Analytical Software Design (ASD) is a new patented technology developed by Verum. It is an application of model-driven design with formal methods (MDD-FM) in which industrial-scale software systems are constructed from mathematically verified software components. It enables software specifications and designs to be mathematically verified at design time, before any code is written. When the design has been verified, code is generated from the verified design. This technology is incorporated in a software design platform called the ASD:Suite.
Code generation is not new; models can be made, for example, in UML or some other notation and code can be generated from them. But how do we know the model is correct? And if we don’t know this, how do we know if the generated code is correct? And if we don’t know this, we are confronted with a testing problem.
ASD:Suite uses the ASD Modelling Language to make two kinds of models, interface models which serve as the specification of the externally visible, required functional behaviour, and design models, which specify the implementation of the design. Although ASD models have no visible mathematical notation and are thus accessible to all project stakeholders, they are sufficiently precise that mathematical models can be generated from them automatically. Where the construction engineer uses his tools to generate a finite element analysis model of his design, the software engineer uses the ASD:Suite to generate a process algebra model of his design. This is then mathematically verified against the specification by applying ASD verification. Design errors uncovered by this verification are removed by the designer by updating the ASD models. These models are then verified again, the process being repeated until all errors have been removed. At the end of the process, we know that the ASD design model is correct and complete before writing a single line of code. The ASD:Suite is then used to generate the corresponding implementation in Java, MISRA C, C++ or C#.

One of the most important aspects of the ASD technology is the semantic equivalence between what is specified in the ASD models, the formal representation of those models used in model-verification and the runtime behaviour of the generated source code.

Because we know the design is correct and complete, we know the generated code is similarly correct and complete and does not need testing. Note that this is not an argument for skipping testing; developing software with the ASD:Suite means that the software enters the testing phase with a very low level of defects and thus testing becomes more effective and meaningful. Instead of trying to “test in” quality, testing becomes an exercise of measuring quality. Testing can focus more on validation of the system.

ASD design-time verification is particularly effective at finding and removing the type of defects most difficult to find by conventional testing. And because it removes them early in the development life cycle, it significantly cuts development cost.
The application of ASD originally started in the domain of “control” as found in embedded software. As a result in the ASD terminology the term “behaviour” is used a lot to express the process of making decisions.

For other domains than embedded software however ASD can be applicable too. Any decision logic can be expressed in an ASD model.

The Domain for ASD

ASD can be used to model:

- **Control (reactive system)**
  - Model the state of a system and all discrete events that can happen and reaction to it (behaviour)

- **Work-flow (business interaction)**
  - Where are we in the process, which tasks have been performed, and which still needed etc.

- **Product life cycle management**
  - Status of a product (new, backorder, delivered, paid, returned, etc.)

- **Etc.**
ASD:Suite Main Concepts

- Based on component-based design
- Models describe all component interaction as discrete decisions on discrete events
- Formal verification (underlying formal techniques are hidden for user)
- Separation of data and control

The main concepts used within ASD:Suite are explained in more detail in the following slides.
A system is decomposed into a number of components that interact with other components. A component can be decomposed into multiple components too. These components can then interact with all other components within the parent component as well as the other components of the entire system.

Component interaction is specified as the service of a component; an action is triggered by a client that calls a function or triggered by a used component that sends a notification. The service describes:
- which interfaces there are
- which functions and notifications are provided on each interface
- what the signature of each function and notification is
- what the semantics of the possible actions are

An ASD component is a functional unit that is described by:
- a specification of its externally visible behaviour (what the component must do); this is specified in an interface model.
- a design how this external behaviour is accomplished making use (parts of) its environment; this is captured in a design model.

This design model is verified for correctness with respect to its specification. In other words, ASD verification checks whether the design correctly executes its specified external behaviour, as specified in the interface model. When the design model is verified and found to be correct, the source code generated from the design model implements the specified behaviour.

An ASD component can be a singleton or non-singleton.

Note that ASD is not a “programming technology”. Not all constructs that are possible in programming languages are available in the ASD:Suite. The ASD:Suite only uses constructs that are formally verifiable.

Note that during the engineering of a system it is likely that an architecture needs refinement to improve verifiability, decoupling, separation of concerns and so on. This process is normal and is not necessarily a sign of an immature architecture.
As indicated before, an ASD component consists of both an interface model specifying the externally visible behaviour of the component and a design model specifying how that external behaviour is accomplished with the use of other components in its environment.

An interface model contains the declaration of the interface calls i.e. the interface signature of the component, including the relevant parameters. A design model uses these definitions from the Implemented Services and Used Services to define the internal behaviour of the ASD component.

ASD models are target language independent with some exceptions.

- The interface declaration, and more specifically the parameter type declaration is language dependent. This is because there is no language independent way to capture all possible user defined types.
- Since generated C code uses a static memory model, not all dynamic features of the ASD language are applicable.

These differences are indicated in the course materials where applicable.

For an ASD component that uses handwritten and legacy code (in ASD known as ‘foreign components’), interface models must be made for verification purposes.
An ASD component implements call and return style semantics, meaning that when a client makes a call on an ASD component, this component processes the call to completion and then replies to the client. Only after the reply, the component is ready to receive a next call from a client.

Replies of an ASD component can be of void or valued type, where a valued type indicates that the call ends with a return value which in turn may indicate the result of processing the call. For example return values ‘OK’ and ‘Fail’ may indicate whether a call was processed successfully or not.

A synchronous action is only a call – reply sequence.

An asynchronous action is a call – reply sequence followed by a notification. Because the client has full control over the occurrence of this notification it is a solicited notification.

A notification can also be unsolicited. In this case the notification occurs uncontrolled in the sense that the client does not know when to expect the notification; it can happen any time without client asking for it. This is called an unsolicited notification.

An unsolicited notification event can be constrained by its service specification. An example for this is a movement sensor that post a notification once when it detects movement for the first time and ignores any other occurrences afterwards.

And it can be unconstrained. An example for this is a temperature sensor that posts new values every second i.e. continuously.
A component is used by a client. In the following action diagrams we will use the terms “Client” and “Component”. Note that a component can act as client too.

A synchronous call (Call Event in ASD) is where the client synchronously invokes a method at the component's service and remains blocked until the component has completely processed the request (serialising call and return semantics). The component returns either a “void” reply (VoidReply in ASD) or a reply value (Valued Reply Event in ASD).

As long the client is blocked it cannot invoke other methods at the same or other components.
An **asynchronous** call (Call Event) is where the client synchronously invokes a method at the component's service and remains blocked until the component sends a reply that it *is going to process* the request (serialising call and return semantics). The component returns either a "void" reply (VoidReply) or a reply value (Valued Reply Event).

As long the client is blocked it cannot invoke other methods at the same or other components.

Eventually the component asynchronously informs the client about the result of the call through a notification (Notification Event).

After the synchronous reply, the client and component continue to run concurrently, i.e. the client is able to perform other tasks while its request is being processed by the component.

Note: this is *not* an asynchronous call in the sense that the method call is decoupled from the client via a queue-mechanism.
Unsolicited Notification

- Example:
  - Notification Event: "TemperatureTooHigh"

Example Diagram: 

- Client
- Component
- Notification arrow from Component to Client
An ASD component interacts with its environment in a client-component relationship, where the client component calls a method on the Application Interface of the component, and the component responds with a void or valued reply on the same interface.

If the component is processing an asynchronous call, it eventually will asynchronously invoke the client with a notification carrying the result of the asynchronous call. The notifications are executed in a decoupled way: the component synchronously posts the notification in a queue of the client. In case of Standard Execution model, the client component has a thread for serving outstanding notifications. In case of the Single-Threaded Execution model the thread is temporarily borrowed from the component by the client for serving the queue or from the component calling the client component. The queue is an integral part of the client component.

The user can specify the maximal length of the queue as part of the verification properties; the default value is 7. ASD verification checks whether there are no situation where there are more elements in the queue than the maximal number allowed. For Tiny-C and C this maximal queue length is used during code generation to define the amount of space to be allocated for the queue. For C++, C# and Java the queue is a dynamic structure.

Because of the decoupling between client and component, a subsystem consisting of ASD verified components will not deadlock when the individual components are verified deadlock-free by using the ASD:Suite.
ASD is all about behaviour of components. It specifically (but not exclusively) can be used to build complex, event driven, reactive and concurrent systems. This includes embedded controllers, device drivers, interface protocols, etc. It is suitable for developing software for any behavioural system that reacts to inputs from its environment.

An ASD model specifies all possible behaviour. This encompasses normal functionality, but also all exceptional cases. An ASD model is therefore a complete specification of the component’s behaviour. This includes a specification of which external input from the environment is allowed and which is not. Disallowed input is referred to as “illegal”. ASD verification checks that an ASD component never exhibits illegal behaviour towards its environment.
Here we describe some important definitions to be used when creating a Sequence Based Specification.

**Trigger**: a Call Event, Notification Event or Reply Event that is **input** to the component

**Action**: a Call Event, Notification Event or Reply Event that is **output** to the component

Call and reply events have a call & return style semantics. The call event indicated the start of a function execution and the reply event indicated the end of the execution.

Triggers and actions are seen from the component perspective. So in a client-component relation between two components, an action from the client is a trigger to the component and vice versa.
The model editor in the ASD:ModelBuilder helps you construct and edit Sequence Based Specifications (SBS). These enable you to define what an action will be for each possible trigger on the interfaces of a component, in every possible state.

The example shows the SBS for the service specification of the Siren component, which is an ASD interface model. It consists of one interface ISiren_API.

On this interface the possible triggers are TurnOn() and TurnOff(). These calls are void methods without parameters. In the conventional software programming world these are the signatures of the interface calls.

In the state SirenOff the trigger TurnOn() will return a void reply to its caller and make a state transition to the new target state which is SirenOn.

In the state SirenOff the trigger TurnOff() is not allowed. By specification, it is illegal behaviour because a user does not find it logical to switch off the siren if it is already off.

In the state SirenOn the trigger TurnOn() is not allowed. By specification it is illegal behaviour because a user does not find it logical to switch on the siren if it is already on.

In the state SirenOn the trigger TurnOff() will return a void reply to its caller and make a state transition to an existing target state being SirenOff.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Event</th>
<th>Guard</th>
<th>Actions</th>
<th>State Variable Updates</th>
<th>Target State</th>
</tr>
</thead>
<tbody>
<tr>
<td>SirenOff</td>
<td>&lt;&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ISiren_API</td>
<td>TurnOn</td>
<td></td>
<td>ISiren_API.VoidReply</td>
<td></td>
<td>SirenOn</td>
</tr>
<tr>
<td>2 ISiren_API</td>
<td>TurnOff</td>
<td></td>
<td>Illegal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 SirenOn &lt;ISiren_API.TurnOn&gt;</td>
<td></td>
<td></td>
<td>ISiren_API.VoidReply</td>
<td></td>
<td>SirenOff</td>
</tr>
<tr>
<td>4 ISiren_API</td>
<td>TurnOn</td>
<td></td>
<td>Illegal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 ISiren_API</td>
<td>TurnOff</td>
<td></td>
<td>ISiren_API.VoidReply</td>
<td></td>
<td>SirenOff</td>
</tr>
</tbody>
</table>

The state SirenOn is not a real state but a transition state.
The SBS can also be represented through a State Transition Diagram (STD).
ASD:Suite is about behaviour; it makes a strict separation between control and data. This means data can be passed between components but if you want to change the behaviour of a component based on data, you must explicitly describe this in the models.
The ASD:ModelBuilder is the ASD:Suite Windows desktop application. Visual Verification and Code Generation are hosted services that are accessible through the ASD:ModelBuilder. Code Generation is also accessible through the ASD:Commandline Client.