Scade Tutorial

Embedded Systems Simulation and Verification

February, 2005

Authors:
Jonas Elmqvist
Jerker Hammarberg
Simin Nadjm-Tehrani
Diana Szentiványi
1. Introduction

1.1 Scade in the development process

One goal in this course is to get insight in the process of specifying a system and analysing the design with respect to its requirements. There are a number of different software tools available that support this process. In this course, you will have the opportunity to use the commercial specification support tool Scade used for example by Airbus in the avionics industry. Scade will be used to create a detailed design specification that completely models the functional behaviour of the system. This high-level design model documents the decisions of the designer. Scade provides both simulation and formal verification of the design model.

The basic scenario in the course is illustrated in figure 1:

![Figure 1. Development process](image)

The scenario aims at representing a real-life project where the user’s requirements have been captured into some user requirements documents that may be incomplete and contain ambiguities. This document is taken by the modeller as input into the requirements engineering work. Parts of the resulting system requirements specification document are then taken as a basis for system design with Scade. The basic steps in system design are:

- Build the architectural design.
- Define the functional behaviour of each component in the Scade formalism.

The formal Scade model can then be verified with regard to some properties, derived from the system requirements. This can be done with Prover Plug-In, a powerful logic tool that is built into Scade. The basic steps in formal verification are:

- Determine and formalize a set of properties to verify.
- Prove that the design satisfies the properties.

The output from the formal verification tool is used to iteratively correct and modify the design and improve the requirements until they are considered complete.

1.2 The design model in Scade

The system design specification defines the architecture and functionality completely and unambiguously. There are different kinds of design models that are suitable for different kinds
of systems. A common approach is to provide a state based description (supported by many design languages e.g. Statecharts). This approach is useful for systems for a rich control structure and few data-dependent complex computations. The system is described by its states and by how the inputs cause transitions between the states. Another approach is to define a data-flow model, useful for systems with less complex control structures but many data-based computations. A data-flow model describes the system by how signals flow from input through the components to output. Most systems are of course naturally described by a combination of the above two approaches.

A common way to do design is to create a design model in a computer tool. In this case the tool is Scade which has the data-flow synchronous language of Lustre (Halbwachs 1993) as an underlying language. The tool supports the development of discrete controllers with a deterministic behaviour (that is, for any input and state there is a unique output determined by the design), and can therefore be used to run a simulation of the model.

A Scade model is built from hierarchical block diagrams. The hierarchical elements are called nodes. A node has inputs and outputs, and its functionality can be defined by a block diagram that contains other subnodes. The actual system interface is defined in a top node.

The nodes itself defines the functionality (or behaviour) of the corresponding (sub)system. Nodes can either be graphical or textual which give the designer two different ways of modelling. When modelling behaviour with a graphical node, the user may use the predefined operators (mathematical, logical) that are found in the toolbar. When modelling the behaviour in a textual node in Scade, the designer uses the synchronous language of Lustre. More on this will be described in section 2.2 and for more information about Lustre, see the material on the course webpage.

Scade presents the model in a graphical user interface (see Figure 2). The hierarchical structure of the system can be viewed in the Workspace window under the tab Framework and each model can be opened in the Edit window, both for graphical modelling and textual modelling. All the built in operators can be view in the Toolbars and information to the designer (such as errors, compilation progress etc.) is presented in the Output window.

Figure 2. Scade Graphical User Interface
1.3 ATM Requirements

Your task in this tutorial is to design a simple ATM in Scade in order to learn the basics of the tool. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The ATM can be a very complex system since there are demands of secure transactions, communication with the central system and handling multiple accounts and users etc. However, in order to get you started in Scade, the ATM you will design will be a simplified version. First of all, you will only consider the case of one user, i.e. one bank account and one credit card. Thus, the bank account can be modelled inside the ATM with the PIN-code “hard coded” in the design and you will not need to model a central system.

A human operator operates and interacts with the ATM. Since the ATM only handles one bank account, the only input for identification is a four digit PIN-code. The operator should be able to withdraw money or ask for a balance receipt. The ATM needs to satisfy the following requirements:

- Correct identification of the user, i.e. checking the PIN-code.
- Withdrawal of a specified amount of money (if requested amount is less or equal of the balance in the account, i.e. no negative balance is allowed).
- Print a balance receipt i.e. sending the current balance of the account as output.
- Abort operation, i.e. if an abort signal is present, no money should be withdrawn from the account and no receipt should be printed.

Your job is to identify different subsystems within the ATM that takes care of the different functions in order to fulfil the requirements. If you think the specification above is vague, make your own design decisions, but you should be able to motivate your decisions. Feel free to add extra functionality to the ATM, such as multiple accounts or possibility to deposit money to the account.
2. System design using Scade

2.1 Overview

This tutorial gets you an introduction to Scade and gets you started in the design specification of the automatic teller machine (ATM). An ATM accepts a cash card, interacts with the user, checks for authentication, dispenses cash, and prints receipts. This tutorial will get you started in the design of the ATM in Scade and your task will be the following:

- Create an initial Scade model to start from by following the steps of the tutorial of section 2.2.
- Identify the functional components of the ATM and draw a simple block diagram of your architectural design. You may use any graphical editor that you prefer, including Scade. Show to the course assistant.
- Design the ATM according to the requirements in section 1.3.
- Simulate the design using the Scade simulator.
- Show the design and the simulation to the course assistant.

2.2 Getting started with Scade

This is a short Scade tutorial that introduces you to the basic techniques and gives some helpful hints to avoid the most prominent problems that might arise otherwise. In addition, by following the tutorial, you will produce a Scade project that will be the basis for the rest of the design of the ATM.

The first and most important hint is: Don’t expect everything to work perfectly! This is part of the reality that the course wishes to give some insight into. In particular, do not use global variables since they work poorly in Scade.

Starting Scade

1. Before starting Scade for the first time, start Visual C++ by selecting Programs → Microsoft Visual C++ 6.0 → Microsoft Visual C++ 6.0 in the Windows Start menu, then quit it again (File → Exit). This registers some necessary keys in the Windows registry.
2. In the Windows Start menu, select Programs → SCADE Suite → SCADE.

Using the documentation

The documentation comes in two forms: The on-line help in the Help menu of Scade and the more extensive pdf documents. The latter can be accessed through the Windows Start menu, then Programs → SCADE Suite → Documentation → Manuals. The Scade manuals are very extensive and useful so be sure to get familiar with them. If you have questions, try to find the answer in the manuals before asking the course assistant.

Some useful settings

1. In the Tools menu, select Options...
2. Check Save before running tools.
3. Activate Automatic Backup.
4. Click OK.

Creating a new project

1. In the File menu select New... .
2. Make sure that the Project tab is in front and choose Scade Project. In the Location field, enter or browse for a folder in your home directory, for instance z:/TDB30 .
3. In the Project name field, enter ATM.
4. Click OK, then Next, then Next and finally Finish, without changing anything in the forms.

Creating a node

In this project, you will create one node called n_ATM that will be the top node for the ATM system itself. It will be your job to define subnodes to n_ATM and to define the ATM's software functionality through them.

1. Click the Framework tab at the bottom of the view to the left.
2. Expand ATM.etp, then ATM in the tree that appears by clicking the + signs.
3. Select the Operators folder by clicking it.
4. The New Node button in the toolbar will now be enabled. Click it. A new node appears in the Operators folder. Or right click the Operators node and choose Insert- >Node.
5. Click node1 directly below Operators and press F2 to rename it. Enter n_ATM and press Enter.

Hint: Do not rename the equation (eq_Node1_1) below the Interface folder. The reference to the node number (in this case 1) may be helpful later when debugging.

Additional nodes are created by repeating steps 4 and 5 above. All the subsystems of the ATM (for example an bank account subsystem, n_Account) should be defined as individual nodes that interact with each other.

Defining inputs and outputs

The inputs and outputs of the n_ATM node will be the only interaction between the system and the operator.

1. Select the n_ATM node in the Framework View by clicking it.
2. The New Input button in the toolbar will now be enabled. Click it. A new input signal appears in the Interface folder of n_ATM. You can also right click the Interface folder and choose Insert -> Input
3. Click Input1 directly below Interface and press F2 to rename it. Enter i_Widraw and press Enter.

Additional inputs for n_ATM are created by repeating steps 2 and 3 above. Outputs are created similarly by clicking the Create Output button or by right clicking the Interface folder and choose Insert -> Output .

1. Create additional input signals for n_ATM: i_PIN_code, i_Abort, i_Balance and i_WidrawAmount.
2. Create outputs signal for n_ATM named o_Balance, o_Abort and o_Cash.

*Note: These inputs and outputs should not be considered to be fixed for n_ATM – you may wish to modify or extend the interface at a later time if needed.*

**Signal Types**

Every signal must have a type. By default, the type of new input and output signals will be the bool type. There are both primitive types (bool, char, complex, int, real, string) and user defined types. Among the different kinds of user defined types, enumeration types define a (small) set of names for values. However, because of problems with the Prover Plug-In, avoid using enumeration types in your models and use integer values instead.

**Setting types for interface signals**

1. In the View menu, select Properties... or press Alt-Enter. The Properties Window will appear.
2. Select the i_Withdraw signal of the n_ATM interface. The properties for the i_Withdraw signal will appear. You can also right click the signal itself and choose Properties...
3. Set the type of i_Withdraw to be bool by selecting it in the Type drop-down list.

The types of the other signals of the n_ATM interface are set by repeating steps 2 and 3 above.

4. Set the types of the signals of n_ATM according to the following:
   - i_Abort: bool
   - i_WithdrawAmount: int
   - i_Balance: bool
   - i_PINcode: int
   - o_Abort: bool
   - o_Balance: int
   - o_Cash: int

**Defining functionality with block diagrams**

As described in "Creating top nodes" above, the n_ATM node is the top node and will contain all the subsystems with its inputs and outputs connected to the subsystems.

- In the Framework View, double click n_ATM. The Network View window will appear inside the Edit Window.

*Hint: To make it easier to keep the diagrams tidy, you may wish to use the grid, to which every block diagram element will be snapped. In the Network View, right-click and check Grid, then right-click again and check Snap to Grid. Another option is to use the buttons in the Layout toolbar.*

1. Drag all the inputs of n_ATM from the Framework View into the left side of the Network View. You will have to do it one by one.
2. Drag the output of n_ATM in the same way to the right of the view.
3. Drag all the subsystem nodes of the ATM to the middle of the Network View. You may resize them by selecting it and using the resize handles that appear.
4. Inside, the Network view, connect the inputs of n_ATM to the corresponding inputs of the subsystem nodes by first clicking on the edge of the arrow and
then on the corresponding pin of the node box. You may move the edges of the connections by selecting it and using the move handles that appear.

5. Connect the outputs in the same way.

Note: If you later need to feed back signals in a block diagram remember that cyclic dependencies are not allowed in Scade. You will have to use the PRE operator (under Time in the Shortcut Bar) which effectively delays the signal one time step.

Defining a constant

The ATM should check if the PIN code from the user is identical to the PIN code for the bank account. This PIN code can be defined by a global constant c_PIN of type int.

1. In the Framework view, select the Constant Blocks folder.
2. The Create Constant Block button in the toolbar will now be enabled. Click it. A new constant block appears in the folder.
3. The Create Constant button in the toolbar should now be enabled. If it isn’t, select ConstBlock1 in the Framework View. Click the Create Constant button. A new type appears in the ConstBlock1 folder.
4. Rename the variable by entering c_PIN and pressing return.
5. Double click ConstBlock1. A window with the constants in the block appears.
6. Change the type of c_PIN to int and its value to a arbitrary 4 digit code (e.g. 1234).

Defining functionality with the Scade textual language

Block diagrams is one of three ways to define functionality in Scade; the other two are the Scade textual language and state machines. The state machines will not be covered by this tutorial, but you are encouraged to take a look at it in the documentation, since they are useful in several types of applications. For example, if you want to model a more complex behaviour of the ATM you might want to use a state machine and modelling all the states of an ATM (ready, waiting for PIN-code, busy etc.). However, for this simple version of an ATM, you may model it only by data-flow models.

This section gives an example how to define a small part of the functionality of the ATM in the Scade language. The part described is the node that models the bank account (from now on refered to as n_Account) which has one input i_WithdrawAmount and one output o_Balance. One of the functionalities of this node is to keep track of the balance in the account and emit the balance through its output signal o_Balance.

1. Add output o_Balance of type int to n_Account.
2. Add input o_WithdrawAmount of type int to n_Account.
3. Bring up the Properties Window for n_Account.
4. In the Kind of node drop-down list, select Textual.
5. Close the Properties Window.

The Scade language defines the functionality of a node by describing equations relating the signals and variables for each time step. Each line in an equation block is an assignment to an output signal or a (local or global) variable.
The following equation block defines parts of the functionality of \textit{n\_Account}:

\begin{verbatim}
let equa eq_Node2_1 [ , ]
  o_Balance = (1000) -> ((pre (o_Balance) - i_WithdrawAmount));
end;
\end{verbatim}

The second line updates \textit{o\_Balance}. The first element after the = sign defines the initial value of the signal, in this case the bank account is initialised with 1000. The \texttt{-} operator separates the definition between the first and the next time step. The \texttt{pre} operator gives the value of a variable or signal of the previous time step. Notice that unlike many other programming and design languages, it is not valid to increase a variable with a line like

\begin{verbatim}
o_Balance = o_Balance - i_WithdrawAmount;
\end{verbatim}

This is in fact a false equation (unless \textit{i\_WithdrawAmount} is 0) and will be rejected in the semantic check.

1. Enter the definition of \textit{n\_Account} in the Textual View.
2. Right-click in the Textual View and select \textit{Apply}.

\textit{Note:} Always select \textit{Apply} after each time you have edited a textual node. If you don't, you may lose your changes without warning.

This is only a part of the functionality of the \textit{n\_Account} node. You will need to add extra functionality to this node, e.g. checking that balance is greater of equal to zero. This is done by adding extra line of codes in the textual editor or modifying the Lustre expression of line 2.

To feed the signals of \textit{n\_Account} into \textit{n\_ATM}, add an instance of \textit{n\_Account} in the network view of \textit{n\_Account}, and then connect the output of \textit{n\_Account} to the corresponding output of \textit{n\_ATM}.

\textbf{Running a simulation}

1. In the Framework View, select the node that you want to simulate. This means that you can simulate individual nodes if you want to check the behaviour of individual subsystems of the ATM.
2. Click the \textit{Simulate} button in the toolbar. Reply \textit{Yes} if Scade ask about applying or saving changes. The node will be compiled via Lustre and C to machine code. You may have to correct errors and try again.

\textit{Hint: The error messages may be difficult to interpret. The Quick Check tool in the Project menu may find some of the errors and gives better error messages.}

3. Click the \textit{Simulate Node} button in the toolbar. The simulation will start.
4. Switch to Instances View by clicking the tab below the Framework View.
5. Expand the tree in the Instances View so that you can see the interfaces that you are interested in.
6. To perform one time step, click the \textit{Step} button in the toolbar.
7. To change an input value for the following time steps, select the input signal in the Instances View, press F2, enter the new value and press Return.
8. To stop the simulation and go back to the Framework View, click the \textit{Stop} button in the toolbar.
3. Formal Verification (Prover Plug-In)

3.1 Introduction

In this part of the project you learn to apply mathematical analysis to discrete models of a system under the design phase. This process is called formal verification. Compared to other forms of mathematical analysis you have performed earlier, the methods applied here aim to automatically prove properties of a system using methods based on mathematical logic.

Thorough verification of the system is naturally very important, particularly for safety-critical systems, where faults can lead to serious human injury or damage of equipment. In fact, verification may be the most time-consuming phase in the development process. Simulation and testing are by far the most widely used verification methods in the industry today. Simulation is done on a model and testing is done on the finished implementation, but both have in common that the engineer feeds some test data into the system and checks whether the produced output is as intended.

Formal verification will allow you to concentrate on properties that were not verifiable using simulations. When simulating, you tested the outputs that the system generates for given inputs. When performing formal analysis, you are often interested in output that is not generated. If simulations were to be used for this purpose, all input sequences would have to be tested to prove that a given output is not generated. Mathematics solves this problem by providing one conclusive proof.

3.2 Properties and observers

Formal verification is performed with regard to a set of properties – requirements on the input, output and internal states. Properties may be derived from the requirements specifications, and must be formalized into some logic or other formal language before the verification tool can perform its task.

In the case of Scade, properties are expressed in the form of observers (see Figure 3.). Observers are nodes that are programmed to continuously monitor the input and output signals of the node to be verified. If the observer finds that the property is being violated, it emits an alarm signal. Proving the property has then been reduced to proving that the alarm signal can never be emitted, whatever input is fed to the model. Properties that can be expressed this way are called safety properties in the formal verification terminology. Intuitively, safety properties capture the designer intention that "nothing bad happens". Formally proving that "something good happens", referred to as liveness properties, is more difficult, and is covered by simulations in this project.

![Figure 3. An observer in Scade](image-url)
Notice that the observer is not meant to be run or simulated – it will only be a part of the static formal analysis.

### 3.3 Getting started with Prover Plug-In

Prover Plug-In is built into Scade, and thus you run it from Scade having your model loaded. This section describes what you need to do to make it work.

#### Creating a verification project

The best way to run a verification smoothly is to create a separate Design Verifier project. This project will contain all the observer and verification nodes, and the project to be verified (in this case, the ATM model) will be imported as a library.

1. In the File menu, select New... and create the Design Verifier project ATM_Verification.
2. In the Select Scade Project dialog screen, choose your ATM.etp model.

#### Adding verification nodes

With Prover Plug-In for Scade, verification is performed with regard to verification nodes. Whatever a node contains, Prover will try to find some input that breaks the assertions set on the boolean output signals, and if it fails finding such input, the property is determined true.

In practice, this means that a verification node contains the top node of the model to be verified (in our case, n_ATM) and the observer nodes. The input signals of the verification node are the same as the inputs of the top node of the model, and the output signals are a set of boolean alarm signals – one from each observer. The constraints to set on the output signals are that they must always be true. In essence, you should model the observer so that it emits false when receiving incorrect behaviour from the observed system.

Verification and observer nodes are added to the ATM_Verification project as usual in Scade. The nodes of the ATM will be available in the Framework Window if you have added the ATM model into you verification project correctly. Try to formulate a property that the ATM should satisfy and see if you can verify it using the Prover Plug-In.

#### Starting Prover Plug-In

Prover Plug-In is by default activated when opening Scade. You can see in the workspace window if the Design Verifier tab is visible. If not, you have to start the Design Verifier by adding a Prover Plug-In file to the verification project.

1. In the Project menu, select Enable Design Verifier...
2. Click OK; no changes in the dialog should be needed.

#### Making Proof Objectives

Assertions on the outputs of the verification nodes are set by adding them to the list of Proof Objectives.

1. In the Framework View, expand the interface of the verification node.
2. Right-click the output to add and select Insert->Proof Objective.

If Proof Objective is not in the pop-up menu, Prover has probably not been properly started – see "Starting Prover Plug-In" above. If it is there, but disabled, it might be that the output signal is not boolean – see "Adding verification nodes" above.
Running the verification

1. In the File menu, select Save All.
2. To run a specific task, right-click it and select Analyze.
3. To run all tasks in a folder, right-click the folder and select Analyze. To skip over a task that takes too long time (say, more than a few minutes), click Skip in the Progress dialog.

The results of the verification are shown in the Browse window at the bottom. If a property is falsifiable, you can double-click it in this window to view the input sequence that causes the violation.

If the verification fails (Raised an error in the Browse window) you can try the following:

1. Make sure that the project was saved before you ran the verification.
2. Make sure that both the ATM and the ATM_Verification projects are semantically correct. Go back to the Framework view, select the top node and select Semantic Check in the Tools menu.
3. Make sure that there are no complex types, such as enumeration types, tuples etc, in the model.
4. If all the above points are cleared, quit and restart Scade and try again.
References


