Failed		Sensor:ISensor.Activate+		Activating

Livelock in design model - SBS of the design model

The following figure shows the SBS of the interface model of the used service. The rule in the design that processes the reply event (rule case number 17 in the previous figure) triggers the same action again in the used component and remains in the current state. The result is an endless cycle of processing reply events while the Client remains starved.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Deactivated (initial state)**					
3	ISensor	Activate+		ISensor.OK		Activated
4	ISensor	Activate+		ISensor.Failed		Deactivated

Livelock in design model - SBS of the used service interface model

verum®

Home Product Technology Resources Training Purchase Company

Guard completeness

A guard on a rule case is a precondition in the form of a Boolean expression. When a guard evaluates to "true", the corresponding rule case is enabled; when it evaluates to "false", the corresponding rule case is disabled. A rule case without a guard is semantically equivalent to one with a constant guard expression "true".

It is a required that every rule in an ASD Interface Model has at least one enabled rule case unless the trigger is a modelling event, in which case it is allowed to have zero enabled rule cases. It is required that every rule in an ASD Design Model has ***exactly one*** enabled rule case. Therefore, except in the case of modelling event rules in interface models, the guards on all rule cases of a given rule in a given state must account for all possible values of the associated state variables that can actually occur in that state. Otherwise, all rule cases would be disabled and this is not allowed.

The following figure shows an ASD model with guards. The model represents a shower; when the temperature is more than 20 then the user can make it colder, and when the temperature is below 30 then it can be made hotter.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**InitialState (initial state)**					
3	IShower	MakeColder	Temperature > 20	IShower.VoidReply	Temperature=Temperature-5	InitialState
4	IShower	MakeColder	Temperature <= 20	Illegal		-
5	IShower	MakeHotter	Temperature < 30	IShower.VoidReply	Temperature=Temperature+5	InitialState
6	IShower	MakeHotter	Temperature > 35	Illegal		-

Guard completeness failure

There is no problem for the MakeColder event: all cases are specified. However, the MakeHotter event is not handled in all cases. For example, when the temperature is between 30 and 35, no behaviour is specified. This is no problem if the system is started with a temperature that is below 30 and is a multiple of 5; then it could never rise above 30 again. However, if the initial temperature is above 30 degrees, then the problem occurs. Depending on the initial temperature, the error will be flagged during verification.

Absence of illegal behaviour

In software systems, one has to adhere to the specification of used libraries. Similarly, an ASD Design should make proper use of its used components and adhere to their service specifications. Consider component AlarmSystem which uses a Siren service. According to the Siren service specification it is allowed to alternately turn on and off a siren. If the AlarmSystem now tries to turn on the siren while it is already on, it is performing an illegal behaviour.

Absence of range violation errors for state variables and used service references

When you define a state variable, Integer or Used Service Reference type, you must specify a range for the expected values. This is similar to having limited-size variables in a software program. Additionally, when you define a Used Service Reference you must specify a cardinality for it, i.e. you define a validity range for the reference. During verification using the ASD:Suite, it is checked that these ranges are not exceeded.

Home Product Technology Resources Training Purchase Company

Absence of state invariant violations

A state invariant is a precondition that must always evaluate to true whenever the component enters in the corresponding state; if omitted, this is equivalent to a state invariant with the constant value "true". In an ASD model, you can specify a state invariant per state. During verification every transition into a state, including self transitions, causes the state invariant to be evaluated. If there are any circumstances where it evaluates to false this is detected and signalled as an error.

Absence of reply event errors

Each occurrence of a call event should have exactly one matching reply event (being VoidReply or a valued reply event). This is verified if you verify your model using the ASD:Suite.

Determinism

All ASD Design Models are required to be deterministic. The most common cause of non-determinism in a design model is overlapping guards. That is, guards are specified within a single rule such that more than one of them can evaluate to true at the same time. In this case, there is no defined deterministic choice between the competing rule cases.

Note that interface models are allowed to be nondeterministic. This is because an interface model is an abstraction of all possible compliant designs that could implement it. An interface model specifies all possible outcomes without revealing any details as to how the given implementation actually resolves its choice between outcomes at runtime.

Absence of queue overflow errors

An ASD Design Model which has used components with notification interfaces also has a queue where notification events are stored before they are processed. During verification it is checked that that this queue does not overflow, i.e. that it remains non-blocking. You can set the queue size for each design model. See "Preparing an ASD model for verification" for details on how to set the queue size.

Home Product Technology Resources Training Purchase Company

Compliance of the implementation with the specification

In general, software systems are first *specified* and then *implemented*. The implementation must comply with the specification. There is usually more than one way to implement a specification. ASD has a similar concept: a component (design model) *implements* a service (interface model). The verification using the ASD:Suite assures interface compliance.

Terms of use | Privacy Policy

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

Data Variables

Data variables are used to remember argument values across rule cases (for a more detailed description of the use of data variables see the Parameter Usage and the Data Variables sections of the User Manual). The Model Verification checks the correct use of the Data Values, i.e. whether a data variable is valid before it is used.

A data variable is read when its value is passed on to another component (client or server) via:

- an [in] or [inout] parameter on an action event
- an [out] or [inout] parameter on a trigger event

A data variable is written when it receives a value from another component or when it is explicitly initialised or invalidated via:

- an [in] or [inout] parameter on a trigger event
- an [out] or [inout] parameter on an action event
- an explicit *Initialise* event that makes the data variable valid (i.e. it gets a useful value)
- an explicit *Invalidate* event that makes the data variable invalid (e.g. to prevent using old data that is no longer relevant)
- an Auto-Initialise property that ensures the data variable is given an initial value upon construction of the variable
- an Auto-Invalidate property that ensures that a data variable is invalidated at the end of each action sequence, which is that the state transition to a normal state (e.g. to prevent using old data that is no longer relevant)

Note: the explicit Initialise / Invalidate and the Auto-Initialise and Auto-Invalidate events are currently for verification purposes only. They have no effect in the generated code at runtime. The purpose of the data variable verification is to check that data variables are never read when they do not have a proper value. The Initalise and Invalidate events are merely an aid for the verification.

The model verification engine will make sure that all data variables have a proper value at the moment that they are read. This can prevent a number of problems, amongst which:

- runtime errors caused by uninitialised data variables that otherwise would take a lot of effort debugging.
- data 'leaking' from one action sequence to another, where a data variable unintendedly has retained a value from a previous action sequence.

In addition it verifies the data invariants, with which one can make sure that in a particular state a data variable is always valid or invalid.

For information on how to use data invariants, see the Invariants section in the user manual.

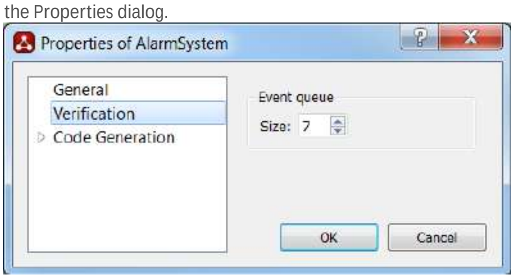
Guidelines for verification

Preparing an ASD model for verification

Even though by default every ASD model is prepared for verification, there are cases when the default configuration settings are not sufficient. For example, the default size of the queue can be too small.

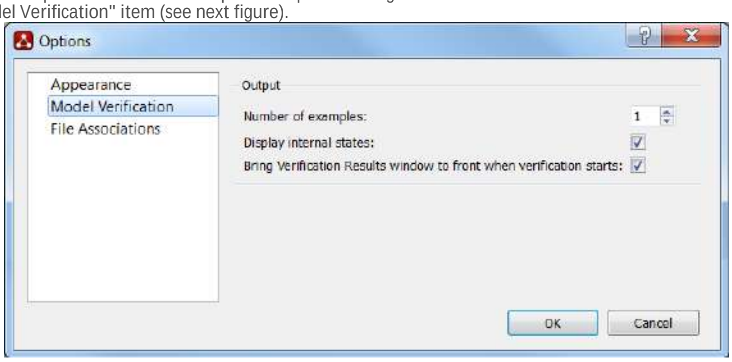
These are the steps for changing the configuration settings to verify a design model:

1. Load the design model in the ASD:Suite ModelBuilder.
2. Specify a desired size for the notification event queue. This represents the number of notification events which can be stored in the queue that is located between the component and the used services.
    - Select the "Properties" menu item in the context menu obtained when pressing the right mouse button having the design model selected.
    - Click the "Verification" item in the Properties dialog.



The Verification Properties dialog

Note:
    - The maximum size for the queue can be 99.
    - High values will increase verification time.
    - Low values might cause the reporting of queue size violation.
    - Click on the "OK" or "Cancel" button to close the "Properties" dialog.
3. Specify the number of counter examples. For each failed check, you get at least one example of a sequence of events that leads to the error situation. The default number of counter-examples per check is 1. You can set a different value, to a maximum of 25, by selecting the "Model Verification" item in the Options dialog window.
    - Select the "Tools -> Options" menu item to open the Options dialog.
    - Select the "Model Verification" item (see next figure).



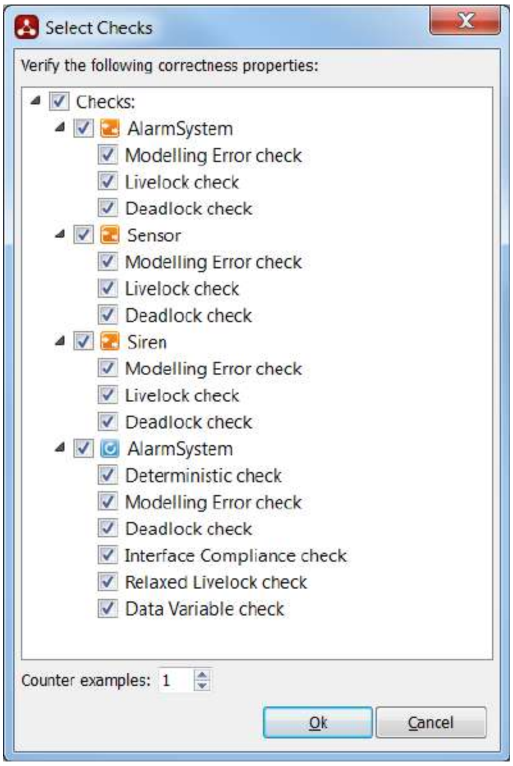
The "Model Verification" tab in the Options dialog

    - Specify the desired number.
    - **Note:** There might not be as many counter-examples as specified, in which case you get less.
4. If you would like to receive an indication about the complexity of your model, check-mark the check-box next to the "Display internal states" text shown in the previous figure. The status of the respective check-box, i.e. checked or not, enables, or respectively disables the showing of verified internal states in the "Verification Results" window. The number of verified internal states is a good approximation of the number of verified executions scenarios.

See "Start verification" for details about starting verification.

Start verification

The easiest way to start verification is to press F5 after you load an ASD model in the ASD:Suite. This will send the opened model(s) to the ASD Server where they are analyzed to see which checks can be performed. When the analysis is done the list of checks is shown in the "Select Checks" dialog.



The Select Checks dialog for design models

Note: See "The Select Checks dialog - select and run checks" for details about the Select Checks dialog and the next steps in verification. These are the alternatives to open the "Select Checks" dialog, i.e. to start verification:

- Select the "Verification -> Verify..." menu item, or
- Press the "Verify" button on the application toolbar, or
- Select the "Verify..." item in the context menu obtained when clicking with the right mouse button on the ASD model in the "Model Explorer" window

Clicking the OK button in the "Select Checks" dialog starts the verification. See "The verification user interface" for details.

Note:

- A model must pass the conflicts check ("Tools -> Check Conflicts") before it can be verified.
- If the code generator version is not specified in the model properties of the to-be-verified model(s) the "Verify" dialog is shown:

The "Verify" dialog

Note:

- In order to ensure that the verified runtime semantics are exactly the same as those of the generated code, the ASD:Suite ModelBuilder will ask you to supply both a target source code language and version. This will make sure that both the verification and the subsequent code generation will be performed with exactly the same ASD semantics. For an example see the next dialog:

Selected target language and code generator version

Note: In case you check-mark the "Save Settings" checkbox the specified settings are saved in the ASD model and will be considered as the default settings for future verifications and code generations using the respective ASD model.

- Verification will start when you click the OK button

Additionally, you can perform all possible checks without using the Select Checks dialog. These are the alternatives to perform all checks:

- Press Ctrl+F5, or
- Select the "Verification -> Verify All" menu item, or
- Press the "Verify All" button on the application toolbar

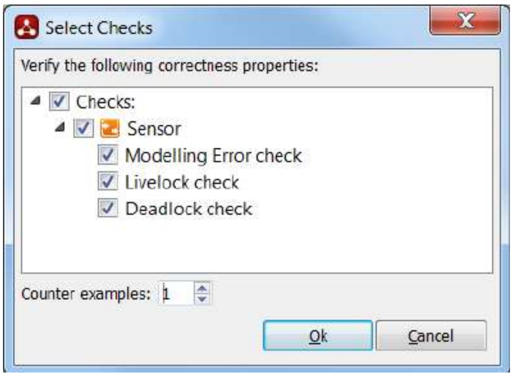
These are the alternatives to re-run the set of checks which you have run for the open model(s) in the current ASD:Suite session:

- Press Ctrl+Shift+F5, or
- Select the "Verification -> Verify Again" menu item, or
- Press the "Verify Again" button on the application toolbar

The verification user interface

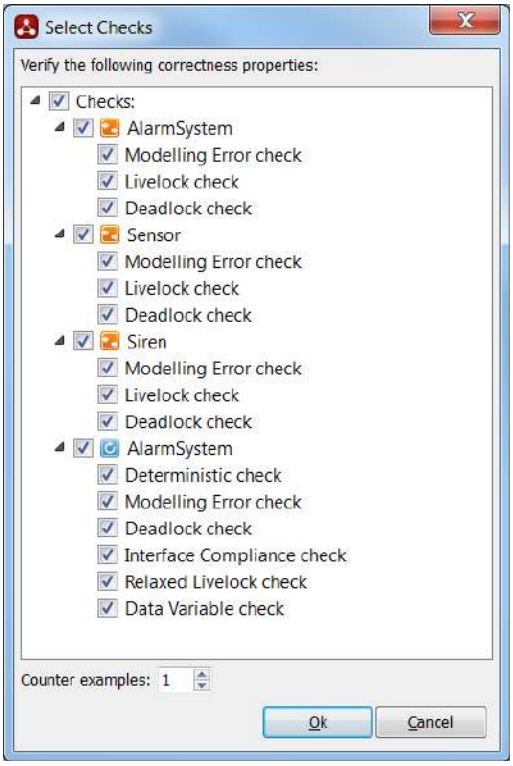
The Select Checks dialog - select and run checks

The following figure shows the Select Checks dialog for verification of interface models,



The Select Checks dialog for interface models

while the next figure shows the Select Checks dialog for design models:



The Select Checks dialog for design models

The following list contains a brief description for the checks shown in **interface model checks**:

- ***Modelling Error check:*** this is a check which ensures verification of the following:
  - Guard completeness
  - Absence of illegal behaviour
  - Absence of range violation errors for state variables
  - Absence of state invariant violations
  - Absence of queue size violation
  - Misplacing of reply events
- ***Livelock check***: This check ensures verification for absence of livelocks.
- ***Deadlock check***: This check ensures verification for absence of deadlocks.

The following list contains a brief description of the checks shown in **design model checks** not shown in **interface model checks**:

- ***Deterministic check***: This check ensures verification for absence of non-determinism in your design model.
- ***Interface Compliance check***: This check ensures verification of implementation adherence to specification.
- ***Relaxed Livelock check***: The livelock check for design models is relaxed in the sense that a number of livelocks that in an actual system will never result in an unresponsive system are not flagged as an error during model verification. Some examples of these are:
  - an unused interface with events that do not result in a state transition in the used interface model
  - spontaneous notification events on a used interface that do not result in a state transition in the design model
- ***Data Variable check***: This check ensures verification of correct use of data variables and data invariants. Note: this check is only visible if data variables are defined and used in the design model.

Note: See "What is verified?" for details about what is verified.

The recommended order of running the checks is from top to bottom. Checks lower in the list may not have meaning or are invalid if a previous check fails. The following list contains a brief description of several dependencies between checks:

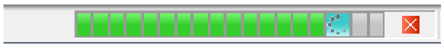
- All checks on interface models and the determinism check on the component design model are independent of each other in the following sense: the failure of one or more of these assertions does not invalidate the successful results of the others.
- All checks for the design model are meaningful only when all the checks on interface models have succeeded.
- The "Modelling Error check" for a design model is only meaningful when all checks on interface models and the "Determinism" check succeed.

Clicking the OK button in the "Select Checks" dialog runs the selected checks. See "The "Verification Results" window and the verification progress bar" for details about verification progress indication and displaying of results.

The "Verification Results" window and the verification progress bar

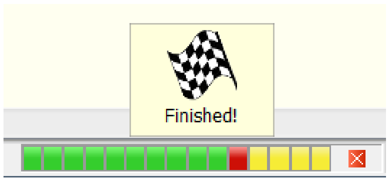
The progress of running a set of checks and the status information is shown in the "Verification Results" window and in the verification progress bar in the status bar of the ASD:Suite ModelBuilder.

The following figure shows the progress of verification,



Verification progress in the status bar

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

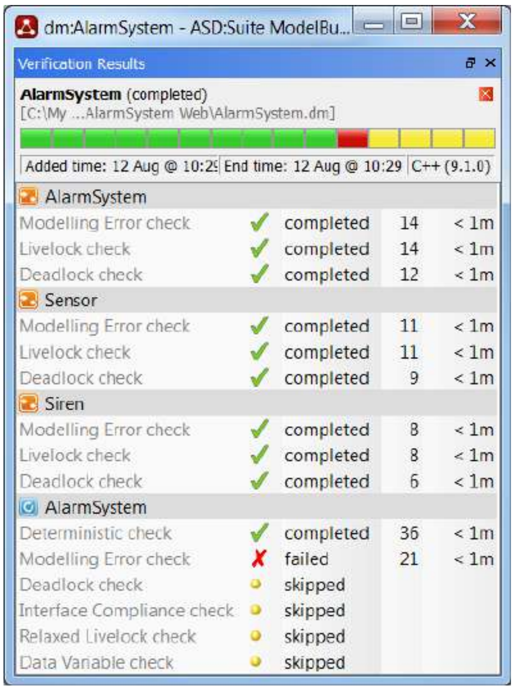


Verification end shown in the status bar

Note: The number of the rectangles shown in the verification progress bar is the same as the number of checks to run. The following list contains an explanation for the items which appear in the verification progress bar:

- The red cross: the button to stop verification
- A green rectangle: a successful check
- A red rectangle: a failed check
- A grey rectangle: a check which did not run yet
- A light blue rectangle with a running circle: a check which currently runs
- A blue rectangle: a failed check due to an internal error
- A yellow rectangle: a skipped check

The following figure shows the "Verification Results" window after running the **design model checks**:



The "Verification Results" window

Double-clicking a failed check in the "Verification Results" window or clicking a red cell in the verification progress bar brings up the failure related information in the "Visual Verification" window.

Note:

- When you see a message in the "Output Window" starting with the following text: "One of the models could not be verified due to an internal error.", check that none of the causes mentioned in "Fixing model compilation errors" occurs, or contact Verum for support.
- Double-clicking a "failed" check while other checks are running, stops the verification and brings up the failure related information in the "Visual Verification" window.

See "The "Visual Verification" window" for details about the information shown in the "Visual Verification" window.

The "Visual Verification" window

The "Visual Verification" window shows verification failure related information. The information is shown in two tabs, the "Sequence Diagram" tab and the "Advanced View" tab. Both tabs show information about the same failure but in a different graphical representation.

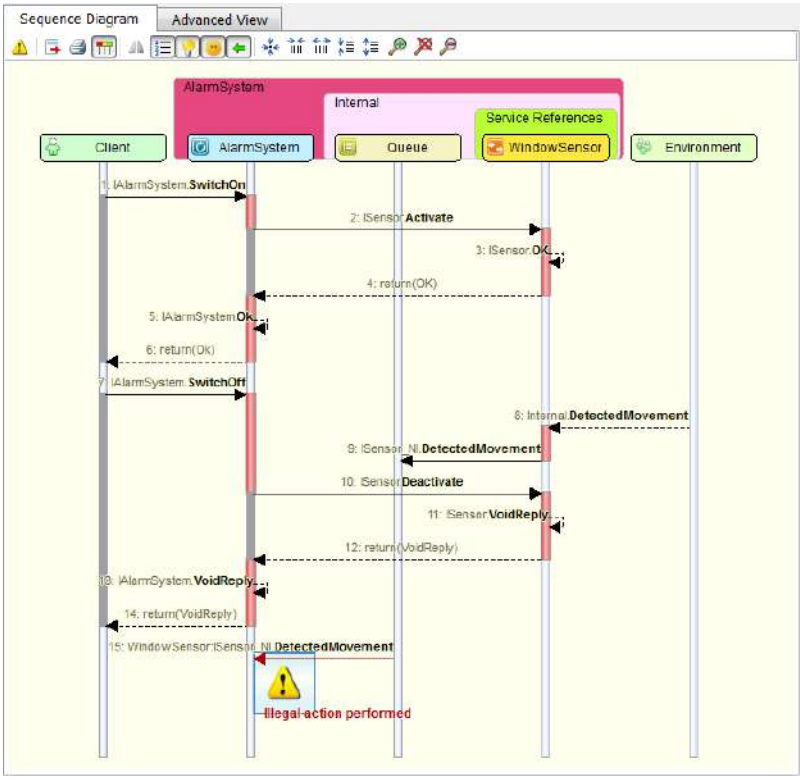
The information provided in the "Visual Verification" window is the main input for fixing the encountered failures using Interactive Visual Verification.

See "The "Sequence Diagram" tab" and "The "Advanced View" tab" for information about each tab individually .

See "Interpreting Verification Results" and "Fixing reported errors" for guidelines on how to identify and fix verification errors.

The "Sequence Diagram" tab

The following figure shows the "Sequence Diagram" tab of the "Visual Verification" window for the failed check shown in **The "Verification Results" window and the verification progress bar**:



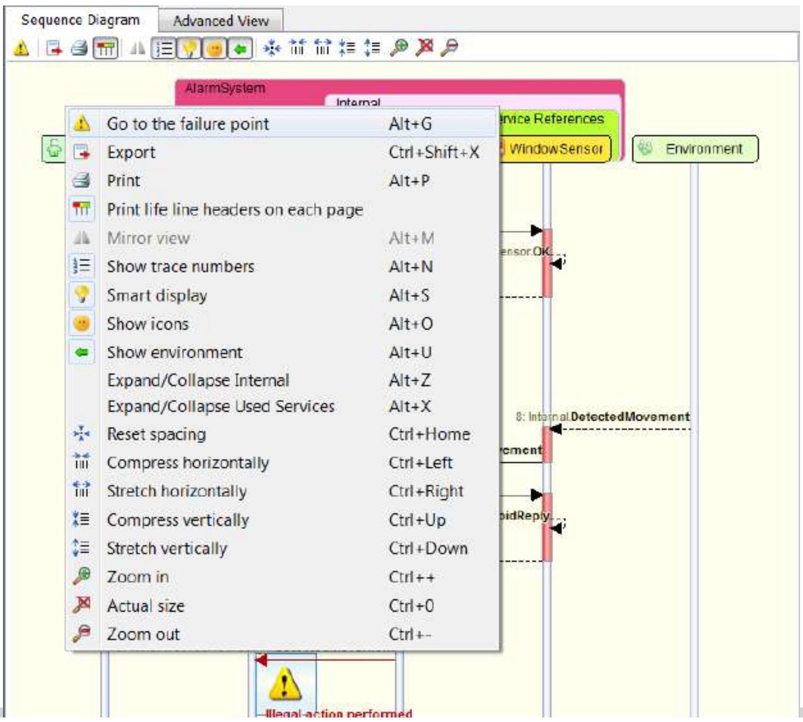
The "Sequence Diagram" tab of the "Visual Verification" window showing a failure trace

Each vertical line in the sequence diagram is called a "lifeline". It represents a component in the system. At the top of each lifeline is a "lifeline header": a coloured box that shows the name of the component.

The sequence diagram shows event occurrences through time. Each event is shown as an arrow between two lifelines. Lifelines are coloured to show thread activity: a red section on a lifeline means that a thread is active in the corresponding component; a dark gray section on a lifeline means that the component is blocked on an operation.

The unblocking of the client thread is marked explicitely (see the events tagged "return" in the picture above). This illustrates that the position of the reply event in the model is not always the moment when the thread of control returns to the client. See the "ASD Runtime Guide" for details about threading and the ASD runtime semantics.

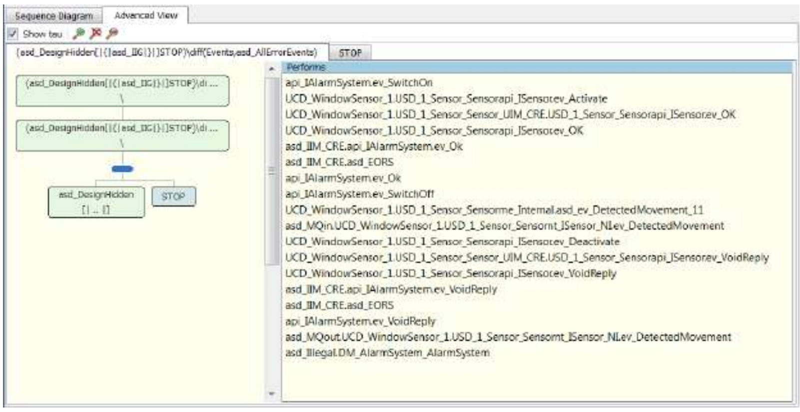
The following figure shows a list of navigation related actions for the "Sequence Diagram" tab, together with the associated keyboard shortcuts, which allow you to easily locate the required information in the sequence diagram.



Navigation related actions in the "Sequence Diagram" tab

The "Advanced View" tab

The following figure shows the "Advanced View" tab of the "Visual Verification" window for the failed check shown in **The "Verification Results" window and the verification progress bar**:



The "Advanced View" tab of the "Visual Verification" window showing a failure trace

This window exposes the mathematics underlying ASD to provide advanced users with a different view on what is wrong with their model.

Interpreting verification results

The following sections provide a guideline to analyse verification errors using Interactive Visual Verification:

Visualizing in the SBS an event selected in the sequence diagram

Replaying a failure trace - forward and backward stepping and jumping

Displaying state and state variable related information

Displaying the content of the event queue

Note:

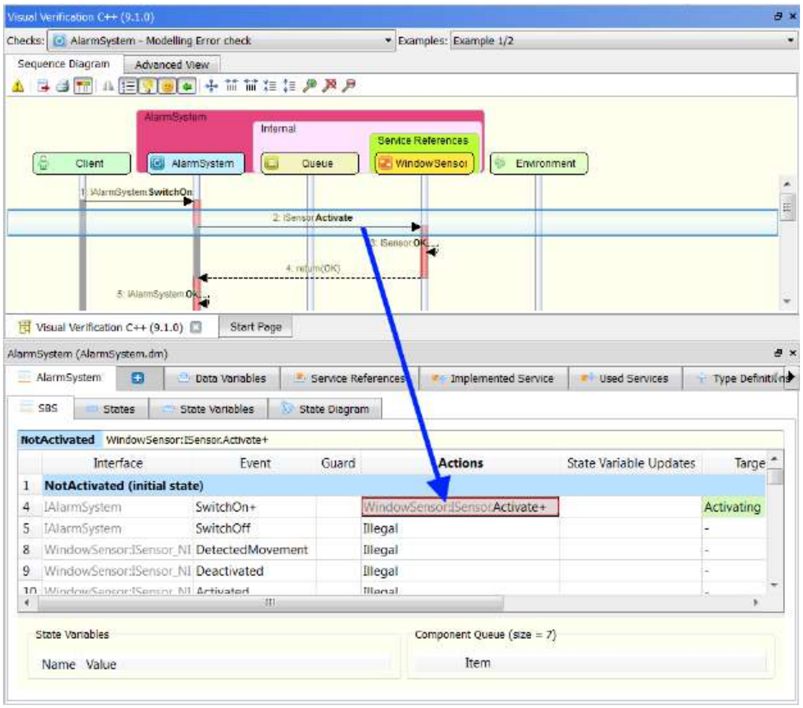
- The checks that fail are indicated by a red cross in the "Verification Results" window or by a red rectangle in the verification progress bar.
- Double-clicking on a "failed" check in the "Verification Results" windows brings up a failure trace in the "Sequence Diagram" tab of the "Visual Verification" window.

Visualizing in the SBS an event selected in the sequence diagram

The following figure shows how you can see where a selected event in the sequence diagram appears in the SBS; in which state, in which rule case, etc.

Single-clicking the event merely shows the current position in all SBS tabs. Double-clicking the event also brings the most appropriate SBS tab to the front. Note that you can also single-click or double-click the warning icon to select the most likely location of the error in your model.

Alternatively, use the Up and Down arrow keys to navigate through the sequence diagram. Use the Enter key to bring the most appropriate SBS tab to the front.



Location of a specified event in the SBS

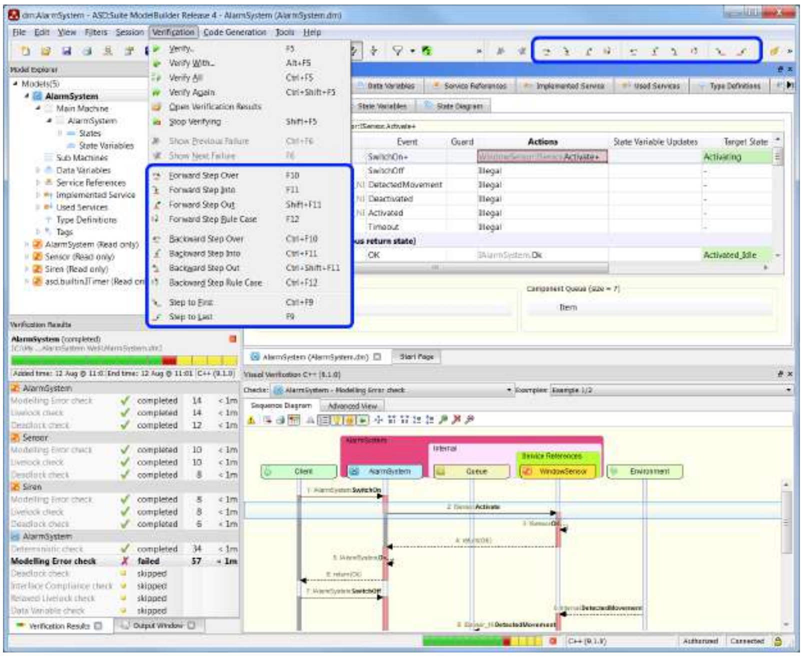
For Interface Compliance Errors, two sequence diagrams are shown. The current trace position in the two sequence diagrams is always synchronized. The left sequence diagram position is highlighted in orange in the SBS tabs, while the right sequence diagram position is highlighted in red.

Replaying a failure trace - forward and backward stepping and jumping

The main advantage of the Interactive Visual Verification is the possibility to "replay" the failure trace and see the sequence of events and operations in the SBS. This provides enough information about why the sequence of events shown in the sequence diagram leads to the reported error.

The replay of the sequence of events shown in the sequence diagram is supported by a set of forward, respectively backward, stepping and jumping operations. These operations are available in each SBS tab. Most operations are NOT available in the sequence diagram, because they are dependent on which SBS is selected.

The following figure shows the alternatives to initiate forward or backward stepping or jumping in the provided trace:



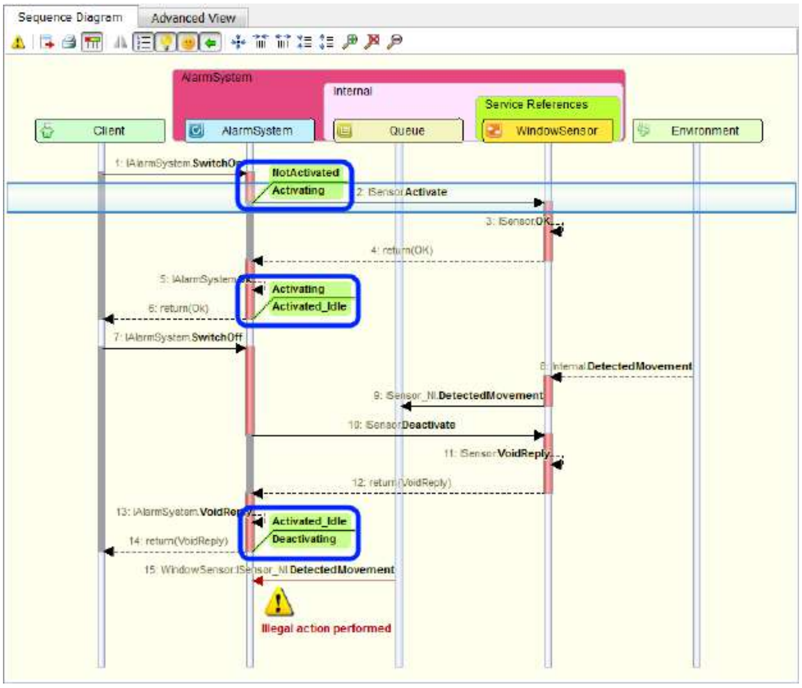
Stepping actions

verum®

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

Displaying state, state variable and data variable related information

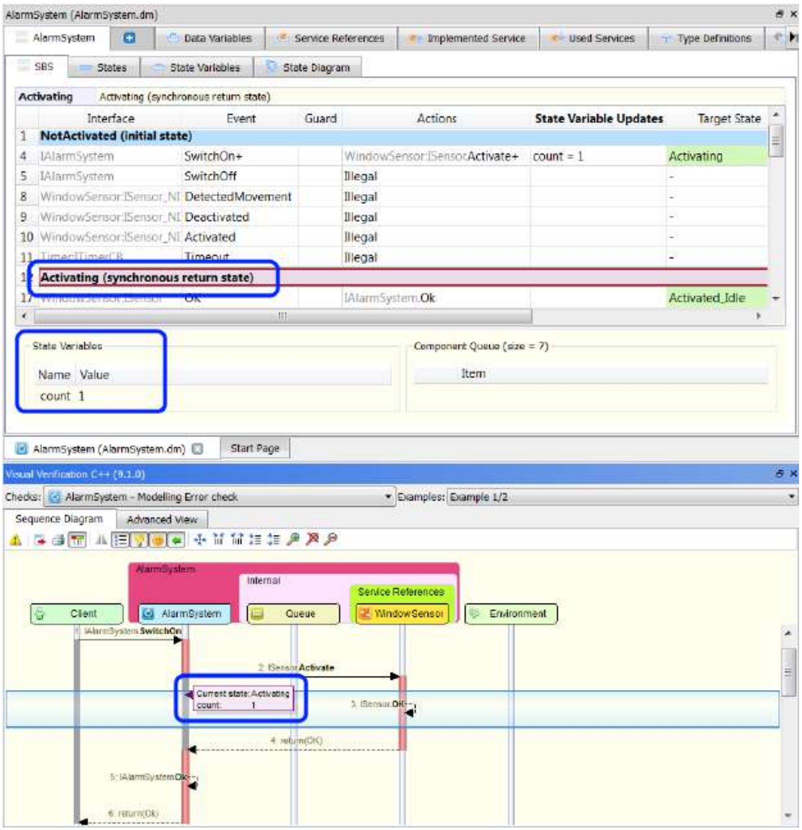
You can hover with the mouse over a lifeline header that corresponds to a component or a used service in the sequence diagram. This provokes green pop-ups that show the state changes on the lifeline (see next figure).



Highlighting the state changes for an ASD component

Hovering over the lifeline itself (as opposed to the header) displays the state in which the component is at the indicated point in time, plus the current value of the state variables (see next figure).

Tip: Press the Ctrl key while hovering over a lifeline to "lock" the mouse onto the lifeline, making it easy to move up and down the lifeline.



Display current state and current value of state variables

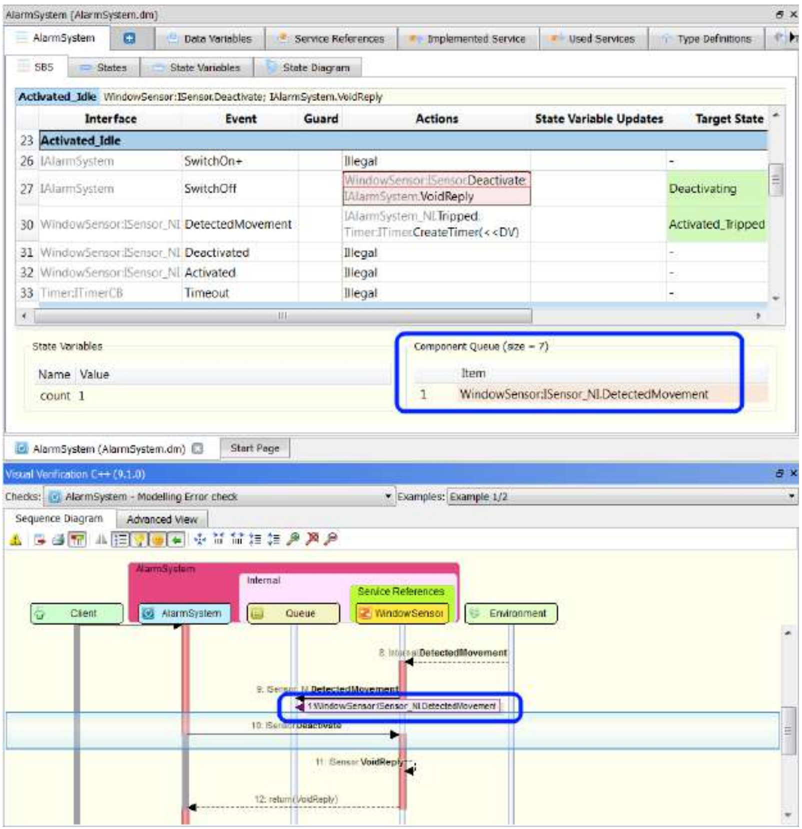
Note: The values of the state variables and data variables are also shown in the watch window located at the bottom of the SBS tab. This value always corresponds to the current position in the trace. When a value changes (because the current position in the trace changes), the background of the value is highlighted in red.

verum

Displaying the content of the event queue

Hovering over the lifeline of the event Queue, the content of the event queue is shown in a box (see next figure). This is the content of the queue at the time corresponding to the position of your mouse on the lifeline. Move the mouse up and down the lifeline to see the queue contents change.

Tip: Press the Ctrl key while hovering over a lifeline to "lock" the mouse onto the lifeline, making it easy to move up and down the lifeline.



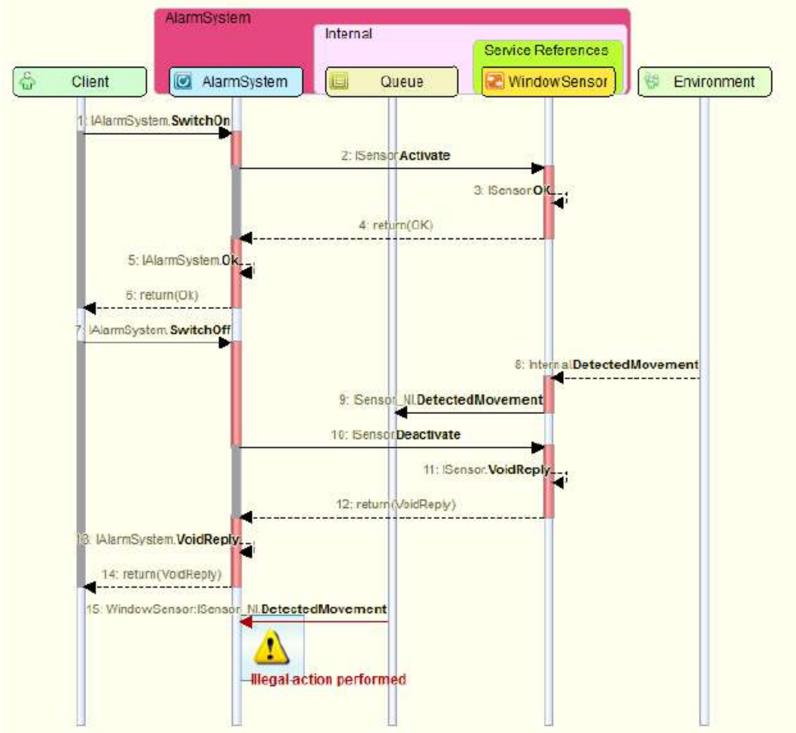
Displaying the content of the event queue

Note: The same content is shown in the watch window located at the bottom of the SBS tab. When the queue contents change, the changed notifications are highlighted in red. Notifications that went out of the queue are shown striked-through before they are removed from view.

Fixing reported errors - Fixing modelling errors

Fixing "Illegal action performed"

This section provides a guideline to fix a modelling error caused by an illegal action. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing a trace with an illegal action

The following figure shows the rule case for the error reported in the previous figure:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
34	**Deactivating**					
37	IAlarmSystem	SwitchOn+		Illegal		-
38	IAlarmSystem	SwitchOff		Illegal		-
41	WindowSensor:ISensor_NI	DetectedMovement		Illegal		-
42	WindowSensor:ISensor_NI	Deactivated		IAlarmSystem_NI.SwitchedOff		NotActivated
43	WindowSensor:ISensor_NI	Activated		Illegal		-
44	Timer:ITimerCB	Timeout		Illegal		-
45	**Activated_Tripped**					
48	IAlarmSystem	SwitchOn+		Illegal		-
49	IAlarmSystem	SwitchOff		Timer:ITimer.CancelTimer; WindowSensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating
52	WindowSensor:ISensor_NI	DetectedMovement		Illegal		-
53	WindowSensor:ISensor_NI	Deactivated		Illegal		-
54	WindowSensor:ISensor_NI	Activated		Illegal		-
55	Timer:ITimerCB	Timeout		Siren:ISiren.TurnOn		Activated_AlarmMode

Rule case with the illegal action in the SBS

In this example, on line 41 in the SBS for the Alarm system it is specified that the "ISensorCB.DetectedMovement" is an illegal behaviour. The presented trace shows the exact sequence of events.

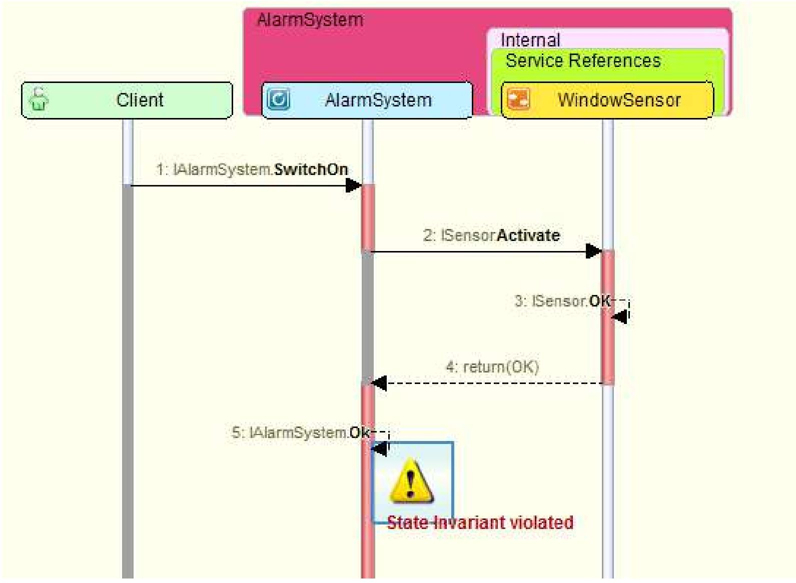
In order to fix the reported error you have to specify a valid action for the trigger for which illegal behaviour is specified.

verum®

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

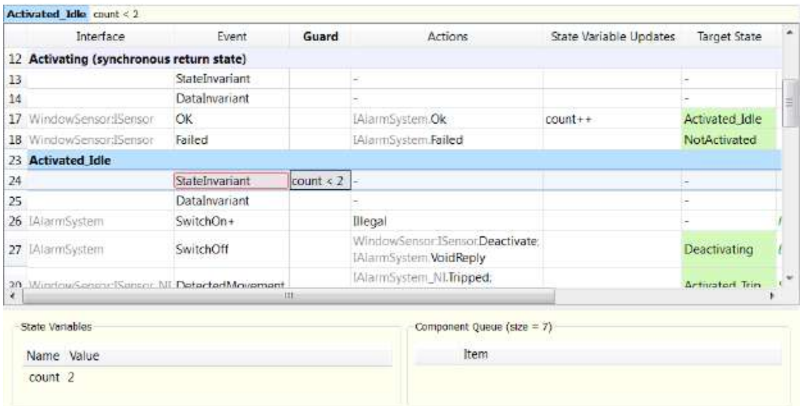
Fixing "State Invariant violated"

This section provides a guideline to fix a modelling error caused by a state invariant violation. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing a trace with a state invariant violation

The following figure shows the rule case for the error reported in the previous figure:

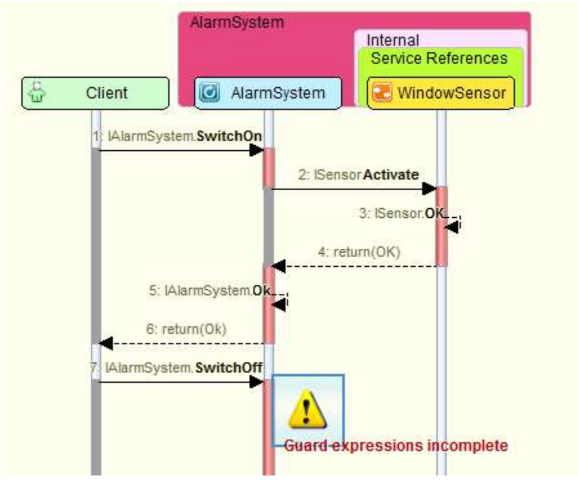


Rule case with specified invariant condition in the SBS

In this example you are shown that the invariant condition specified at line 24 in the SBS does not hold when entering the state 'Activated_Idle'.

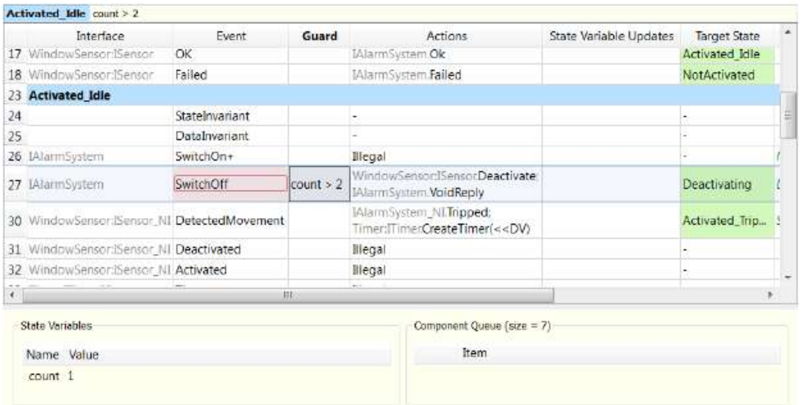
Fixing "Guard expression incomplete"

This section provides a guideline to fix a modelling error caused by an incomplete guard expression. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing a trace leading to an incomplete guard expression

The following figure shows the rule case for the error reported in the previous figure:



Rule case with incomplete guard expression in the SBS

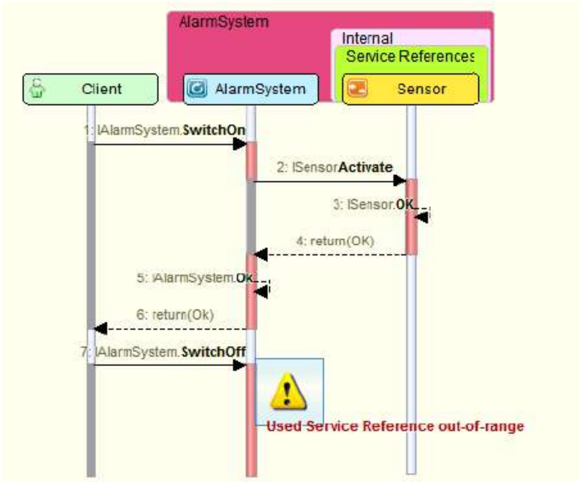
In this example you are shown that the guards on the rules case starting on line 27 does not cover every case that can actually occur at runtime. The guard expression has evaluated to "false" and so there is no enabled rule case for that trigger event.

verum®

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

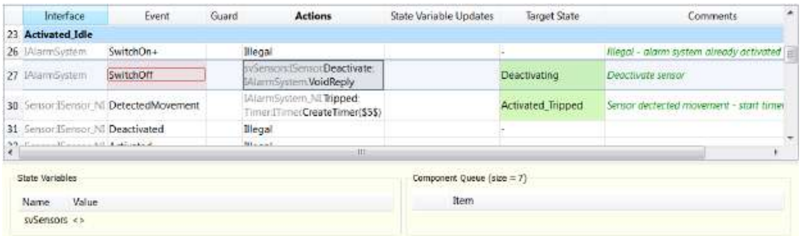
Fixing "Range error" or "Used Service Reference out-of-range"

This section provides a guideline to fix a modelling error caused by range violation. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing a trace leading to range violation

The following figure shows the rule case for the error reported in the previous figure:

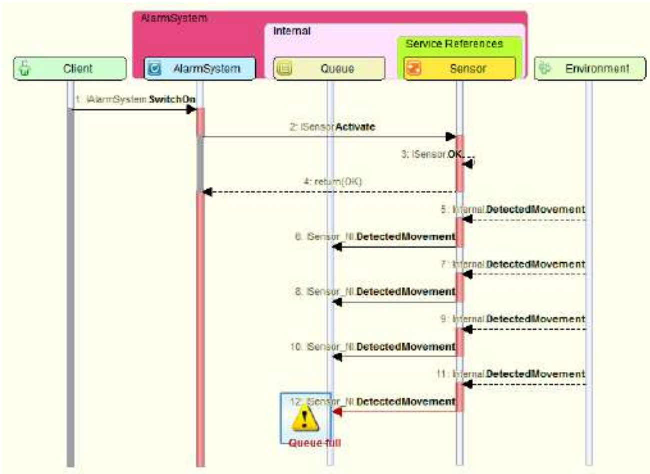


Rule case in the SBS where the range violation occurs

In this example the used service reference state variable, called svSensors, is empty in the state Activated_Idle, when the SwitchOff event is received. The Deactivate action cannot be sent to an empty used service reference, hence the Out-of-range error.

Fixing "Queue full"

This section provides guidelines to fix a queue full error. This error occurs when the component queue of the ASD component (if present) turns out to be blocking, i.e. that the chosen size for the component queue is not big enough to handle all the notifications between the ASD component and its used services. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing a trace leading to a full queue

The following figure shows the rule case causing the error reported in the previous figure:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
13	**Deactivating**					
15	ISensor	Activate+		Illegal		-
16	ISensor	Deactivate		Illegal		-
18	Internal	Deactivated		ISensor_NI.Deactivated		Deactivated
19	**Triggered**					
21	ISensor	Activate+		Illegal		-
22	ISensor	Deactivate		ISensor.VoidReply		Deactivating
23	Internal	[DetectedMovement]		ISensor_NI.DetectedMovement		Triggered

Rule case in the SBS where the queue overflow occurs

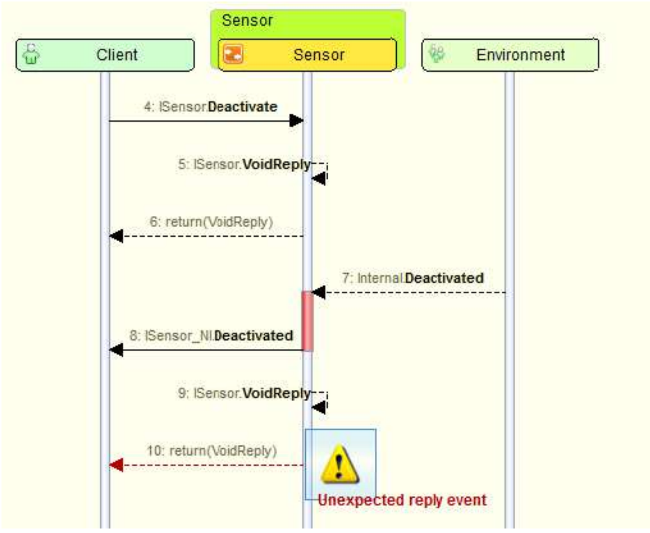
In this example you are informed that the Sensor component sends a large amount of "DetectedMovement" notifications filling up the notification queue of the modelled component.

Note: The real cause of the problem is not in the SBS of the modelled component but in the service specification of the Sensor component, in state Triggered, where an uncontrolled amount of notifications are sent to the modelled component.

verum®

Fixing "Unexpected reply event"

This section provides guidelines to fix an "Unexpected reply event" error. This error occurs when a reply event appears in the trace at an unexpected place, or it is the wrong reply event. For example, not a VoidReply for a Void Call Event or a reply event without a call event. The following figure shows the failure trace:



Failure trace in the "Sequence Diagram" tab showing an unexpected reply event

The following figure shows the rule case causing the error reported in the previous figure:

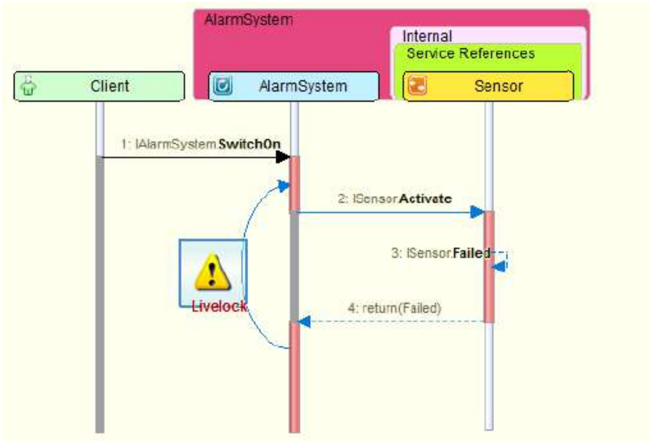
	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Deactivated (initial state)**					
3	ISensor	Activate+		ISensor.OK		Activated
4	ISensor	Deactivate		Illegal		-
7	**Activated**					
9	ISensor	Activate+		Illegal		-
10	ISensor	Deactivate		ISensor.VoidReply		Deactivating
11	Internal	[DetectedMovement]		ISensor_NI.DetectedMovement		Triggered
13	**Deactivating**					
15	ISensor	Activate+		Illegal		-
16	ISensor	Deactivate		Illegal		-
18	Internal	Deactivated		ISensor_NI.Deactivated; ISensor.VoidReply		Deactivated

Rule case in the SBS where the unexpected reply is specified

In this example you are informed that a VoidReply reply event is sent when it is not expected since it arrived already.

Fixing livelocks

This section provides guidelines to fix a livelock. The following figure shows the failure trace for the livelock in the "Visual Verification" window:



The "Visual Verification" window showing the infinite loop

The following figure shows the problem as specified in the SBS of the interface model:

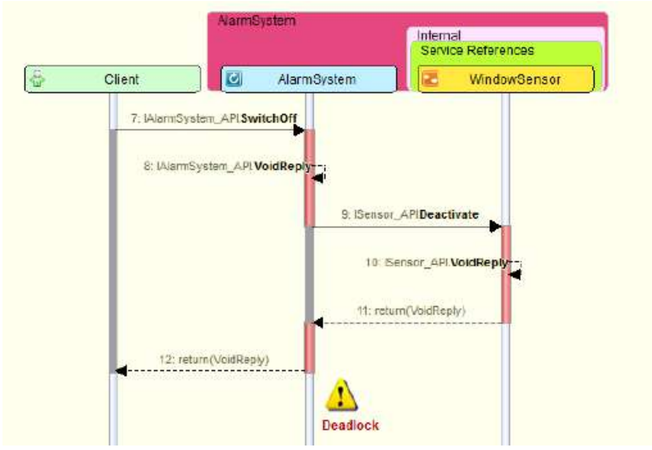
	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**NotActivated (Initial state)**					
4	IAlarmSystem	SwitchOn+		Sensor:ISensor.Activate+		Activating
5	IAlarmSystem	SwitchOff		Illegal		-
8	Sensor:ISensor_NI	DetectedMovement		Illegal		-
9	Sensor:ISensor_NI	Deactivated		Illegal		-
10	Timer:ITimerCB	Timeout		Illegal		-
11	**Activating (synchronous return state)**					
16	Sensor:ISensor	OK		IAlarmSystem.Ok		Activated_Idle
17	Sensor:ISensor	Failed		Sensor:ISensor.Activate+		Activating

Problem causing a livelock as specified in the SBS

In this example the livelock is caused by repeatedly trying to Activate the Sensor when activation fails. This 'loop' is not visible to the outside world. For any client using this service it seems that the component stops working.

Fixing deadlocks

This section provides guidelines to fix a deadlock. The following figure shows the failure trace in the "Sequence Diagram" tab of the "Visual Verification" window:



Deadlock shown in the "Visual Verification" window

In this example the AlarmSystem and Sensor arrive in a state after SwitchOff and Deactivate where no progress can be made.

The following figure shows the problem as specified in the SBS:

SBS of AlarmSystem design model

	Interface	Event	Guard	Actions	State Variable Updates	Target State
9	**Activated_Idle**					
12	IAlarmSystem_API	SwitchOn+		Illegal		-
13	IAlarmSystem_API	SwitchOff		IAlarmSystem_API.VoidReply; WindowSensor:ISensor_API.Deactivate		Deactivating
14	WindowSensor:ISensor_CB	DetectedMovement		IAlarmSystem_CB.Tripped; Timer:ITimer.CreateTimer(5)		Activated_Tripped
15	WindowSensor:ISensor_CB	Deactivated		Illegal		-
16	Timer:ITimerCB	Timeout		Illegal		-
17	**Deactivating**					
20	IAlarmSystem_API	SwitchOn+		Illegal		-
21	IAlarmSystem_API	SwitchOff		Illegal		-
22	WindowSensor:ISensor_CB	DetectedMovement		NoOp		Deactivating
23	WindowSensor:ISensor_CB	Deactivated		IAlarmSystem_CB.SwitchedOff		NotActivated
24	Timer:ITimerCB	Timeout		Illegal		-

SBS of Sensor interface model

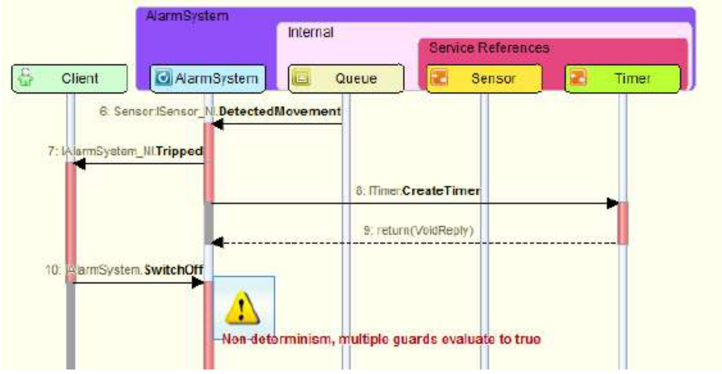
	Interface	Event	Guard	Actions	State Variable Updates	Target State
7	**Activated**					
9	ISensor_API	Activate		Illegal		-
10	ISensor_API	Deactivate		ISensor_API.VoidReply		Deactivating
11	ISensor_INT	Detected		ISensor_CB.DetectedMovement		Triggered
12	ISensor_INT	[DeactivationComplete]		Disabled		-
13	**Deactivating**					
15	ISensor_API	Activate		Illegal		-
16	ISensor_API	Deactivate		Illegal		-
17	ISensor_INT	Detected		Disabled		-
18	ISensor_INT	[DeactivationComplete]		ISensor_CB.Deactivated		Deactivated

Problem causing a deadlock as specified in the SBS

In the SBSs we can see that the AlarmSystem design is waiting for a Deactivated notification event, while the Sensor never sends this event. In this specific case the Deactivated notification event is not sent by the Sensor because the triggering modelling event DeactivationComplete is 'optional' (this is indicated by the square brackets around the modelling event), which means that it may or may not occur. In this specific trace the modelling event never occurs, resulting in a deadlock situation. This can be resolve by making the modelling event 'inevitable' which means that given that nothing else happens, the modelling event will eventually occur.

Fixing non determinism

This section provides guidelines to fix non determinism. The following figure shows the failure trace for the non determinism in the "Visual Verification" window:



The "Visual Verification" window showing the non determinism

The following figure shows the problem as specified in the SBS of the design model:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
45	**Activated_Tripped**					
48	IAlarmSystem	SwitchOn+		Illegal		-
49	IAlarmSystem	SwitchOff	TimerRunning == true	Timer:ITimer.CancelTimer; Sensor:ISensor.Deactivate; IAlarmSystem.VoidReply	TimerRunning = false	Deactivating
50	IAlarmSystem	SwitchOff	TimerRunning	Siren:ISiren.TurnOff; Sensor:ISensor.Deactivate; IAlarmSystem.VoidReply		Deactivating
53	Sensor:ISensor_NI	DetectedMovement		Illegal		-
54	Sensor:ISensor_NI	Deactivated		Illegal		-
55	Sensor:ISensor_NI	Activated		Illegal		-
56	Timer:ITimerCB	Timeout	TimerRunning	Siren:ISiren.TurnOn	TimerRunning = false	Activated_Tripped
57	Timer:ITimerCB	Timeout	otherwise	Illegal		-

Problem causing a non determinism as specified in the SBS

In this example the non determinism is caused by multiple guards evaluating to true after a sequence of events. This non determinism is fixed when the guards are specified such that there is always exactly one guard that evaluates to "true".

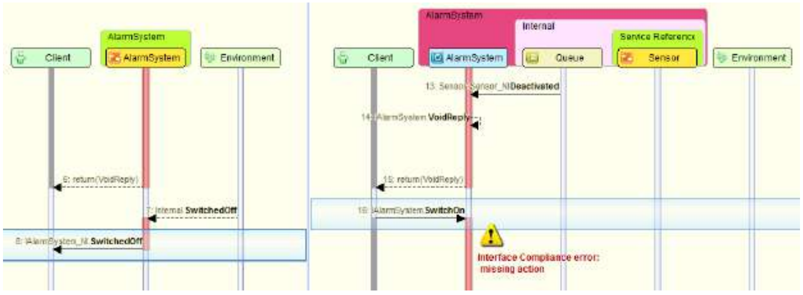
Fixing "Interface Compliance error"

This section provides guidelines to fix interface compliance errors. These errors occur when the design model does not implement its interface model.

In case of an interface compliance error, it can have typically two main causes: it can be a "trace error" or a "refusals error":

- When the interface compliance error is a trace error, this does *not* mean: "The interface tries to do something which is not allowed by the design". Instead, it means: "The design does something that is not allowed by the interface". The corresponding text in the sequence diagram is "Action not allowed".
- When the interface compliance error is a refusals error, this does *not* mean: "The interface tries to do something which is not allowed by the design". Instead, it means: "The design refuses to do something of which the interface says the design must be able to do". The corresponding text in the sequence diagram is "Missing action".
  Note that in this case, the last action in the sequence diagram of the interface is just one possible action that the interface model can do and which is refused by the design; it is very possible that other actions are also allowed and are also refused by the design. In one of the upcoming versions of the ASD:Suite we will show all possible actions that are allowed by the interface and that are refused by the design rather than just one possible counter example.

The following figure shows the "Visual Verification" window with the failure trace:



"Sequence Diagram" tab for interface compliance error

In the previous figure you see the performed sequence of events until the interface compliance error is encountered. In the left pane you can see the sequence of events as specified in the interface model, while in the right pane you see the sequence of events as specified in the design model.

In this example you are informed that, according to the service specification, i.e. behaviour specified in the interface model, the next visible event should be "IAlarmSystem_NI.SwitchedOff", in the "Deactivating" state. In the right pane of the previous figure you see that this "IAlarmSystem_NI.SwitchedOff" does not occur in the "Deactivating" state. Instead the design has already moved on to the "NotActivated" state, and wants to process an "IAlarmSystem.SwitchOn" event. This is called an interface compliance error, i.e. the design does not comply to its implemented service.

The following figure shows the situation as specified in the SBS of the interface model:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
8	**Activated_Idle**					
10	IAlarmSystem	SwitchOn+		Illegal		-
11	IAlarmSystem	SwitchOff		IAlarmSystem.VoidReply		Deactivating
12	Internal	[AlarmTripped]		IAlarmSystem_NI.Tripped		Activated_AlarmMode
14	**Deactivating**					
16	IAlarmSystem	SwitchOn+		Illegal		-
17	IAlarmSystem	SwitchOff		Illegal		-
19	Internal	SwitchedOff		IAlarmSystem_NI.SwitchedOff		NotActivated
20	**Activated_AlarmMode**					
22	IAlarmSystem	SwitchOn+		Illegal		-
23	IAlarmSystem	SwitchOff		IAlarmSystem.VoidReply		Deactivating

Behaviour specified in an interface model which causes interface compliance error

while the following figure shows the situation as specified in the SBS of the design model:

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**NotActivated (initial state)**					
4	IAlarmSystem	SwitchOn+		Sensor:ISensor.Activate+		Activating
12	**Activating (synchronous return state)**					
17	Sensor:ISensor	OK		IAlarmSystem.Ok		Activated_Idle
18	Sensor:ISensor	Failed		IAlarmSystem.Failed		NotActivated
23	**Activated_Idle**					
27	IAlarmSystem	SwitchOff		Sensor:ISensor.Deactivate		Deactivating
30	Sensor:ISensor_NI	DetectedMovement		IAlarmSystem_NI.Tripped; Timer:ITimer.CreateTimer(5)		Activated_Tripped
34	**Deactivating**					
41	Sensor:ISensor_NI	DetectedMovement		NoOp		Deactivating
42	Sensor:ISensor_NI	Deactivated		IAlarmSystem.VoidReply		NotActivated

Behaviour specified in a design model which causes interface compliance error

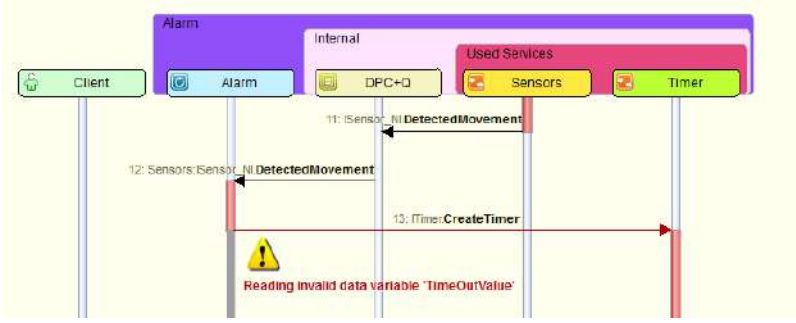
verum®

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

Fixing Data Variable errors

Reading invalid data variable

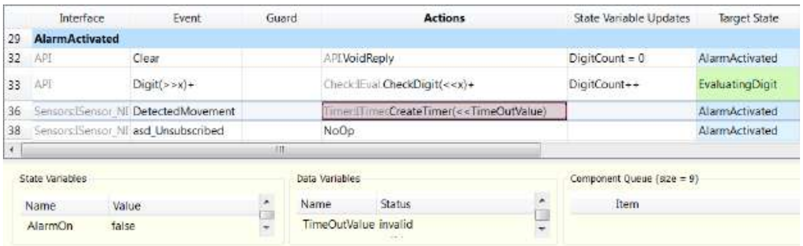
This section provides guidelines to fix a data variable error. The following figure shows the failure trace in the "Sequence Diagram" tab of the "Visual Verification" window:



Data variable error shown in the "Visual Verification" window

In this example the data variable error is caused by the fact that the data variable 'TimeOutValue' is invalid when it is used as an [in] parameter on event.

The following figure shows the problem as specified in the SBS:



	Interface	Event	Guard	Actions	State Variable Updates	Target State
29	AlarmActivated					
32	API	Clear		API.VoidReply	DigitCount = 0	AlarmActivated
33	API	Digit(>>x)+		Check:IEval.CheckDigit(<<x)+	DigitCount++	EvaluatingDigit
36	Sensors:ISensor_NI	DetectedMovement		Timer:ITimer.CreateTimer(<<TimeOutValue)		AlarmActivated
38	Sensors:ISensor_NI	asd_Unsubscribed		NoOp		AlarmActivated

State Variables

Name	Value
AlarmOn	false

Data Variables

Name	Status
TimeOutValue	invalid

Component Queue (size = 9)

Item

Problem causing a data variable error as specified in the SBS

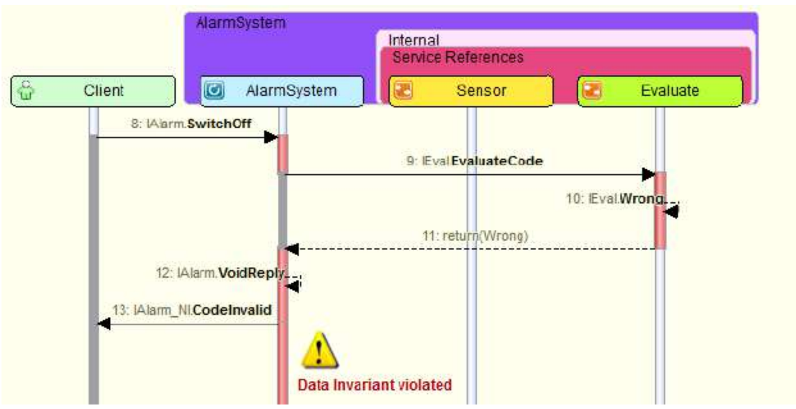
This may be resolved in various ways:

- Make sure that in case the TimeOutValue is received from a client or used component earlier in the design model trace, the value is stored in the TimeOutValue data variable.
- Set the Auto-initialise property in the data variable declaration tab to 'Yes' (only if also at runtime the data variable is initialised upon construction)
- Explicitly Initialise the data variable with an Initialise(><TimeOutValue) event earlier in the trace
- Make sure the data variable is not Auto-invalidated unintendedly and that it is not explicitly Invalidated earlier in the trace

Note: When a data variable is used as an [out] (or [inout]) parameter on a trigger event on a client application interface, the point at which the value of the data variable is read is at the point of the corresponding reply event. This can be several rule cases further on in the trace. In case of an error with thisdata variable, the sequence diagram will show the error at the reply event, not at the corresponding trigger event.

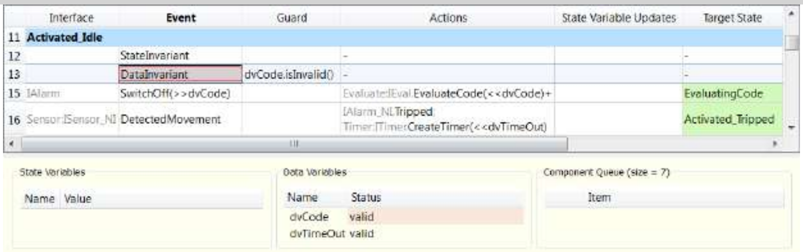
Data invariant violated

A data invariant error is shown in the failure trace as:



Data invariant error shown in the "Visual Verification" window

This data invariant error shows in the SBS as:



Problem causing a data invariant error as specified in the SBS

Here the data variable *dvCode* is valid in the state *Activated_Idle* (as can be seen in the Watch-window), while the DataInvariant in this state specifies that it should be invalid. This can be resolved in various ways:

- Changing the data invariant
- explicitly invalidating the data variable in an earlier state
- setting the Auto-invalidate property for this data variable to true

Fixing model compilation errors

When you see a message in the "Output Window" starting with the following text: "One of the models could not be verified due to an internal error." you are experiencing a model compilation error.

These are the most common causes for model compilation errors:

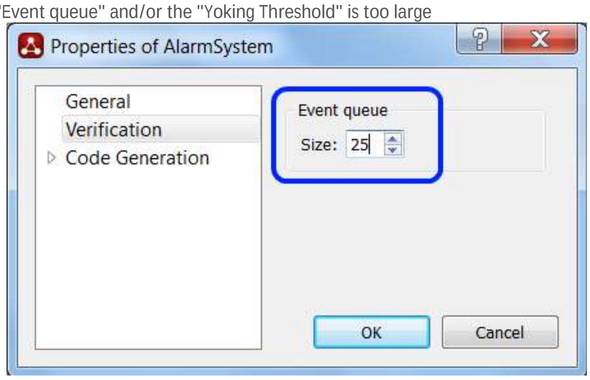
1. An integer state variable is declared with a large range:

	State Variable	Type	Constraint	Cardinality	Initial Value
1	Counter	Integer	[0:100000]	n/a	0
2					

Integer state variable with extremely large range

Note: When updating models build with previous versions of the ASD:Suite the range of existing integer state variables, if not specified, is set to [-2147483647:2147483647]. You are advised to review these ranges and set realistic values. This does not affect your verification, i.e. if you were able to verify the model before updating you will be able to do so after the update too.

2. The "Size" specified for the "Event queue" and/or the "Yoking Threshold" is too large



Example of a large queue size

	Notification Event	Yoking Threshold	Comments
1	DetectedMovement()	15	Sensor has detected movement
2	Deactivated()	[no threshold]	Notification that Deactive has completed
3			

Example of a large yoking threshold

3. A large amount of state variables

	State Variable	Type	Constraint	Cardinality	Initial Value	Comments
1	MAX_NR_OF_SENSORS	Integer	[3:3]	n/a	3	Maximum number of sensors in the system
2	MAX_NR_OF_LEDS_PER_SENSOR	Integer	[5:5]	n/a	5	Maximum number of leds in a rack
3	nrOfsensorsPlaced	Integer	[0:15]	n/a	0	The number of sensors currently placed in the machine
4	nrOfsensorsIndexed	Integer	[0:15]	n/a	0	The number of sensors that have been indexed since the door was opened
5	nrOfsensorsInspected	Integer	[0:15]	n/a	0	The number of sensors that have been inspected since the door was opened
6	nrOfsensorsScanned	Integer	[0:15]	n/a	0	The number of sensors that have been scanned since the door was opened
7	nrOfledsDetected	Integer	[0:20]	n/a	0	The number of leds that have been detected in the sensor being indexed
8	nrOfledsInspected	Integer	[0:20]	n/a	0	The number of leds that have been inspected in the sensor being scanned
9	nrOfledsScanned	Integer	[0:20]	n/a	0	The number of leds that have been scanned in the sensor being scanned
10	indexing	Boolean	n/a	n/a	false	A sensor is being indexed
11	inspecting	Boolean	n/a	n/a	false	A sensor is being inspected
12	scanningsensor	Boolean	n/a	n/a	false	A sensor is being scanned
13	nrOfledProgressEvents	Integer	[0:20]	n/a	0	The number of progress events for scanning a led that have been sent
14	MAX_NR_OF_PROGRESS_EVENTS	Integer	[0:15]	n/a	0	The maximum number of progress events sent for scanning a led
15	scanningled	Boolean	n/a	n/a	false	A led is being scanned
16	nrOfsensorsRemoved	Integer	[0:20]	n/a	0	Indicates the number of sensors that have been removed (Maximum of 2)
17	loginLevel	Integer	[0:15]	n/a	0	Login level: 0 meaning not logged in
18	logoutTimerEnabled	Boolean	n/a	n/a	false	The logout timer is enabled or not
19	doorLocked	Boolean	n/a	n/a	false	Indicates if the door is considered locked
20	loginProcedure	MyEnumType	n/a	n/a	UNKNOWN	Indicates the login procedure being used
21	doorOpen	Boolean	n/a	n/a	false	Indicates if the door is open
22	errorCount	Integer	[0:20]	n/a	0	
23	serviceModeRequest	Boolean	n/a	n/a	false	
24	softwareAvailable	Boolean	n/a	n/a	false	

Example of using a large amount of state variables

The next step, after identifying the cause, is to take corrective measures. This may include reducing the number of state variables, reducing the complexity of the model (possibly by splitting into multiple models), or, for the case noted above, control the increase and decrease of the integer state variable so that the values remain always within the specified range.

In case this does not resolve your problem, please send the models to Verum Support, using the menu option File - Email DM+IMs, for further analysis.

Guidelines for reducing verification time

Verification phases in ASD

ASD carries out design verification in 3 distinct phases:

1. All needed models are sent to the ASD Server and translated to the corresponding mathematical models. This step is short and is influenced only by the number and physical size of the models. It is not affected by the complexity of the models, the numbers of states etc. and is short enough to be neglected.
2. The mathematical models are compiled. The result of this compilation is one or more models compiled for verification.
3. The models are verified by computing the result of the checks listed when the verification starts. This is the "real" verification phase.

verum®

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

Limit the use of state variables

All possible values of the state variables in a model are represented as states in the compiled model for verification. It is principally this reason why state variables in the ASD models are considered only during verification instead of also trying to analyse and verify data computations in general; this would lead to way too many system states to check.

When compiling the mathematical model, the compilation time is proportional to the number of rule cases in the model plus the effect of state variables in the potential state space increase. This time increases quite slowly with respect to the number of rule cases but it increases much more quickly with respect to the number and ranges of state variables. This is because states are created during compilation to reflect every possible combination of values of all state variables.

For example, a design model with 5 state variables of type Boolean potentially increases the number of states being verified by a factor of 2^5. If we add 5 integer state variables each with a range of 0:10, the total potential increase in the number of states is $2^5 * 11^5$. In addition to ordinary state variables, ASD features state variables of type Used Service Reference for manipulating sequences of used services. Because these data types are sequences, their effect on the state space is greater than the effect of other types. For example, a used service reference of cardinality 4, increases the state space by a factor proportional to the number of sequences of cardinality 1, 2, 3 and 4 that can be constructed from the total population assigned to the variable.

When many states have been created to replace state variables, many of these states are unreachable because many of the possible combinations of values never actually occur. These unreachable states are identified and removed before verification starts and this of course adds to the time taken during compilation.

Considering all of the above it is recommended to follow the recommendations listed in the following list:

1. Use state variables sparingly and limit their range as much as possible.
2. Where values in different state variables are mutually exclusive, consider using an Enumeration type to cover all the mutually exclusive combinations and eliminate state variables or reduce the ranges of those that remain.
3. After modifying an ASD model:
    - Review all state variables and remove all those no longer referenced in any guard. State variables referenced only in a state variable update expression or not referenced at all still lengthen the compilation process.
    - Ensure all state variables are declared with their minimum ranges. An integer declared with a range of 0:10 and to which only values 1 and 9 are assigned nevertheless lengthens the compilation phase. Replacing this with a range of 0:2 and thus using the entire declared range is much more efficient and compiles much more quickly.
4. Use state variables of type Used Service Reference sparingly and ensure their cardinality is the minimum required for their purpose.

Limit the use of unsolicited notification events

ASD allows the specification of two types of notification events on component interfaces: *Solicited Notifications* or *Unsolicited Notifications*.

Solicited Notifications have the following properties:

1. They occur as a direct or indirect result of a client invoking an application event; they do not just spontaneously arise.
2. They occur a finite number of times (usually once) per client invocation of an application event.

In contrast, *Unsolicited Notifications* can spontaneously arise from the environment (used services) without any action being required by the client to provoke them. Unsolicited notifications are said to be *unconstrained* if there is no limit in the models on their occurrence. They are said to be *constrained* if the models from which they originate limit the number of occurrences to some finite number.

Notification events have little effect on the compilation time of mathematical models but they can have a big effect on the time it takes to verify a design. This is because every possible ordering of notification events in the component queue must be considered during verification.

The component queue is represented in the mathematical model as a state machine whose state space is all possible orderings, lengths and contents of the queue. As a consequence:

1. *Solicited Notifications* are extremely constrained by their nature and therefore add the fewest states to the component queue.
2. The effect of *Constrained Unsolicited Notifications* on the queue state space depends on how constrained they are. The more of them that can occur without client interaction and the more places in the model in which these can occur, the bigger the increase in the queue state space.
3. *Unconstrained Unsolicited Notifications* cause the biggest increase in the queue state space. The more states in the model where these can occur, the bigger the impact. In practice, such events will have to be yoked to avoid flooding the queue and making it blocking. The yoking threshold is not arbitrary; it should be a true reflection of the relative event arrival rates and service times and should reflect the assumptions that must hold in the real world execution. Yoking will also help reduce verification time, both as a side effect of leading to a reduced queue size and because it reduces the number of orderings the queue can contain.

Considering all of the above it is recommended to follow the recommendations listed in the following list:

1. Minimise asynchronous behaviour in designs. If there is no functional requirement for an asynchronous action, avoid it. This has a bigger impact on the state space to be explored than any other single measure and thus on the verification time.
   This is a worthy design goal irrespective as to whether or not ASD is being used. Asynchronous behaviour rapidly increases the state space and thus the amount of testing required. Asynchronous behaviour makes testing greatly more difficult because of the nondeterminism it introduces. Asynchronous designs are the most difficult designs to make correctly.
2. Minimise the use of unconstrained unsolicited notifications. If these originate from an interface model specifying an interface to a foreign (non-ASD) component (as is frequently the case) then unconstrained notifications can be constrained by adding state variables and guards to the interface model. This of course requires the designer to be able to reason that, for example, if each application performs a given event twice, it is equivalent to performing the event any number of times greater than 2.
3. Modify designs from a "push" model to a "pull" model. The Observer/Publisher feature of the ASD:Suite and the Singleton Event feature provide an easy means for changing to a "pull" model.
4. Modify designs to "flow control" the events by a handshake. This solution should be taken if it is possible that the arrival rates of notification events at runtime could overwhelm the system.
5. Where several different notifications have the same behavioural effect, consider combining them into a single notification event and use a data parameter to discriminate between them.
6. Keep the specified queue size to the necessary minimum.

Apply the Hierarchical Controller Pattern

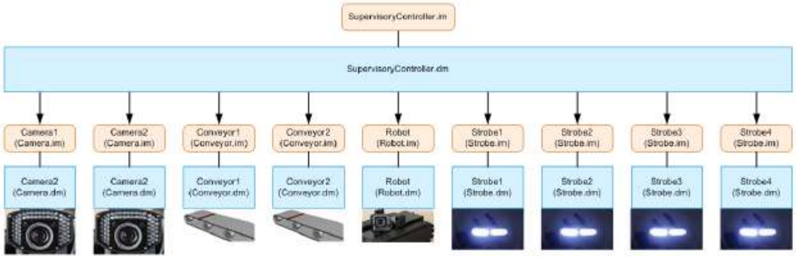
Complexity is a combinatorial game; increasing complexity increases verification time. Complexity in verification is about the number of states and execution traces that have to be explored; this sometimes conflicts with how an experienced software engineer will view the same design. A design that might seem straightforward to an experienced human can nevertheless result in a model with a very large state space to be explored.

When a component becomes very complex, it is frequently (but not always) a symptom of inappropriate abstraction. Partitioning a complex component into a number of simpler components also partitions the state space and will result in several small verifications instead of one, possibly very long one.

Consider an industrial visual inspection system that receives manufactured parts and inspects them with one or two cameras (depending on configuration and type of inspection run), rejecting parts that are defective. Parts arrive on a conveyor and are picked up by a robot, held in the field of view of the cameras, inspected and then returned to the conveyor or dropped into a rejection bin.

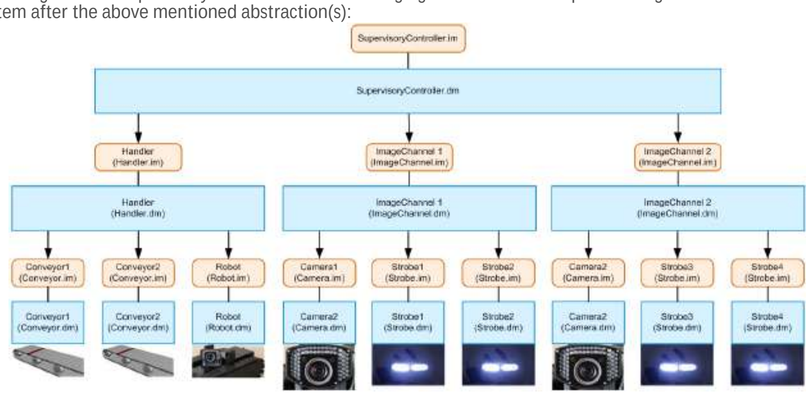
A Supervisory Controller for such a system must receive signals from the conveyor announcing the arrival of parts to be inspected, it must control the robot, the cameras, the light sources and cater for being starved (no parts to inspect) or blocked (the robot cannot unload the current part because the next machine in the line is not ready to receive it). There are a number of approaches that could be taken:

1. The controller could be implemented as a single component directly controlling 2 conveyers,, one robot, two cameras, four light sources, together with their associated input and status signals. Such a controller will be very complex and *fragile*; any change to the externally visible behaviour of the conveyers, cameras, robot, or light sources will result in changes to the controller. The following figure shows the component diagram for such a system:



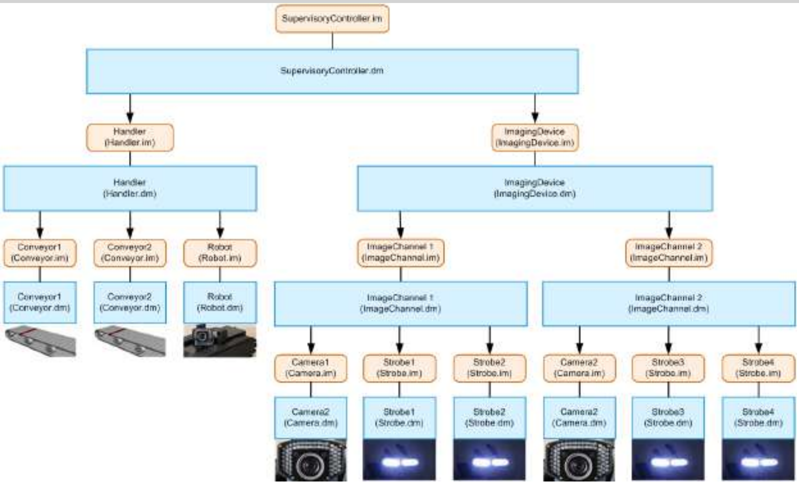
Component diagram for industrial visual inspection system - initial thoughts

2. We could apply abstraction and hierarchical control principles as follows:
   - The robot, two conveyors and associated signals could be combined into an abstraction called "Handler" that takes care of all coordination between the robot and conveyors and handles all the associated input signals. The Handler component performs all control actions and offers a simpler interface to the Supervisory Controller. All interactions between the robots and the conveyors are no longer visible to or handled by the Supervisory Controller.
   - Combine one camera and two light sources together to form an abstraction called "Image Channel" that takes care of all actions needed to acquire images, reset CCDs between images, control integration time, cycle the strobe lights used as light sources and synchronise exposure with strobes. The Image Channel component performs all actions specific to the camera and its strobe lights. Its interface to the Supervisory Controller will be smaller and more abstract, leading to less complexity in the Supervisory Controller itself.
   - The Supervisory Controller in such a system now has to control one Handler and one or two Image Channels. Such a design is also more *robust* in the face of change; changes to the details of the camera interfaces and configuration of strobe lights, for example, no longer require changes to the Supervisory Controller. The following figure shows the component diagram for the industrial visual inspection system after the above mentioned abstraction(s):



Component diagram for industrial visual inspection system - abstraction - first phase

   - We may further refine this abstraction by introducing an "Imaging Device" as an abstraction between the Supervisory Controller and the two Imaging Channels. This Imaging Device abstraction presents a device controlling one or two Imaging Channels and handles all coordination between them, further reducing the complexity of the Supervisory Controller. It localises implementation knowledge as to how the two Imaging Channels are realised and whether or not they are both present. The following figure shows the component diagram after this refinement:



Component diagram for industrial visual inspection system - abstraction - refinement

This leads to reduced verification because the Handler, Imaging Device, Image Channels and Supervisory Controller can now be verified independently.