UNIVERSITY OF APPLIED SCIENCES
TECHNIKUM WIEN

Master Embedded Systems
System Architecture and Engineering
SS-2008

Introduction to Embedded Software Verification

Thomas Reinbacher, BSc (es07m106@technikum-wien.at)

Abstract

Since today's embedded systems software projects are approaching an unprecedented level of design and implementation complexity, traditional concepts like software testing and debugging are reaching their limits of useful application. The whole software industry is striving for solutions to compensate this software verification gap and formal verification methods like model checking and theorem proving are widely accepted as promising and contrary approaches. While model based formal verification is well established in the context of Software Engineering, it has played a minor role in context of Embedded Systems in the past. The reasons are quite diverse like, e.g., past model checking tools were only capable of handling small designs with a few hundred lines of code and the needed behavioral models were, especially for Embedded Systems, quite complex to generate and in most cases too inaccurate to derive reliable assumptions whether the design satisfies particular properties of its specification. Fortunately, recent advances in research contributed to the ever-more integration of formal verification into the embedded systems design flow.

This is an introductory paper about embedded software verification. Basic terms like Testing, Debugging and Verification are discussed and summarized. Furthermore, formal verification methods are introduced and explained such as Model Checking and Theorem Proving. Major strengths and weaknesses of formal verification methods are pointed out. Finally, an approach to apply model checking for microcontroller assembly code is presented.

Keywords

embedded software verification, formal verification methods, temporal logic, model checking, theorem proving, model checking microcontroller code

1. Motivation - The verification problem

The main problems at the root of most verification bottlenecks center around the relationship between software design complexity, verification complexity, engineering resources and time as well as resource constraints. As designs grow ever more complex, the verification effort they are causing grows exponentially or as a rule of thumb - as design doubles in size the conventional verification effort can easily quadruple. As a result the verification effort of nowadays embedded systems projects consumes up to 70 percent of the entire engineering budget (assumptions based on [G07]).

As one can imagine, caused by the increasing reliance on embedded software, traditional methods like Debugging and Testing are insufficient to master the design and verification challenges of safety critical, highly reliable embedded systems that we are encountering in state-of-the-art computer systems.

2. Verification vs. Debugging vs. Test

The evolution of software that satisfies its expectations is the primary target of a successful software development process. For the achievement of this goal various practices must be applied throughout the development process. One has to keep in mind that terminology in this area is often confusing and the same concepts are used in literature for different purposes (compare [C88] and [HS02]). This section is used to clarify terminology by explaining three fundamental concepts.

Debugging is the process of analyzing and possibly extending the given program that does not meet its specification in order to find a new program that is close to the original and meets the specification. As a result, debugging is the process of *diagnosing* the precise nature of a *known error* and *correcting* it afterwards [C88].

Verification is the process of proving or demonstrating that the program correctly complies to its specification. In a more formal way, verification shows that the program satisfies a given specification by a (mathematical) prove.

Testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results [IEEE83]. In testing the implementation of the system is taken as already realized and is stimulated with certain (hopefully well-chosen) inputs and the reaction of the system is observed. Whereas verification proves conformance with a given specification, testing *finds cases* where a *program* does *not meet its specification*. It is important to note, that testing can never be complete, since it is built up solely of observations. Hence, only a small subset of all possible instances of a systems behavior can be taken into consideration.

A often cited quote [D69] by Edsger W. Dijkstra from the early 1970's clearly highlights the main drawback of software testing.

Program testing can be used to show the presence of bugs, but never to show their absence!

A contrary concept to debugging and testing is to prove that a system operates correctly. That is done in a similar way as one can prove that, e.g., the result of an addition of any two positive numbers is always a positive number itself. The technical term for this (mathematically) demonstration of the correctness of the system is *formal verification*.

The basic idea is to construct a formal model of the system under investigation, that covers the possible behavior of the system. The correctness requirement (aka claim) is written in a formal specification language, that describes the required behavior of the system. Based on this two input parmeters, formal verification proves whether the formal system model agrees with the given correctness requirement.

Whereas Testing and Debugging are well integrated in today's embedded software development, the use of formal method has only recently gained attention of the embedded software industry. As a result, this paper is focused mainly on formal methods and their applications in the context of embedded software.

3. (Embedded) Software Verification

Rudimentary proofs of correctness about computer programs have been available since the early days of computer science, but academic developments were routinely ignored by industry justified by the supposed "impracticability" of the advances in research [O08].

Basically, literature distinguishes two main areas of formal software verification approaches. The first one is a rather mathematical related one, called *Theorem Proving*. In theorem proving a proof of correctness is achieved through the derivation of a theorem. However, software verification can also be achieved without mathematical proofs. The more popular approach to formal verification is called *Model Checking* and is very well received in modern-day software development processes. This section will give a brief overview about these two approaches.

3.1. Verification through Theorem Proving

Based on reseach performed by Hoare and Dijkstra in the 1970s, the term theorem proving refers to derive postconditions based on preconditions and the executable program. Hoare introduced the concept of the "*Hoare triple*" [H69] which describes the theorem proving concept:

$$\{Q_{PRE}\}P\{R_{POST}\}$$

Q_{PRE}: *precondition*
R_{POST}: *postcondition*
P: *entire program or single function call*

The "hoare triple" can be read as "*if the precondition Q_{PRE} holds before program P is executed, R_{POST} holds after the execution of P*".

In theorem proving, axioms and rules of interference are used to derive R_{POST} based on Q_{PRE} and P [O08]. Theorem proving (or automated deduction) involves two disciplines: *mathematics* and *logic*. Implementation and specification are expressed as formulas in a formal logic. A key difference between theorem proving and model checking is that theorem provers do not need to entirely visit a program's state space in order to verify certain properties.

One has to keep in mind that theorem proving is always an iterative process. Properties are proved interactively. This implies that a user must generally think about the proof step to apply and the theorem prover then executes this step, gives the new proof state as output and awaits the next proof steps. This requirement implies that an (experienced) user has to guide the tool and as a result, theorem proving is rather a time-consuming venture than a "idiot-proof, one-click" verification step. This is probably the main reason why theorem proving is only rarely accepted by the software industry.

The rather (complex) mathematical theory behind this concept is far beyond the scope of this paper. The interested reader is referred to relevant literature [S01][O08].

Theorem Proving - points to remember

- Prove correctnes of the program through derivation of a theorem
- Explore only certain system states of the model that are needed to prove the theorem
- Derive the system's model and the property (or claim) as a logical formula
- User intervention and guidance is necessary for the verification process
- Theorem Proving necessitates expertise in the field. To prove small theorems large human investment is necessary
- Able to show the absence of errors
- Can deal with infinite state spaces
- The theory behind theorem proving is quite complex
- How to verify the theorem prover itself?

3.2. Model Checking in a nutshell

Model checking is basically a fairly straightforward brute force exploration of the states of a given system. The essential concept behind this approach is to prove whether a given model (a set of system requirements or a simulation model) satisfies a certain specification property. From a logical viewpoint, the system is described by a semantical model (Kripke structure), and the abovementioned property is described by a logical formula.

As a result, proving a certain property is performed by determining the truth of formulas in certain system states. In order to be able to perform model checking, one needs a modeling language in which the system can be described, a notation for the formulation of properties and algorithms to walk through the state space. As

described in a later section of this paper, properties are formulated in a temporal logic. In recent years there has been considerable research on efficient search algorithms in order to ensure minimal effort when traversing system states. At the time model checking was first introduced in the 1980s, it was only possible to handle small and simple systems with a maximum of a few thousand states. With increased computing power and by using more elaborated data structures for storing system states, it is now possible to check systems with up to a few million states.

Recapitulating, a more formal definition of the model checking problem can be found in [R07]:

Given a system model M and a formula ϕ, model checking is the problem of verifying whether or not ϕ is true in M.

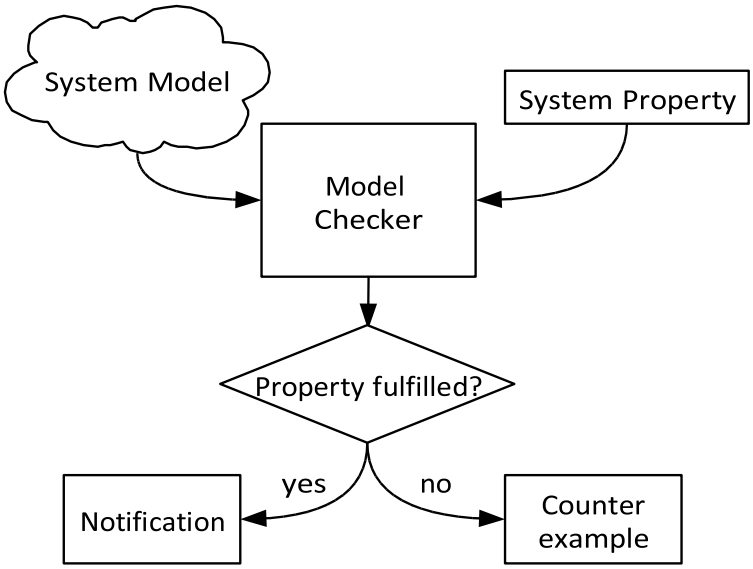

Figure 1: A typical model checking working flow.

As shown in Fig. 1, traditional model checking is composed of three major steps:

Define a formal model of the system that is subject to verification by creating a model of the system in a language that fits the model checker's input language. Those modeling languages are usually tight coupled to the model checker itself, like the *PROMELA* language of the *SPIN* model checker [H97].

Provide a particular system property that should be proved. In other words, a special kind of a question about the system's behavior is formulated that should be answered by the model checker.

Invoke the model checking tool and receive a notification whether the given system property was fulfilled or not. In case the system property could not be verified, a counterexample is generated to finger-point to the source of error in the simulation model.

Model checking is an automatic, model-based, property verification approach [R07].

Using Kripke structures to represent a system's behavior

In model checking finite (nondeterministic) state machines are used to represent the behavior of the system. A special type of these state machines are Kripke structures. Every single state is labeled with boolean variables (atomic propositions), which are the evaluations of expressions in that state. These expressions correlate to particular system properties, e.g. boolean expressions over variables or registers.

A Kripke structure M is represented as an ordered sequence of four objects:

$$M = (S, s_0, R, L)$$

S: *finite set of states* s_0:
initial state
R: *transition relation*
L: *interpretation function*

The *transition relation* specifies for each state whether and which successor states are possible.
The *interpretation function* labels each state with the set of *atomic propositions* that are true in that state.

Coffee vending machine example

A simple coffee vending machine model is introduced to demonstrate a possible application of Kripke structures. It's textual functional description reads as follows:

- After inserting a coin, the user can choose her/his favorite coffee.
- A coffee is only brewed after a valid selection is made.
- The user is able to abort the procedure at any time.

Figure 2 shows the resulting Kripke structure, with all the states, transitions and state variables.

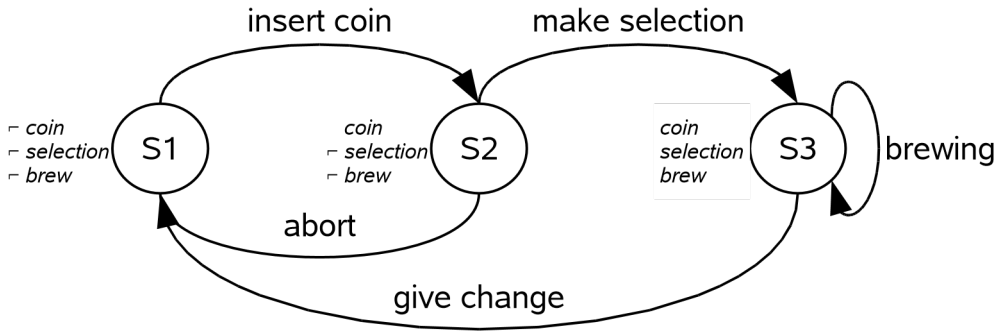

Figure 2: The coffee vending machine Kripke structure.

From transition systems towards computation paths

By simply unwinding the Kripke structure it is possible to visualize the resulting computation paths. (Fig. 3)

CTL at a glance

Model checking is based on mainly temporal logic. One representative is **CTL** (Computation Tree Logic). CTL is a combination of a linear temporal logic and a branching-time logic and was introduced by Clarke & Emerson in 1980 [CE81].

In a linear temporal logic various operators are provided to describe events along a single computation path. In contrary, a branching-time logic provides operators to quantify over a set of states that are successors of a given (the current) state. CTL combines these two kinds of operators.

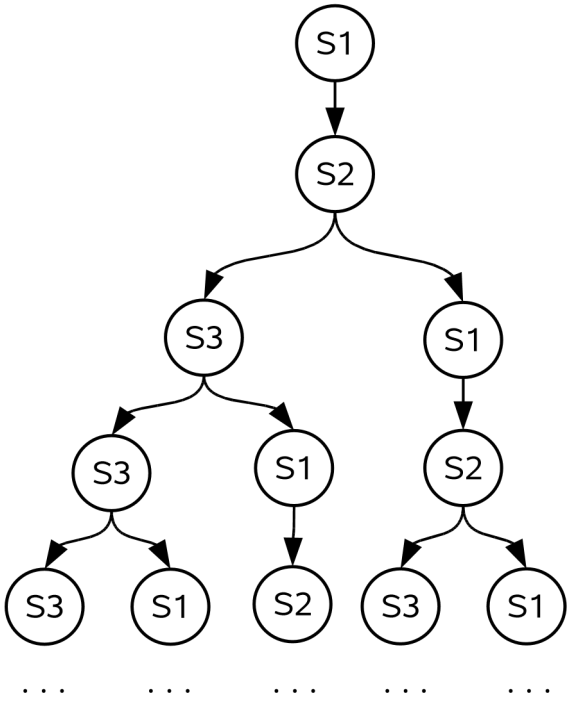

Figure 3: The first few computation paths of the coffee vending machine example.

CTL Syntax

Each CTL operator consists of two symbols. The first one is either **A** (which stands for **A**ll paths), or **E** (there **E**xists a path). The second one is either **X** (the ne**X**t state), **F** (in **F**uture state), **G** (**G**lobally in the future) or **U** (**U**ntil). **A** and **E** are *path quantifiers*, **X**, **F**, **G** and **U** are *linear-time operators*.

In model checking formulas to express certain properties are constructed (solely) by combining path quantifiers and temporal operators:

A	for every path
E	there exists a path
Xϕ	ϕ holds next time (i.e.: the next state)
Fϕ	ϕ holds sometime in the future
Gϕ	ϕ holds globally in the future
ρ**U**ϕ	ρ holds until ϕ holds

Expressing a CTL system property for the coffee vending machine example For the coffee vending machine some conceivable system properties could be:

(1) In spoken language:

A coffee is brewed after a selection was made.

In CTL syntax this would result in:

$$\mathbf{E}[(\textit{selection})\mathbf{U}(\textit{brew})]$$

As long as no selection was made, no coffee is brewed in any cases. After a selection is made, coffee is brewed for sure.

(2) In spoken language:

A coffee is brewed in any case.

In CTL syntax this would result in:

$$\mathbf{AG}[(brew)]$$

Coffee will be brewed from now on, no matter what happens.

The dark side of model checking: State Space Explosion

Kripke structures can become large and complex for real life systems. The logical consequence is, that the state space of such a system is tremendous, or - in the worst case - even infinite. Hence, without any optimizations of the state space it is impossible for most of the applications to explore the entire state space, since time and memory are limited.

If model checking should be performed on a typical embedded system, a variety of components could result in a state space multiplication. Suppose that the embedded system uses an *n*-bit ADC to read environment variables. Every time the ADC value is read, state space is expanded by 2^n. For a 16-bit wide ADC this would result in adding additional 65536 paths. Fig. 4 visualizes such a transition.

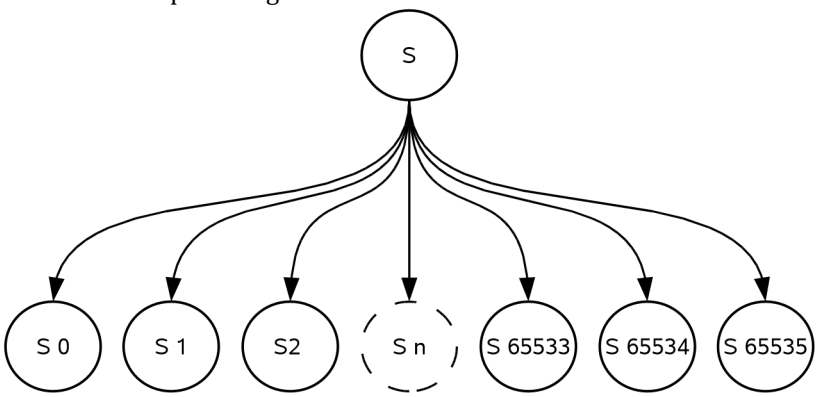

Figure 4: State Space Explosion after reading ADC values.

Model Checking - points to remember

- Performing an exploration of the entire model in order to verify a certain property
- The system's model is regarded as (finite) state machine
- The property to be proved is specified as logical formula
- Verification is the activity to check whether the structure is a model of the formula or not
- Keep in mind: generated model ! real life system
- Able to show the absence of errors
- Can provide counterexamples
- Fully automatic approach
- Sometimes limited to "small models"
- Problems with infinite state spaces
- Any verification using model checking is only as good as the model of the system
- Any verification using model checking is only as good as the stated claim

- How to verify the model checker itself?

3.3. Applying Model Checking to Microcontroller Assembly Code

In recent years, a trend in model checking can be observed that lifts this verification approach to a more broaden level of applications. Instead of formulating the required system model of the real life problem in a way the model checker can interpret, models are derived directly from source code. For example, a few selected model checking tools are able to generate a system's model from the ANSI C code [H00].

In the context of embedded systems the heavy use of microcontroller and compiler specific C extensions is very common in order to access microcontroller specific functions, like enabling/disabling interrupts, reading from and writing to periphery or specific data types. Hence, model checking of ANSI C code does not cover the whole needs to verify software for microcontrollers, or more general software for embedded systems.

Recently, model checking assembly code became the focus of research projects [SRWK06]. It has some tremendous advantages compared to model checking programs written in high level programming languages. The code that is deployed to the hardware is checked and not just an intermediate representation. Any errors introduced during the development process can be found. (e.g. compiler errors, toolchain errors, wrong periphery setup and errors not visible in the C code at all).

Assembly code is always tightly coupled to the underlying microcontroller. Thus, in order to apply model checking on assembly code a whole model of the microcontroller is needed. As one can imagine, these models are complex and labour intensive to create. On the other hand, after having created the microcontroller model, every piece of software that targets this specific microcontroller can be exposed to model checking.

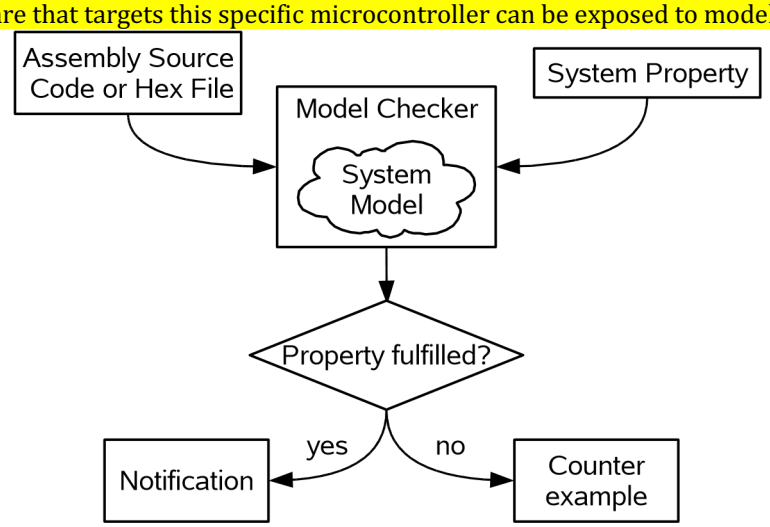

Figure 5: Extended Model Checking Design flow [R08].

The most notable advantage of this approach over traditional model checking is, that (almost) no expert knowledge on system modeling is presumed from the user. This results in good usability of such a tool and low tool-related training effort.

With *[mc]square* (**M**odel **C**hecking for **M**icro **C**ontrollers) [SRWK06], the Technical University of Aachen developed an explicit model checker which is precisely tailored for formal verification in context of microcontrollers that represents such a "one-click" tool.

In Section 3.2. we have presented the traditional model checking design flow. With the above mentioned ideas and concepts this design flow (Fig. 5) is even more simplified (from the user's point of view).

Applying Model Checking to Microcontroller Assembly Code - points to remember

- Errors of all development stages are detectable
- Assembler instructions are (relatively) easy to handle
- Instructions are clean and well documented

- Operates as close as possible to actual execution
- Hardware dependency
- Large effort for supporting new microcontroller families
- Bigger State Space (one have to know the whole μC status for each stage)

4. Conclusion

After having discussed the basics of Theorem Proving and Model Checking, the concept of Model checking for embedded software was introduced. Whereas Theorem Proving is rather an mathematical venture, model checking is based on proving properties along computation trees. CTL is a powerful temporal syntax in which properties are specified.

There is no clear answer on which one of the aforementioned verification approaches is more powerful. Model checking is not considered to be "better" than theorem proving. Both have different problems and their benefits.

Model checkers are rather user-friendly and have a good usability. The use of theorem provers requires considerable expertise of the user to guide and assist the tool. Model checking is often able to generate a counter-example which can be used as input information for following debugging sessions. The problem of State Space Explosion is a serious one when models are approaching the size of real life systems. Whenever the verification of infinite state spaces is needed, theorem proving shows its strengths.

There is a rapidly increasing interest of the industry in formal verification methods. Leading software engineering companies have built their own model checkers and initiated their own research groups and research centers, focused on formal verification.

Summarized, formal verification often leads to early detection of errors in the requirements and/or the design, thereby leading to large savings in later work. At the same time, projects utilizing (formal) software verification are in most cases less error-prone than systems developed without these techniques.

5. Glossary and Abbreviations (in alphabetical order)

ADC Analog-to-Digital Converter is an electronic integrated circuit, which converts continuous signals to discrete digital numbers. (based on [W08])

CTL (Computational Tree Logic) is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be 'actual' path that is realised.

Formal Verification is the mathematical demonstration of the correctness of a system.

Kripke Structure is a type of nondeterministic finite state machine used in model checking to represent the behaviour of a system. It is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. A labelling function is used to assign a set of properties to each states.

Model Checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model.

PROMELA (Process or Protocol Meta Language) is a verification modeling language. PROMELA models can be analyzed with the SPIN model checker [H97].

Property/Claim is a certain behaviour of the system that should be proved right or wrong

Theorem is a statement, which is logically derived from the axioms/rules of a system through the application of a prove. (based on [W08])

Theorem Proving is a concept of formal verification where the correctness of a system is proved trough the derivation of a theorem.

6. Literature

[SRWK06] B. Schlich, M. Rohrbach, M. Weber and S. Kowalewski, *Model Checking Software for Microcontrollers,* Technischer Bericht AIB-2006-11, RWTH Aachen, 2006

[R08] T. Reinbacher, *MCS-51 Simulator Integration into the [mc]square Model Checker,* Technical Report, Institute of Embedded Systems, FH Technikum Wien, 2008

[CE81] E. Clarke, E. Emerson, *Desing and synthesis of synchronisation skeletons using Branching Time Temporal Logic,* In Proc. Workshop on Logics of Programs, Lecture Notes in ComputerScience, Vol. 131, Springer, 1981

[R07] F. Raimondi, *Computational Tree Logic and Model Checking - A simple introduction,* Validation and Verification Course Notes, University College London, 2007

[H97] G. Holzmann, *The Model Checker SPIN,* IEEE Transaction on Software Engineering, Vol. 23, No. 5, pp. 279-295, May 1997

[H00] J. Holzmann, *Logic Verification of ANSI-C Code with Spin,* Proc. SPIN2000, Springer Verlag, LNCS 1885, pp. 131-147

[H69] C. Hoare, *An axiomatic basis for computer programming,* Communications of the ACM. 12(10): 576585, Oktober 1969.

[IEEE83] ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology, 1969

[D69] E. Dijkstra, *The Humble Programmer,* ACM Turing Lecture 1972

[C88] J. Collofello, *Introduction to Software Verification and Validation,* SEI Curriculum Module SEI-CM-13-1.1, Arizona State University, 1988

[HS02] B. Hailpern, P. Santhanam, *Software debugging, testing, and verification,* IBM Systems Journal, Vol 41, No 1, 2002

[S01] M. Schumann, *Automated Theorem Proving in Software Engineering,* ISBN-10: 3540679898, Springer, Berlin, 2001

[O08] M. Ouimet, *Formal Software Verification: Model Checking and Theorem Proving,* Embedded Systems Laboratory Technical Report (ESL-TIK-00214), MIT, Cambridge, 2008

[G07] S. Gru¨nfelder, *Vier Artikel zum Thema Software Testing,* Software Testing Course Notes, FH Technikum Wien, 2007

[W08] Wikipedia, the free encyclopedia, *Theorem,* Website, visited March 2008

7. List of Questions

(1) What is refered to as the verification gap?

(2) Briefly explain the following terms in general and point out their differences:

(2a) Debugging

(2b) Verification

(2c) Testing

(3) What is refered to as the Model Checking Problem?

(4) Name and briefly descibe the three basic steps that you have to pass through when applying a typical model checking session?

(5) How would you model the procedure of “ordering a glass of beer” as a Kripke structure? Transform the found Kripke structure into computation paths and try to state a claim that you want to prove (i.e. *I will get my beer for sure ...*)

(6) Descibe the main application areas of formal verification. What are the strenghts and weaknesses?

(7) What is the main concept behind Theorem Proving? Why do most of the available tools need heavy user guidance in order to achieve reasonable results?

(8) State space explosion is one of the main challenges in model checking. Consider a microcontroller which is operating in an embedded system. What actions would lead to tremendous expansion of the state space?

(9) What advantages does model checking for assembly code provide?