University of Konstanz
Department of Computer and Information Science

Bachelor Thesis for the degree
Bachelor of Science (B.Sc.) in Information Engineering

Evaluation of the Matlab Simulink Design Verifier versus the model checker SPIN

by
Florian Leitner
(Matr.-Nr. 01 / 612538)

1^{st} Referee: Prof. Dr. Stefan Leue
2^{nd} Referee: Prof. Dr. Ulrik Brandes
Konstanz, July 21, 2008

Abstract

An increasing number of industrial strength software design tools come along with verification tools that offer some property checking capabilities. On the other hand, there is a large number of general purpose model checking tools available. The question whether users of the industrial strength design tool preferably use the built-in verification tool or a general purpose model checking tool arises quite naturally. In this bachelor thesis, the Simulink Design Verifier and the SPIN model checking tool are compared. The comparison is based on the case study of an AUTOSAR compliant memory management module. The comparison is both functional in that it analyzes the suitability to verify a set of basic system properties, and quantitative in comparing the computational efficiency of both tools. In this context, it is also described how Simulink / Stateflow models can be manually translated into the input language of the model checker SPIN.

Kurzfassung

Eine steigende Anzahl von industriell eingesetzten Software Design Werkzeugen, bietet Möglichkeiten “Korrektheitseigenschaften“ zu verifizieren. Andererseits, gibt es eine große Anzahl an allgemeinen Model Checking Werkzeugen. Natürlich stellt sich nun die Frage, ob der Benutzer eines industriellen Software Design Werkzeuges nun das eingebaute Verifizierungs-Werkzeug benutzen sollte, oder ob er eines der allgemeinen Model Checking Werkzeuge bevorzugen sollte.

In dieser Bachelorarbeit wird der Simulink Design Verifier mit dem model checking Werkzeug SPIN verglichen. Dem Vergleich liegt die Fallstudie eines AUTOSAR kompatiblen Speicherverwaltungs-Modul zu Grunde. Der Vergleich ist sowohl funktional, da analysiert wird, inwieweit sich die Werkzeuge dazu eignen eine ausgewählte Menge an “Korrektheitseigenschaften“ zu verifizieren, als auch quantitativ da die Effizienz der beiden Werkzeuge verglichen wird. Im Zusammenhang wird auch beschrieben, wie Simulink / Stateflow Modelle manuell in die Eingabesprache des Model Checker SPIN übersetzt werden können.

Acknowledgments

Leider läßt sich
eine wahrhafte Dankbarkeit
mit Worten nicht ausdrücken.
- *Johann Wolfgang von Goethe* -

Firstly, I want to thank my supervisor Stefan Leue for his outstanding guidance and mentoring, but also for giving me the chance to present my work at FMICS 08.
I want to thank Ulrik Brandes for agreeing to be the second referee of this thesis.

I thank the whole software engineering group at the University of Konstanz for providing a great and friendly work climate.

I am indebted to my family for the support and encouragement during the last years.
My special gratitude goes to Viktoria for her love and understanding and for her continuous motivation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

It is increasingly common for industrial strength software design tools for embedded real-time systems to provide verification capabilities. This capabilities aid in checking software design models for important system properties. Examples include the SDL tool Telelogic TAU[1], the Statemate ModelChecker[2] and the Simulink Design Verifier[3] which is an optional component of the Matlab/Simulink tool set. These verification tools are tightly integrated with the design tool that they belong to. The design tools are typically domain specific. The Telelogic TAU tool, for instance, enjoys widespread popularity in telecommunications while Matlab/Simulink is a de-facto standard for software design in automotive system engineering [1].

On the other hand, there is a large number of model checking tools available that are not tightly integrated with some software design tool, but instead can be used as stand-alone tools and possess a tool-specific input language. Examples of such tools include SMV [2], NuSMV [3] and SPIN [4]. There is a tendency in the literature to view symbolic model checking tools such as SMV and NuSMV to be more suitably applicable in the hardware domain, since they seem to deal particularly well with regularly structured systems. Symbolic model checking also seems to more adequately deal with data oriented design models. Explicit state model checking on the other hand, as implemented in SPIN, is generally viewed to cope more easily with irregularly structured concurrent software systems [4].

Designers of safety-critical software applications are faced with the challenge to decide whether they should rely on the use of the built-in verification engines, or whether they should invest effort and resources into either a manual translation of their design models into the input language of a general purpose model checker, or even into building a translation tool that automatically translates their design into a general purpose model checking tool. It is the objective of this bachelor thesis to discuss the basis for this decision

[1] http://www.telelogic.com/products/tau/index.cfm
[2] http://modeling.telelogic.com/products/statemate/add-ons/testing-and-validation.cfm
[3] http://www.mathworks.com/products/sldesignverifier/

making in the setting of a software system from the automotive domain. For the purpose of this comparison, we consider the Simulink Design Verifier, since it is integrated into a domain-specific design tool, and the model checker SPIN, since it seems an obvious choice for the analysis of the type of concurrent software that we analyze.

For the assessment of the alternatives we use the design of a *Non Volatile Random Access Memory Manager* (NVRAM) component from the AUTOSAR[4] automotive open system architecture. AUTOSAR is being defined by many representatives of the automotive industry in order to facilitate the collaboration of automotive component suppliers and car manufacturers by specifying component interaction interfaces. The choice of the NVRAM component is justified by the fact that it is relatively generic in the architecture, since every electronic control unit (ECU) will need to provide its own NVRAM component.

The study that we present compares the two verification approaches both qualitatively and quantitatively. We propose a set of central requirements on the NVRAM component that we wish to be verified for our design. We then discuss whether and how they can be verified using either of the approaches. For those requirements that can be verified using both approaches we compare the performance of both. We also comment on the effort necessary to convert the Stateflow model to Promela, the input language of the SPIN tool. For the time being this translation is manual. We decided not to translate Stateflow directly into Promela, but to use the visual modeling tool VIP [5] as a target for the manual translation. VIP is a visual modeling interface that allows for concurrent system modeling using hierarchical, communicating state machines. VIP compiles efficient Promela code that encodes states using a state-label and goto mechanism. Notice that our analysis does not hinge upon the use of VIP, even though we perceive it as supporting the manual translation well.

1.2 Related Work

The verification engine underlying Simulink Design Verifier is based on SAT solving technology [6]. There is a substantial body of literature available that discusses formal semantics and analysis approaches to Matlab/Simulink and the Stateflow language. An operational semantics of Stateflow based on Plotkin rules [7] is defined in [1]. The authors observe that in spite of the relative complexity of the Stateflow language, the portion that is generally recommended to be used in practics has a surprisingly easy semantics. In [8] the formal semantics of a fragment of Stateflow is described based on a modular representation called communicating pushdown automata. Various papers describe translations of the Stateflow language into various model checking tools. A translation into the symbolic model checker SMV is presented in [9] while [10] discusses a translation into the NuSMV symbolic model checker, a successor to SMV. The paper [11] suggests an analysis of Simulink models using the SCADE design verifier in the particular setting of system safety analysis. Another translation from Stateflow to Lustre is described in [12]. Both share the interpretation of Simulink and Stateflow as a synchronous language, as mandated by the use of Lustre as the target language for the translations. Invariance checking

[4] http://www.autosar.org

of Simulink/Stateflow automotive system designs using an symbolic model checker is proposed in [13]. A first attempt to validate Stateflow designs using SPIN is presented in [14]. However, the translation tool used is not publicly available and it is therefore difficult to compare with this approach. In conclusion, we are not aware of any study that relates the Simulink Design Verifier tool to the SPIN model checker, which is the focus of this bachelor thesis.

1.3 Structure of the Thesis

In Chapter 2 an overview about Simulink, Stateflow and the Simulink Design Verifier on the one hand and VIP, SPIN and Promela on the other hand is given. The Translation from Simulink / Stateflow to VIP / Promela is described in Chapter 3. An introduction to the NVRAM case study is given in Chapter 4. The evaluation of the verification capabilities of Simulink Design Verifier and SPIN will be presented in Chapter 5. We conclude in Chapter 6.

Chapter 2

Simulink and SPIN

2.1 Matlab / Simulink / Stateflow

2.1.1 Overview

Matlab[1] is developed and distributed by The MathWorks[2] and is a Software for numerical computations. The name Matlab is derived from MATrix LABoratory. Simulink[3] extends Matlab with the capabilities to design and simulate dynamic and embedded systems in a graphical environment. Simulink models are block diagrams, that consist of blocks from the Simulink block library, which are connected with Simulink signals. Stateflow[4] extends Simulink with the possibility to develop state machines. All together Matlab, Simulink and Stateflow build an industrial strength software design tool for embedded systems, that is widely used in the avionics and automotive field.

2.1.2 Simulink

Simulink is an extension of Matlab, that allows for design and simulate embedded systems, which are modeled as block diagrams. In this section an overview of Simulink and its semantic is given based on [15]. Simulink is capable of model both continuous systems and discrete systems. One Simulink block can consist of one and more other Simulink blocks and therefore build a hierarchical block diagram. In Simulink there are two kind of blocks, nonvirtual blocks and virtual blocks. Nonvirtual blocks are used to represent elementary systems. Whereas virtual blocks have no effect in the definition of the systems of equations described by the block diagram and are just used to make the modeling more convenient. An example of such a virtual block is the bus creator block, which allows for grouping two or more signals, thus the user needs only to draw one instead of two (or more) signal lines from one block to another.

Simulink block diagrams define time-based relationships between signals and state variables. By evaluating those relationships over time, a solution for the block diagram can

[1] http://www.mathworks.com/products/matlab/
[2] http://www.mathworks.com
[3] http://www.mathworks.com/products/simulink/
[4] http://www.mathworks.com/products/stateflow/

be obtained. The start and stop time for this evaluation is specified by the user. Signal values are defined for all points in time between the simulation of the block diagram has been started and the time when the simulation is finished. The relationships between signals and state variables are defined by a set of equations represented by blocks. Each block consists of a set of equations, which are defining the relationship between the input signals, the output signals and the state variables. The state variables are representing the computational results of the previous time step.

2.1.3 Stateflow

Stateflow supports the creation of Stateflow diagrams, which are a variant of the finite state machine notation established by Harel [16]. In addition, it is also possible to draw stateless flow chart diagrams. A Stateflow chart can be integrated into Simulink and Data can be passed from Simulink to Stateflow charts and vice versa. To organize the Stateflow charts of complex systems, Stateflow allows for hierarchical and parallel Stateflow charts. But even though Stateflow allows to define parallel states, the execution order of this states is always statically known and sequential. Hierarchy means, that you have the possibility to define parent and child states. In the following a brief overview of the Stateflow Elements is given based on [17] Each state can be labeled with the following labels:

```
name
entry:  entry actions
during:  during actions
exit:  exit actions
```

Where `name` represents the name of a state, `entry actions` are all actions that are executed when the state is entered, `during actions` are all actions that are executed while in a state and `exit actions` are executed when a state is left. To describe these actions, Stateflow uses an own C like language that is defined in [17]. The transitions leading from one state to another can be labeled in the following way:

```
event[condition]{condition_action}transition_action
```

If the `event` occurs, the transition will be taken, if the `condition` is true. Specifying an event is optional if no event is specified, the transition can be taken if the `condition` is true. In case no `condition` is specified, this will have the same effect then specifying a condition that is always true. The `condition action` is executed as soon as the condition is evaluated to true. If no condition is specified the `condition action` will always be executed. After the transition destination has been determined to be valid the `transition action` will be executed.

The Semantics of Stateflow are described informally in the user guide [17] and a large portion is defined through its behavior while simulated. In [1] Hamon and Rushby define an operational semantics for Stateflow. Tiwari describes in [8] the formal semantics of a fragment of Stateflow based on a modular representation called communicating pushdown automata.

2.1.4 Simulink Design Verifier

Simulink Design Verifier (SDV) is an extension of the Mathworks Matlab / Simulink tool set. It performs exhaustive formal analysis of Simulink models to confirm the correctness of Simulink models with respect to given properties. The user can specify these properties directly in Simulink and Stateflow as assertions. The assertions are specified in the form of prove objectives. To prove these objectives, Simulink Design Verifier searches for all possible values for a Simulink or Stateflow function in order to find a simulation that satisfies an objective. Simulink Design Verifier does not expect a model to be closed. It will compute all possible input signals automatically. As written in [18] Simulink Design Verifier uses the Prover Plug-In [6] in order to prove properties. The Prover Plug-In is developed and maintained by Prover Technology [5]. The key idea of the Prover Plug-In is to model problems as propositional logic formulas and to apply proof methods to decide whether the formulas are tautologies, that is the the formula evaluates to true for all assignments, or not. This problem is solved by the Prover Plug-In using a reductio ad absurdum argument, where a combination of different analyses is used to check whether the negation of the formula is unsatisfiable.

As was also observed in [1], Simulink does not support an interpretation of concurrency in terms of a nondeterministic interleaving of concurrent events. Even if Stateflow charts could be executed concurrently, Simulink will interpret their execution in an predefined, deterministic order based on the visual layout of the chart. The Simulink Design Verifier does not provide any means to check a model for any type of concurrency related properties, such as race conditions. Likewise, deadlocks could only be detected if the one deterministic execution that Simulink permits happens to end in a deadlocked state.

[5] http://www.prover.com/

2.2 SPIN / Promela / VIP

2.2.1 SPIN and Promela

SPIN is an explicit state model checking tool. The Promela modeling language, which is interpreted by SPIN, allows for describing concurrent system models. The concurrent processes in Promela synchronize via asynchronous or synchronous message passing, and via shared variables. SPIN implements a simple Depth-First Search algorithm for safety properties, and a nested Depth-First Search for properties specified in Linear Temporal Logic. In order to deal with large state spaces SPIN offers bit state hashing as well as automated partial order reduction. SPIN has been applied in the analysis of many concurrent software systems. A very comprehensive overview of SPIN is given in [4].

2.2.2 VIP

VIP was designed as a visual input language for the SPIN model checker. VIP follows the ideas of the modeling language UML-RT [19] for the visual specification of system designs. Concurrent software components are modeled as *capsules*. The behavior of the system is specified using hierarchical state machines. Out of the diagrams VIP compiles Promela code, the input language of the SPIN model checker. In the compiled Promela model states are represented using Promela labels, and state transitions using `goto` statements. Transitions and states can be labeled with Promela code. Currently, VIP does not support model checking result interpretation at the level of the VIP model. However, it is easy to trace counterexamples in the generated Promela code back to the original VIP model. For details on VIP we refer the reader to [5].

Chapter 3

Translating the Model

3.1 Translation Workflow

In order to compare the capabilities of the Simulink Design Verifier and SPIN, the model has to be translated from Simulink / Stateflow into VIP / Promela. In [14] a Tool that performs such a translation is described, but hence this tool is not public available we had to come up with our own mapping. Our approach to map Stateflow elements to Promela differs in the way that in [14] Promela variables where used to model states whereas we are using Promela labels and `goto` statements to model states. The reason for this decision is that according to measurements documented in [20] the state space size for using Promela variables to store states is about twice the size as if Promela labels and `goto` statements are used. While the state vector size remains the same for both variants.

The translation is based on the semantics of Stateflow described in [1] and [8]. Since the Simulink / Stateflow model will be translated in C code in order to run on a embedded system, we allowed deviations of translation from the Simulink / Stateflow semantics in case of being more suitable to model the actual system.

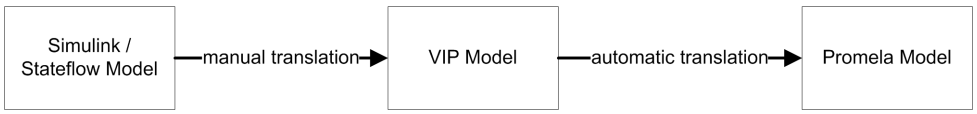


Figure 3.1: Translation Workflow.

Figure 3.1 shows the translation workflow, first the Simulink / Stateflow model is manually translated into a VIP model. After that the automatic translation from VIP to Promela is used to build a Promela model. VIP was selected for this first step of translation, because it provides a graphical user interface and thus a similar way of modeling systems like with Simulink / Stateflow. Hence, human errors that are potentially introduced via a direct translation into Promela code are avoided. We describe here the mapping between Simulink / Stateflow elements and VIP / Promela elements.

3.2 Translating Simulink Elements

3.2.1 Simulink Subsystems

Simulink Subsystems can be translated to VIP capsules or Promela proctypes.

3.2.2 Simulink Signals

Simulink Signals can be translated into global variables. While in Simulink an output port of one subsystem is connected via a signal to an input port of an other subsystem, in VIP / Promela we propose the usage of one global variable that can be accessed by both capsules / proctypes.

3.2.3 Inports / Outports

Due to the fact, that signals are translated into global variables and in- and out-ports represent writing or reading of signal values, in- and out-ports can be translated to variable assignments and the reading of variable values.

3.2.4 Logic and Math Operators

Logic and math operators can be modeled by performing the same logical or mathematical operation with Promela.

3.2.5 Data Stores

Data Stores can be represented in VIP / Promela using variables. The read / write data store element can be represented with reading the value of a variable and respectively assigning a new value to a variable.

3.2.6 Mux / Demux

Mux block combines signals into an vector, while a demux block takes an vector as input and has one output signal for each element of the vector. This behavior can be modeled in VIP / Promela using an array of variables.

3.2.7 Sources

Simulink provides a group of block elements called sources, which are creating output signal sequences with defined algorithms (e.g. Random Numbers, Sine Wave, Pulse Generator etc.). These blocks can be implemented in VIP / Promela, by implementing the underlying algorithm.

3.2.8 Sinks

Sinks (e.g Scopes, Displays etc.) are used in Simulink for simulation purposes and are not needed to be translated.

3.3 Translating Stateflow Elements

3.3.1 State Chart

A Stateflow chart, can be translated to a capsule in VIP and a proctype in Promela.

3.3.2 States

VIP allows for building hierarchical state machines, thus a Stateflow state can be directly translated into an VIP state. In Promela states are represented by Promela labels.

3.3.3 Transitions

In VIP transitions can be drawn from one state to another. In Promela transitions are modeled as `goto` statements that are pointing to the Promela labels.

3.3.4 Events

Basically events are triggers on falling or rising edges of a signal. Therefore we translated events as bit variables that will be set to one if they are available and to zero if not. One can also group events to an bit array (i.e. one or more bytes). These bit variables or bit arrays will then be checked in the guard of the VIP transitions. In Promela the variables will be checked using `if` statements.

3.3.5 Conditions

Conditions will be linked with an logic “and“ to the event condition mentioned above in the guard of the transition or in the `if` statement, respectively.

3.3.6 Actions

Transition / Condition Actions can be represented in VIP by using the action field of the transactions. In Promela actions can be written after the `if` statement and before the `goto` statement.

3.3.7 Default Transitions

Default Transitions can be translated to start-states in VIP programs. By using an `if` statement with `true` as condition and a `goto` statement at the very beginning of the proctype, that is representing the state machine, one can model a default transition in Promela.

3.3.8 Connective Junctions

Connective Junctions can be modeled in VIP by using a state with no entry, during or exit actions. In Promela, connective junctions can be modeled using labels. Other than

that, some additional steps are necessary to preserve the semantics of Stateflow, which are described in 3.4.3.

3.3.9 Input / Output Variables

Input and Output variables of Stateflow chart, are representing the values of Simulink signals and as earlier mentioned, Simulink signals can be translated to global variables both in VIP and in Promela.

3.3.10 Local Variables

Local variables can be translated to variables that are declared in the VIP capsule, respectively in the Promela proctype.

3.4 Preservation of the Semantics of Simulink and Stateflow

3.4.1 Concurrency

VIP creates a concurrent proctype for each capsule which results in a concurrent system model. However, as already mentioned in 2.1.4 Simulink and Stateflow models are executed in a sequential order. To model non-concurrent execution in VIP / Promela, a global variable lock is introduced, that controls the execution of the state machines and produces the same sequential execution order like in the Simulink / Stateflow order. The same concept applies to parallel Stateflow states, where the execution order is also sequential. Nevertheless, in some cases it may be desired to have the concurrent version available for verification, because this model may be more close to the real implementation.

3.4.2 Nondeterminism

Stateflow does not allow for having nondeterministic state machines. If two outgoing transitions of a state are valid a the same time (like in Figure 3.2, one of them will be selected in an deterministic way. This is done using the Stateflow *12 o'clock* rule [12]: starting at 12 o'clock position of a state and proceeding clockwise around the state, conditions are evaluated and the first transition that is enabled will be taken. If for example in Figure 3.2 p and q are true at the same time, always the transition labeled with $[p = true]$ will be taken. Since this approaches relies on the graphical positions of the transitions, it may

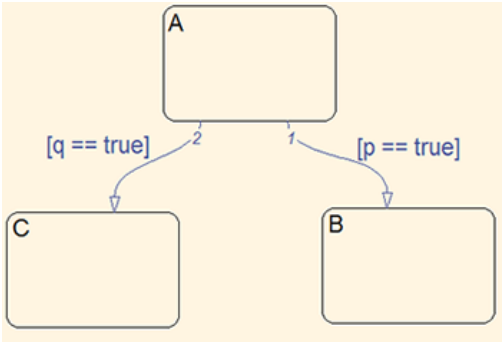
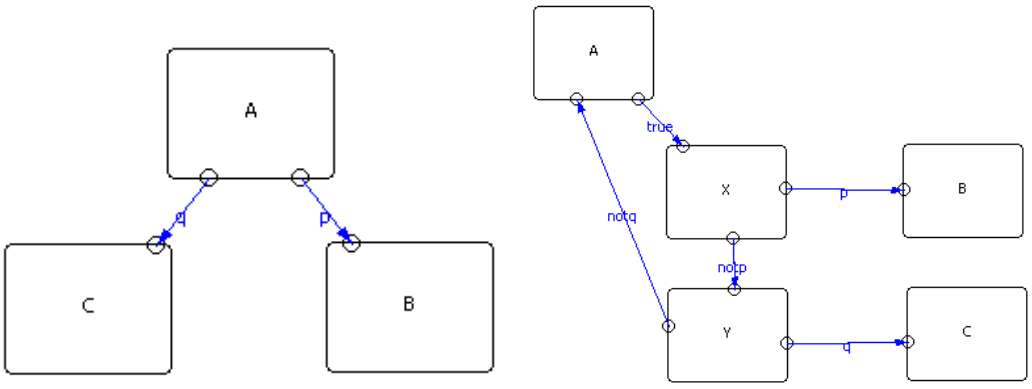


Figure 3.2: Stateflow graph with two enabled transitions.

lead to undesired behavior and small changes (removing a transition or reordering two transitions) can have a big impact on the systems behavior. Hence we propose to allow SPIN to nondeterministically choose one of the enabled transitions. The corresponding translation for this approach to VIP can be seen in Figure 3.3(a). Figure 3.3(b) shows how the deterministic semantic of the *12 o'clock rule* can be modeled with VIP.



(a) Non-deterministic version. (b) Deterministic version.

Figure 3.3: Translation of Figure 3.2 to VIP

3.4.3 Backtracking

We translate connective junctions to states. But to preserve the semantics of Stateflow, some additional steps are needed. Stateflow uses the 12 o'clock rule to search for enabled outgoing transitions (step 1). If an enabled transition is found, it will be taken (step 2). If now a connective junction is reached, the outgoing transitions of the connective junctions will be checked according to the 12 o'clock rule (step 3). If one of the transitions is enabled and leads to a state it will be taken and the search is completed (step 4). (Step 2 and 3 can occur repeatedly.) But if none of the outgoing transitions of a connective junction is enabled, Stateflow will backtrack to the previous connective junction, without undoing the actions made while executing the transitions. An example of this behavior is given in Figure 3.4, the gray arrows show the execution order. At first, the transition two the left connective junction is taken, then according to the 12 o'clock rule the transition labeled with 1 is taken. At the next connective junction, there is no enabled transitions and Stateflow is backtracking to the first connective junction without undoing the transition action ($x = x + 2$). Then the transition labeled with 2 is taken. When state C is entered, x is 4 and not as one would expect 2.

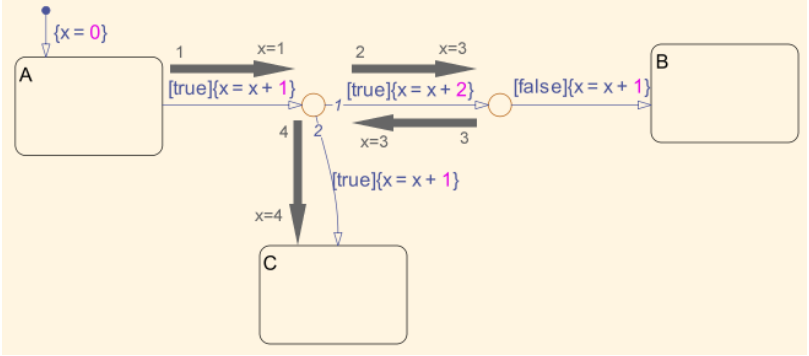


Figure 3.4: Backtracking in Stateflow

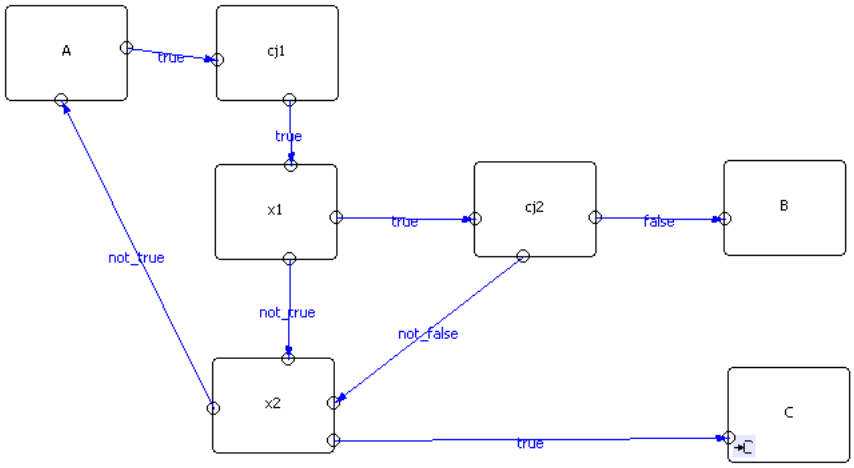


Figure 3.5: Backtracking in VIP

Figure 3.5 shows how the backtracking behavior can be achieved in VIP. The states starting with cj are representing the connective junctions and the states starting with x are needed to represent the 12 o'clock rule. The transition actions are specified in the action field of the transition properties.

Chapter 4

The NVRAM Case Study

4.1 AUTOSAR and the NVRAM Manager

The complexity of electronic and software components in cars has reached a very high level and is still increasing. Consequently, the automotive industry is seeking new techniques to ensure the quality and most important the safety of automotive products. The Automotive Open System Architecture (AUTOSAR) is a standardized software architecture whose main goal is to increase the hardware independence of single software modules and strives towards high reuse. AUTOSAR also promotes the usage of commercial and well tested off-the-shelf software and will also help to reduce the development costs of automotive systems. A further objective of AUTOSAR is to increase interoperability of automotive control unit software components in order to facilitate integrating components manufactured by different manufacturers. Figure 4.1 shows the different layers of the AUTOSAR software architecture.

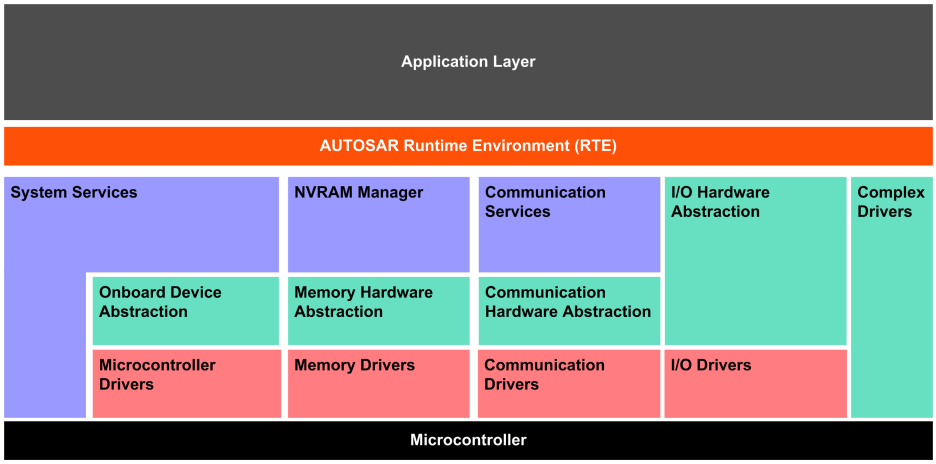


Figure 4.1: AUTOSAR Software Architecture

For this case study we selected the AUTOSAR Non Volatile Random Access Memory Manager (NVRAM Manager) component as specified in [21]. The NVRAM Manager provides the central memory management of AUTOSAR control units. The random access memory (RAM), the read only memory (ROM) and the non volatile memory (NVM) are managed by the NVRAM Manager. The application has only access to the RAM. Upon request from the application the NVRAM Manager copies data from ROM to RAM, RAM to NVM or vice versa. The NVRAM Manager provides services to ensure the data storage and maintenance of non volatile data according to the individual requirements in an automotive environment. An example for such a requirement is to handle concurrent access to the memory. The main reason why we choose the NVRAM Manager for this case study, was its independence from the application. Each automotive application has to control memory, that is why the NVRAM Manager will be found in almost every AUTOSAR control unit regardless whether this is an airbag control unit or an engine control unit.

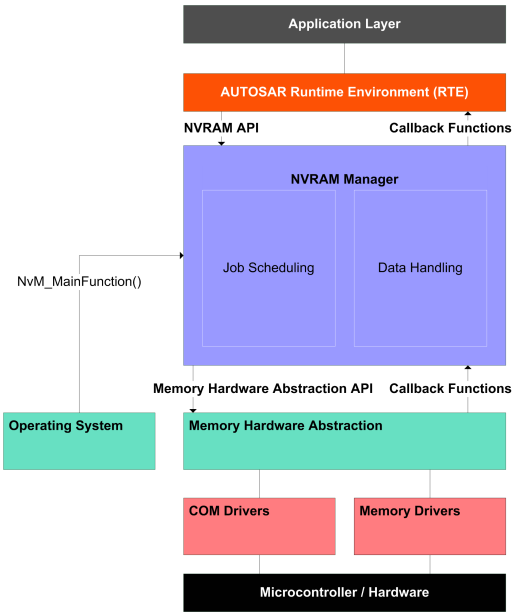


Figure 4.2: The NVRAM Manager and its environment.

The block diagram (Figure 4.2) shows the interface of the NVRAM Manager with other layers and modules. The interface is provided through the API and callback functions. Neither the NVRAM Manager nor the layers above the NVRAM Manager have direct access to the non volatile memory. All access from application to the memory is routed through the AUTOSAR Runtime Environment (RTE) to the NVRAM Manager. The NVRAM Manager calls an API function of the memory hardware abstraction, where the call is forwarder either to the COM driver (external memory) or to the memory drivers (internal memory). The job processing is controlled by a cyclic operating system task, which can be concurrent to the API calls of the application.

4.2 The NVRAM Simulink Model

The NVRAM Manager consists of two parts, one for the job scheduling and one for the data handling. For the purpose of this case study, these two parts are both modeled as Simulink Subsystems. The central function of the NVRAM Manger is the job scheduling, the data management part is just needed to store information about the configuration of the several memory blocks. Therefore we decided to focus this case study on the job scheduling part.

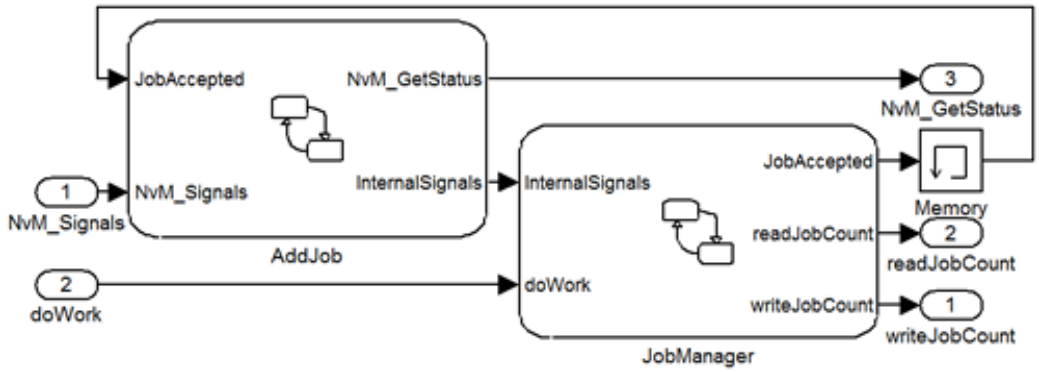


Figure 4.3: The Stateflow charts AddJob and JobManager.

The job scheduler is modeled using two Stateflow diagrams (Figure 4.3). One of these diagrams is responsible for adding new jobs to the queue (AddJob), and the other one represents the scheduler functionality (JobManager). These two Stateflow diagrams are communicating via Simulink signals. Simulink signals can be compared to the concept of using global variables for message passing. The AddJob Stateflow diagram is responsible for translating the incoming signal calls to a job type and to forward the request to the JobManager. There are five request types, namely write, erase, invalidate, restore and read a block. The requests write, erase, invalidate and restore are translated into write jobs and the read request is translated into a read job. The JobManager has two queues, one for write jobs and one for read jobs. This construction models the fact that write jobs have a higher priority than read jobs.

The signal NvM_Signals represents the calls from the application layer to NvM_Read, NvM_Write, NvM_Invalidate, NvM_Restore and NvM_Erase and is used to add new jobs to the queue. The signal doWork starts the job processing and is called by the NVRAM function NvM_MainFunction. The NvM_MainFunction is triggered by a cyclic operating system task.

We are basing the comparison of Simulink Design Verifier and SPIN on the Matlab/Simulink design model for the NVRAM manager that we sketched above. Some of the coding features in this model are adequate at the design level, but would impede state space exploration since they lead to very large state spaces. We therefore performed the following manual abstractions in order to obtain a verification model:

- Normally, the queues are modeled as multidimensional arrays and each job contains the memory address of the source location and the memory address of the target

location. For verification purpose it is sufficient if this array is a one dimensional array containing just the type of the job.

- Since the size of the queues has no impact on the functionality, we limited the size of the arrays to two elements. If the queue is full and a request is pending, we will accept this job without adding it to the queue. This abstraction is not property preserving with respect to deadlocks. In our experimental evaluation, we will include a model variant in which we block on a full queue in order to show that we can detect the resulting deadlock.
- The job scheduler normally calls the function of the data handling part to perform the requested operations, but since this does not affect the behavior of the job scheduler we omitted these calls from the model.

Figure 4.4 shows the Stateflow graph of the JobManager. It consists of 5 states. The JobManager waits in the idle state for a new command. If the AddJob Statechart sets the signal InternalSignal to two or four either a new write (state WriteQueueAdd) or a new read job (state ReadQueueAdd) will be added. If the doWork signal is set to one by the environment it will start to process the write jobs that are in the write queue, once this is finished it will process all read jobs and then return to the idle state.

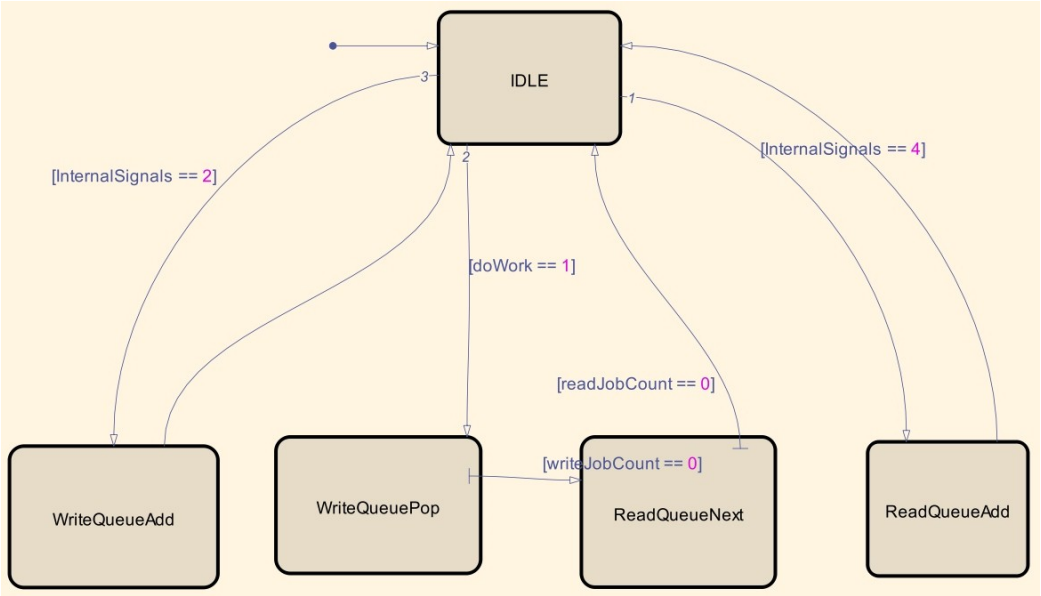


Figure 4.4: Stateflow graph of the JobManager.

4.3 The VIP / SPIN Model

As we argued above, the Simulink model of the NVRAM manager was manually translated in a visual specification using the VIP tool. The behavior of the system is specified using hierarchical state machines. Out of the diagrams Inter-process communication in the NVRAM case is modeled by means of access to global variables, which we use to mimic the Simulink signal-based synchronization of processes.

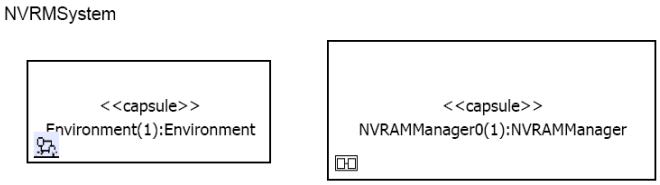


Figure 4.5: VIP graph of the NVRAM System Structure.

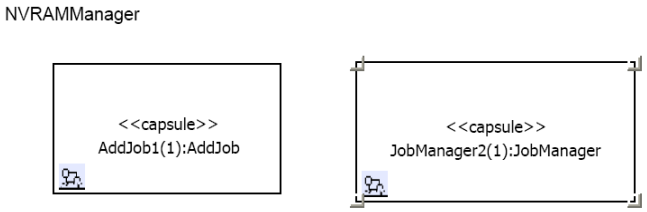


Figure 4.6: VIP graph of the NVRAM Manager.

The top level system structure of the NVRAM component consists of two capsules, the `Environment`, which contains the environment behavior, and the `NVRAMManager` itself, as depicted in Figure 4.5. The `NVRAMManager` is decomposed into two capsules named `AddJob` and `JobManager`, c.f. Figure 4.6. Inside these capsules we used the same state machine machine structure that is also used in the Stateflow charts of the Simulink NVRAM model. Figure 4.7 gives the top level state machine behavior of the JobManager that can easily be compared to the structure of the Stateflow JobManager state machine in Figure 4.4. Actions are attached to states and transitions, the Figure shows the guard attached to the transition labeled `AddWriteJob`. State machines in VIP are hierarchical, and Figure 4.8 gives the state decomposition of the `WriteQueueAdd` state in Figure 4.7. Notice the similarity to the corresponding state decomposition in the Stateflow model as given in Figure 4.9.

The syntax of the Simulink / Stateflow language is very similar to C and thus also easy to translate to Promela. Since VIP does not support definition of signals between two state charts, we used the concept of global variables for message passing. But as explained earlier, this represents a good mapping of the Simulink signal concept. In contrast to the Simulink Design Verifier, SPIN can only verify closed models. Closed means that the environment behavior is also a part of the model. Therefore we added a third capsule Environment that is non-deterministically sending either write or read requests or is starting the job processing. The behavior of the Environment capsule is given in Figure 4.10.

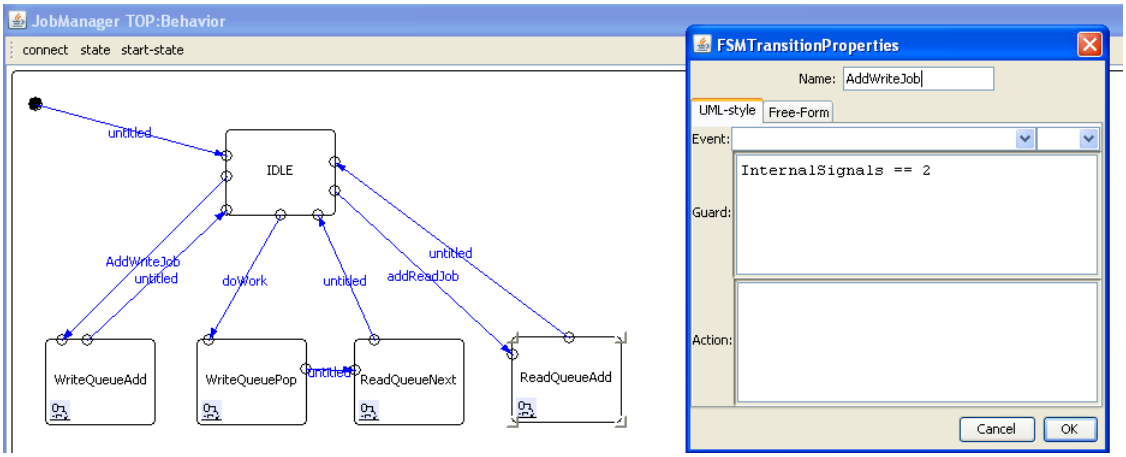


Figure 4.7: VIP graph of the JobManager behavior.

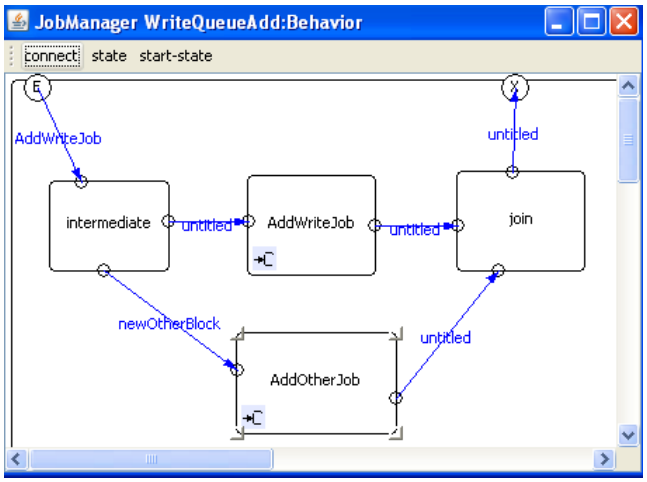


Figure 4.8: VIP graph of the WriteQueueAdd substate.

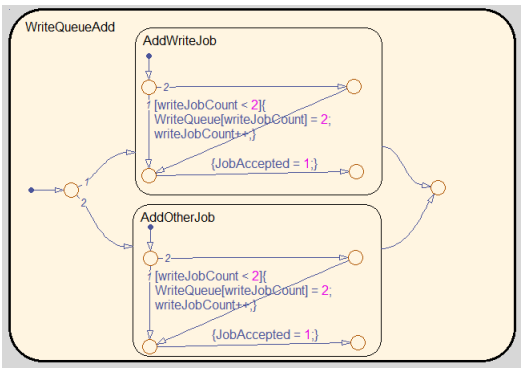


Figure 4.9: Stateflow graph of the WriteQueueAdd substate.

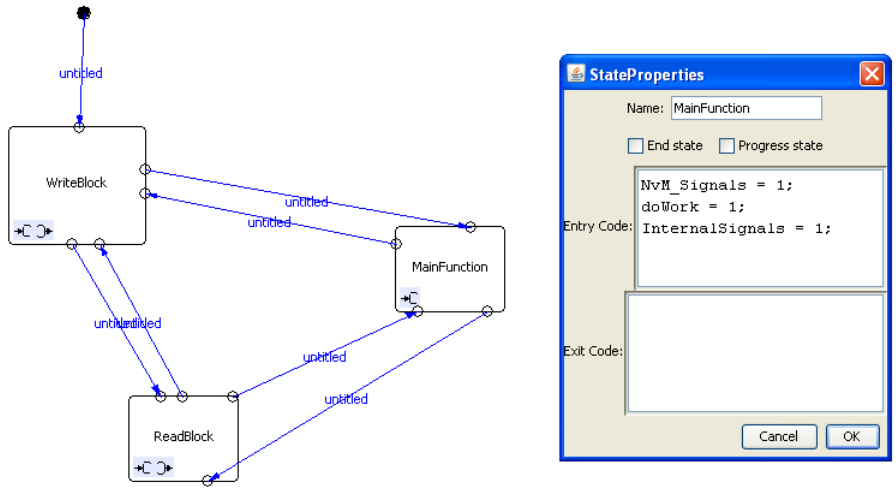


Figure 4.10: VIP graph of the Environment behavior.

VIP creates a concurrent proctype for each capsule which results in a concurrent system model. However, the two state machines in th Simulink model are executed in a sequential order. The concurrent execution of these capsules corresponds to the reality of the implementation, in which the corresponding tasks will be executed concurrently. Simulink does not allow for the modeling of concurrent behavior. To model the non-concurrent execution in VIP/Promela, we introduced a global variable lock that controls the execution of the state machines. The value of the lock variable indicates whether it is the turn of the AddJob, the JobManager or the Environment.

4.4 Verification Properties

In our case study we focus on 6 central properties required to hold of the NVRAM manager. These properties are derived from the following requirements on the NVRAM manager. We give these requirements using the numbering used in the corresponding AUTOSAR specification [21]:

- BSW013 - Handling of concurrent accesses to NVRAM: The NVRAM manager shall provide a mechanism to handle multiple, concurrent read / write orders, e.g. queuing.
- BSW08541 - Guaranteed processing of accepted write request: The NVRAM manager shall guarantee that an accepted write request will be processed.
- BSW08542- Job order prioritization: The NVRAM manager shall provide a prioritization for job processing order. The highest priority job shall be processed first; jobs with the same priority level shall be executed in FIFO order.

[21] does not specify how many different priorities there are. We assume that the NVRAM manager will distinguish between write and read requests. As we will show in 5.1.4 the model can be easily extended to support more than two different priorities.
To capture the above mentioned requirements, we are using the following properties:

- P1: All write jobs will be processed, before any read request is processed.
  This property represents BSW08541 and assures, that the job order prioritization is met.

- P2: If a request is pending, it will eventually be added to the queue.
  The requirement BSW08541 is represented by this property. This property assures, that pending request will be added to the queue, that is, no job will be ignored.

- P3: If a request was accepted, it has been added to the queue.
  This property assures proper behavior of the queues and hence represents requirement BSW013.

- P4: One single request will only be added once to the queue.
  Like P3 this property assures proper behavior of the queues and therefore represents requirement BSW013.

- P5: If a request was added to the queue, it eventually will be processed.
  This property represents both requirement BSW013 and requirement BSW08541. Because it assures that all requests that have been added to the queue will be processed at some point in the future (BSW08541) and it assures proper behavior of the queues (BSW013).

- P6: If a request was processed, it will be removed from the queue.
  This property assures proper behavior of the queues and hence represents requirement BSW013.

Notice that some of these properties establish temporal contexts (such as P5) and hence require some form of temporal logic capture and corresponding model checking, while others are merely assertional.

Chapter 5

Simulink Design Verifier vs. SPIN

5.1 Property Checking and Quantitative Comparison

In order to perform a quantitative comparison of the performance of Simulink Design Verifier and SPIN we measured the runtime of the verifications runs. The runtime here means the time that is needed to analyze the model. One would prefer to have other measurements than runtime to make a quantitative comparison, but unfortunately Simulink Design Verifier does not make any quantitative characteristics of a verification run available. To reduce the impact of other processes running on the same machine that may slow down the verification, we performed each run 10 times and produce here the average runtime (see Appendix). All tests have been executed on the same machine with 2GB of main memory and a 2 GHz dual core CPU.

5.1.1 Assertion Checking

We have introduced the properties to be verified in Section 4.4. Properties P1, P3 and P4 can be modeled as assertions in the following way:

- In order to prove property P1 we added the prove objective "writeJobCount == 0", written in Stateflow as `dv.prove(writeJobCount,'0');`, to the the state where processing of read jobs is started. The expected result for the verification is that the assertion will be satisfied. In the Promela model we added the assertion `assert(writeJobCount == 0);` at the corresponding position to the Promela model.
- Property P3 requires that if a request was accepted, it has been added to the queue. We abstract the queue to a queue with a queue size of 2. If the queue is full and a new job arrives, it will be accepted but not added to the queue. We represent this property using the following formula:

$$true == (JobAccepted == ((writeJobCount == 2) \vee (WriteJobCount_{t-1} == WriteJobCount - 1)) \vee ((readJobCount == 2) \vee (ReadJobCount_{t-1} == ReadJobCount - 1)))$$

This formula also covers property P4 which states that a single request will only be added once to the queue. Since it is not possible to insert logical formulas into prove objectives we modeled the formula using Simulink elements (c.f. Figure 5.1). The outcome of the formula then goes into the AddJob Stateflow chart, where it is checked in the `ErrorStatusReqPending` state with the command `dv.prove(inJobAcceptedState,'1');`.

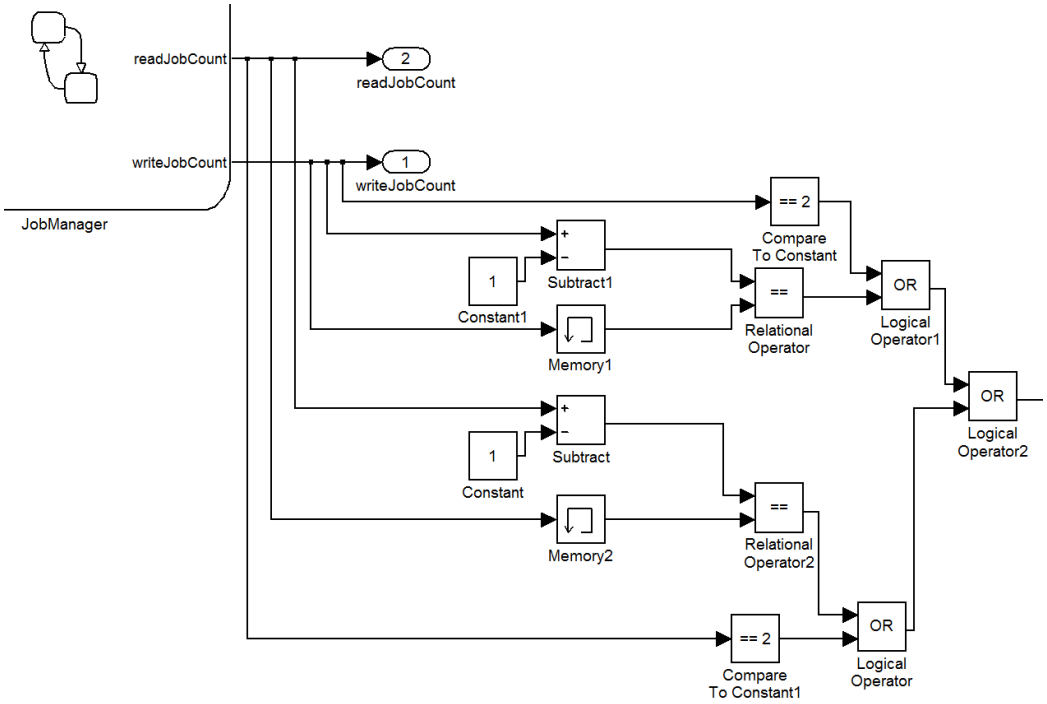


Figure 5.1: Logical formula of assertion 2.

In SPIN we modeled this property using the following assertion:

```
assert(((writeJobCount == 2 || ((writeJobCount-1)==writeJobCountMem))
  || (readJobCount == 2 || ((readJobCount-1) == readJobCountMem))));
```

The assertion is placed in the AddJob state chart in the `ErrorStatusReqPending` state.

Figure 5.2 shows the average runtime in seconds for the assertion checking. Both for SPIN and for Simulink Design Verifier we computed the average out of 10 repeated verification runs in order to factor out system dependent runtime delays. In addition to the sequentialized Promela model we also performed the same verification runs on the original concurrent version of the Promela model.

Property	Tool and Model	Runtime (sec.)	Property satisfied
P1.	Simulink Design Verifier	2.308	yes
	SPIN sequential version	0.0013	yes
	SPIN concurrent version	0.0166	yes
P3/4.	Simulink Design Verifier	3.19	yes
	SPIN sequential version	0.0019	yes
	SPIN concurrent version	0.0198	no
P1 & P3/4.	Simulink Design Verifier	3.623	yes
	SPIN sequential version	0.0022	yes
	SPIN concurrent version	0.0385	P1. yes P3/4. no

Figure 5.2: Runtime comparison for assertion checking

5.1.2 Deadlock Checking

A deadlock occurs when concurrent processes are waiting for each other or a shared resource in a cyclic fashion. As indicated earlier, we have one model variant in which writing a new job to a full queue will result in a blocking of the system. SPIN detects the resulting deadlock and produces an error trail leading into the deadlock state.

Simulink Design Verifier does not provide for deadlock detection. If one forces the Simulink simulation environment into the corresponding deadlock state it will be stuck in that state and wait for input.

5.1.3 Temporal Property Checking

The remaining properties P2, P5 and P6 can not be modeled as assertions since they assert properties of temporal contexts. The verification of temporal properties that cannot be encoded in assertions is beyond the capabilities of Simulink Design Verifier. We are hence restricted to SPIN in order to verify these properties. The properties are modeled using linear temporal logic (LTL) [22] in the following way:

- Property P2 states that if a request is pending, it will eventually be added to the queue. We capture this with the following LTL formula:

  $$\Box(reqPending \rightarrow (\Diamond reqAccepted))$$

  The proposition *reqPending* holds if the value of the variable `NvM_Signals` is set to 2 or 4, which is the case if a write or read request is pending. *reqAccepted* holds if the value of the variable `JobAccepted` is true and this is the case if a job was added to the queue. Translated into natural language the formula expresses that it is always the case that if *reqPending* holds, it implies that eventually *reqAccepted* will hold. As expected the property was proven satisfied by our model using SPIN.

- Property P5 states that if a request was added to the queue, it eventually will be processed. We capture P5 using the following formula:

$$\Box(reqAccepted \rightarrow (\Diamond reqProcess))$$

reqAccepted is the same proposition as in the previous LTL formula. *reqProcess* holds if the job scheduler is in a state in which either a read or a write request is being processed. To represent this proposition in the Promela model we introduced an auxiliary variable called `processing`, which is set to 1 upon entry into a processing state, and to 0 upon exit. The result of the SPIN verification was that the LTL formula is not satisfied by the model. An analysis of the SPIN counterexample revealed that the cause for this violation is not located inside the model, but in environment behavior that potentially violates fairness. As already explained, the environment behavior nondeterministically sends a read request, a write requests or starts the job processing. Because of this nondeterminism it is possible, that the job processing is never started and therefore the proposition *reqProcess* will never hold. For a second run, we changed the environment model to the deterministic sequence send write request, start job processing, send read request and start from the beginning. Now the property was satisfied.

- The LTL formula

$$\Box(reqProccess \rightarrow (\Diamond decrement))$$

was used to capture property P6 which states that if a request was processed, it will be removed from the queue. The variable `decremented` is set to 1 if the job was removed from the queue, and it is set to 0 when the state is left. The proposition *decrement* holds, if decremented is 1. *reqProcess* is the same like in the previous LTL formula. This property is also satisfied by our model.

Figure 5.3 shows the average runtime in seconds for the LTL property verification based on the sequentialized and the concurrent Promela model.

Property	Tool and Model	Runtime (sec.)	Property satisfied
P2.	SPIN sequential version	0.004	yes
	SPIN concurrent version	0.015	no
P5.	SPIN sequential version	0.015	yes
	SPIN concurrent version	0.015	no
P6.	SPIN sequential version	0.015	yes
	SPIN concurrent version	0.015	yes

Figure 5.3: Runtime comparison for LTL verification.

The property violations of P2 and P5 are caused by an unexpected interleaving of environment and system actions. The common cause is that the environment starts a new command before the processing of the previous command has been completed.

5.1.4 Scaling of the Model

So far, we only compared Simulink Design Verifier and SPIN with one relatively small model. To see how the runtime relates to the size of the model, we added a third and a fourth queue to the model. This change would be necessary if one wants to model more than two priorities. We verified Property 1. and Property 3/4. with 2 queues, with 3 queues and with 4 queues. As can be seen in Figure 5.4 the Verification with Simulink Design Verifier takes noticeable longer while in SPIN the runtime for the sequential model remains the same and the runtime for the concurrent model doubles from 3 queues to 4 queues but is still smaller then 0.1 seconds.

Number of queues	Tool and Model	Runtime (sec.)	Property satisfied
2 queues	Simulink Design Verifier	3.623	yes
	SPIN sequential version	0.0022	yes
	SPIN concurrent version	0.0385	P1. yes P3/4. no
3 queues	Simulink Design Verifier	19.065	yes
	SPIN sequential version	0.002	yes
	SPIN concurrent version	0.0201	P1. yes P3/4. no
4 queues	Simulink Design Verifier	26.789	yes
	SPIN sequential version	0.0016	yes
	SPIN concurrent version	0.0836	P1. yes P3/4. no

Figure 5.4: Runtime comparison for different number of queues

5.2 Qualitative Comparison

The following appear to be central aspect to the qualitative comparison of verifying system properties using SPIN and Simulink Design Verifier:

- We observed a significant discrepancy between the concurrent software architecture that we were modeling and the deterministic execution semantics that Simulink imposes. In applications like the NVRAM manager the capabilities of SPIN to deal with concurrency and to reveal concurrency related bugs appear to be far superior to the capabilities of Simulink Design Verifier.

- Simulink Design Verifier and SPIN allows for the checking of assertions. However, Simulink Design Verifier does not allow for expressing properties that go beyond assertion checking and which require a formalization using temporal logic formulae or equivalent mechanisms. Notice that according to [23] temporal properties following the response pattern, as we use here to capture P2, P5 and P6, belong to the most frequently used properties in practice.

- Simulink Design Verifier automatically closes the system against its environment, while in SPIN it is necessary to incorporate the environment into the model, which

requires extra modeling effort, but hives the user the chance to control the environment behavior.

- The translation from the limited subset of Simulink constructs used in the NVRAM example to VIP was fairly straightforward.
- Both Simulink Design Verifier and SPIN provide counterexamples, if a property is violated.
- The author of this thesis was very familiar with AUTOSAR and the NVRAM component, had a moderate amount of using Matlab/Simulink and SPIN, but was new to LTL model checking and the use of the Simulink Design Verifier. He estimated the effort for the different modeling and verification tasks as follows (given in Person Days (PD)):
  - Editing of the original Simulink NVRAM model: 10 PDs.
  - Editing of the abstracted Simulink NVRAM model used in the verification: 2 PDs.
  - Familiarization with VIP and editing of the VIP models: 2 PDs.
  - Property formalization and performing Simulink Design Verifier verification runs for properties P1, P3 and P5: 2 PDs.
  - Property formalization and performing SPIN verification for all 6 properties: 3 PDs.

  It can be concluded that the use of SPIN certainly requires additional effort, which on the other hand gives access to verification capabilities that Simulink Design Verifier does not possess.

An overview of the qualitative comparison is also given in Figure 5.5.

	Simulink Design Verifier	VIP / SPIN
Model and verify concurrent models	No	Yes
Support for nondeterministic models	No	Yes
Graphical modeling	Yes	Yes
Detection of deadlocks	No	Yes
Support for verifying temporal properties	No	Yes
Counterexamples	Yes	No
Automatically closing of the System	Yes	No

Figure 5.5: Feature comparison Simulink Design Verifier vs. VIP / SPIN

5.3 Threats to Validity

This case study cannot claim any form of empirical validity, since just one problem instance has been investigated. We also limited our considerations to the relatively simple Simulink language constructs that were used in our case study. While other researchers also consider a limited scope of the Simulink language, our conclusions regarding the good correspondence between the Stateflow and the VIP/Promela model may be invalid in case a different fragment of Simulink / Stateflow is considered. However, we are confident that a large portion of Simulink and Stateflow can easily be interpreted, not at least due to the results in [14]. Although our results in 5.1.4 indicate that SPIN will remain superior in terms of verification runtime for assertional checking, if the size of the model grows, we have not investigated how this advantage develops as the model becomes semantically more complex.

Chapter 6

Conclusion

6.1 Conclusion

We have presented a case study in which we compared the use of the Simulink Design Verifier and the SPIN model checker in the verification of important properties of the AUTOSAR NVRAM component. The quantitative comparison of both approaches is hampered by the unavailability of statistics regarding the verification run of Simulink Design Verifier. In terms of runtime SPIN appears to be superior for the problem instances that we considered. From a qualitative perspective, SPIN appears to be more suitable in supporting the concurrent nature of the NVRAM software than Simulink Design Verifier. SPIN is is also able to verify both assertional and temporal properties, while Simulink Design Verifier can only check assertions. Unlike SPIN, Simulink Design Verifier does not require a system model to be closed, which saves modeling effort compared to SPIN. In conclusion we can say, that the effort for translating the NVRAM Manager to VIP has paid off. Not only because SPIN was superior in terms of runtime, but but also because it was the only way to verify all properties we had selected for verification.

Our case study addressed a component that can be classified as belonging to general system infrastructure that the AUTOSAR architecture defines. In further research we will investigate whether for other such components an equally straightforward and well scaling translation from a limited fragment of the Simulink language into VIP and/or SPIN is possible. We will then consider an integrated tool environment, in which models and verification results can be exchanged amongst the components Simulink, VIP and SPIN.

Chapter 7

Appendix

7.1 CD

All models used for this bachelor thesis are stored on the attached CD. In addition the CD comprises a digital copy of this bachelor thesis.

7.2 Measurements

	Simulink Design Verifier			SPIN sequential version			SPIN concurrent version		
	Runtime		Result	Runtime		Result	Runtime		Result
	AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX
Property 1:	**2,308**	**1,36**	**4,16**	**0,0013**	**0,001**	**0,002**	**0,0166**	**0,015**	**0,031**
Run 1	1,58		satisfied	0,001		satisfied	0,015		satisfied
Run 2	1,53		satisfied	0,002		satisfied	0,015		satisfied
Run 3	3,34		satisfied	0,001		satisfied	0,015		satisfied
Run 4	1,4		satisfied	0,001		satisfied	0,015		satisfied
Run 5	3,37		satisfied	0,001		satisfied	0,015		satisfied
Run 6	4,16		satisfied	0,002		satisfied	0,015		satisfied
Run 7	1,45		satisfied	0,001		satisfied	0,031		satisfied
Run 8	3,27		satisfied	0,001		satisfied	0,015		satisfied
Run 9	1,36		satisfied	0,001		satisfied	0,015		satisfied
Run 10	1,62		satisfied	0,002		satisfied	0,015		satisfied

Figure 7.1: Measurements for property 1.

	Simulink Design Verifier			SPIN sequential version			SPIN concurrent version		
	Runtime		Result	Runtime		Result	Runtime		Result
	AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX
Property 3+4:	**3,19**	**3,1**	**3,26**	**0,0019**	**0,001**	**0,005**	**0,0167**	**0,015**	**0,031**
Run 1	3,21		satisfied	0,001		satisfied	0,031		violated
Run 2	3,12		satisfied	0,004		satisfied	0,015		violated
Run 3	3,14		satisfied	0,002		satisfied	0,031		violated
Run 4	3,25		satisfied	0,001		satisfied	0,015		violated
Run 5	3,12		violated	0,001		satisfied	0,015		violated
Run 6	3,1		satisfied	0,001		satisfied	0,015		violated
Run 7	3,24		satisfied	0,005		satisfied	0,015		violated
Run 8	3,26		satisfied	0,001		satisfied	0,015		violated
Run 9	3,2		satisfied	0,001		satisfied	0,015		violated
Run 10	3,26		satisfied	0,002		satisfied	0.00		violated

Figure 7.2: Measurements for property 3/4.

	Simulink Design Verifier			SPIN sequential version			SPIN concurrent version		
	Runtime		Result	Runtime		Result	Runtime		Result
	AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX
Property 1 & Property 3/4	**3,623**	**3,57**	**3,73**	**0,0022**	**0,001**	**0,005**	**0,0385**	**0,015**	**0,078**
Run 1	3,62		satisfied	0,004		satisfied	0,046		p3/4 violated
Run 2	3,63		satisfied	0,002		satisfied	0,062		p3/4 violated
Run 3	3,57		satisfied	0,001		satisfied	0,078		p3/4 violated
Run 4	3,57		satisfied	0,001		satisfied	0,062		p3/4 violated
Run 5	3,57		satisfied	0,002		satisfied	0,031		p3/4 violated
Run 6	3,62		satisfied	0,005		satisfied	0,015		p3/4 violated
Run 7	3,65		satisfied	0,002		satisfied	0,015		p3/4 violated
Run 8	3,66		satisfied	0,001		satisfied	0,015		p3/4 violated
Run 9	3,73		satisfied	0,003		satisfied	0,046		p3/4 violated
Run 10	3,61		satisfied	0,001		satisfied	0,015		p3/4 violated

Figure 7.3: Measurements for property 1 & property 3/4.

	Simulink Design Verifier			SPIN sequential version			SPIN concurrent version		
	Runtime		Result	Runtime		Result	Runtime		Result
Scaled model: 3 Queues	**AVG**	**MIN**	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**	**MIN**	**MAX**
Property 1 & Property 3/4	**19,065**	**16,05**	**20,69**	**0,002**	**0,001**	**0,004**	**0,0201**	**0,013**	**0,029**
Run 1	19,96		satisfied	0,001		satisfied	0,024		p3/4 violated
Run 2	19,98		satisfied	0,002		satisfied	0,025		p3/4 violated
Run 3	20,12		satisfied	0,001		satisfied	0,013		p3/4 violated
Run 4	16,55		satisfied	0,001		satisfied	0,015		p3/4 violated
Run 5	19,72		satisfied	0,004		satisfied	0,024		p3/4 violated
Run 6	20,39		satisfied	0,003		satisfied	0,029		p3/4 violated
Run 7	20,69		satisfied	0,002		satisfied	0,02		p3/4 violated
Run 8	17,37		satisfied	0,002		satisfied	0,014		p3/4 violated
Run 9	19,82		satisfied	0,002		satisfied	0,016		p3/4 violated
Run 10	16,05		satisfied	0,002		satisfied	0,021		p3/4 violated

Figure 7.4: Measurements for property 1 & property 3/4 with 3 queues.

	Simulink Design Verifier			SPIN sequential version			SPIN concurrent version		
	Runtime		Result	Runtime		Result	Runtime		Result
Scaled model: 4 Queues	**AVG**	**MIN**	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**	**MIN**	**MAX**
Property 1 & Property 3/4	**26,789**	**26,03**	**27,98**	**0,0016**	**0,001**	**0,003**	**0,0836**	**0,067**	**0,113**
Run 1	26,7		satisfied	0,001		satisfied	0,067		p3/4 violated
Run 2	26,8		satisfied	0,002		satisfied	0,072		p3/4 violated
Run 3	26,72		satisfied	0,001		satisfied	0,072		p3/4 violated
Run 4	27,98		satisfied	0,002		satisfied	0,076		p3/4 violated
Run 5	27,97		satisfied	0,001		satisfied	0,084		p3/4 violated
Run 6	26,03		satisfied	0,001		satisfied	0,091		p3/4 violated
Run 7	26,09		satisfied	0,003		satisfied	0,113		p3/4 violated
Run 8	26,13		satisfied	0,001		satisfied	0,087		p3/4 violated
Run 9	26,48		satisfied	0,001		satisfied	0,096		p3/4 violated
Run 10	26,99		satisfied	0,003		satisfied	0,078		p3/4 violated

Figure 7.5: Measurements for property 1 & property 3/4 with 4 queues.

Bibliography

[1] Hamon, G., Rushby, J.: An operational semantics for stateflow. Int J Softw Tools Technol Transfer **9** (2007) 447–456

[2] McMillan, K.: Symbolic Model Checking. Kluwer Academic (1993)

[3] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Proc. CAV 2002. (2002)

[4] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addision–Wesley (2003)

[5] Kamel, M., Leue, S.: VIP: A visual editor and compiler for v-promela. In: Proc. TACAS 2000. Number 1785 in Lecture Notes in Computer Science, Springer Verlag (2000) 471–486

[6] Andersson, G., Bjesse, P., Cook, B., Hanna, Z.: A proof engine approach to solving combinational design automation problems. In: Proc. 39th Design Automation Conference (DAC'02), IEEE Computer Society Press (2002) 725

[7] Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University (1981)

[8] Tiwari, A.: Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International (2002) `http://www.csl.sri.com/~tiwari/stateflow.html`.

[9] Banphawatthanarak, C., Krogh, B., Butts, K.: Symbolic verification of executable control specifications. In: Proc. 1999 IEEE Int. Symposium on Computer Aided Control System Design, IEEE (1999)

[10] B., M., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Proc. ICFEM 2006. Number 4260 in Lecture Notes in Computer Science, Springer Verlag (2006) 606–620

[11] Joshi, A., Heimdahl, M.: Model-based safety analysis of simulink models using scade design verifier. In: Proc. SAFECOMP 2005. Number 3688 in Lecture Notes in Computer Science, Springer Verlag (2005) 122–135

[12] Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a safe subset of simulink/stateflow into lustre. In: Proc. EMSOFT'04, ACM (2004)

[13] Sims, S., Cleaveland, R., Butts, K., Ranville, S.: Automated validation of software models. In: Proc. ASE 2001, IEEE Computer Society Press (2001)

[14] Pingree, P., Mikk, E., Holzmann, G., Smith, M., Dams, D.: Validation of mission critical software design and implementation using model checking. In: Proc. Digital Avionics Systems Conference, IEEE (2002)

[15] The MathWorks Inc.: Using simulink. Available from URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf (2008)

[16] Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8** (1987) 231–274

[17] The MathWorks Inc.: Stateflow and stateflow coder 7 user guide. Available from URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf (2008)

[18] The MathWorks Inc.: Simulink design verifier 1 user guide. Available from URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/sldv/sldv_ug.pdf (2008)

[19] Selic, B., Rumbaugh, J.: Using UML for modeling complex real-time systems. `http://www.ibm.com/developerworks/rational/library/139.html` (1998)

[20] Kamel, M.: On the visual modeling and verification of concurrent systems. Master's thesis, University of Waterloo (1999) Available from URL http://fee.uwaterloo.ca/ m2kamel/research/thesis.ps.

[21] AUTOSAR GbR: Requirements on memory services v. 2.2.1. Available from URL http://autosar.org/download/AUTOSAR_SRS_MemoryServices.pdf. (2008)

[22] Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)

[23] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st Conference on Software Engineering (ICSE), IEEE Computer Society (1999)