Comparison between formal verification and runtime verification of state machines in modeled software controllers

Leonardus M. M. Veugen
tis.veugen@gmail.com

Introduction

The realization of a complex software controller component can be speed up significantly by applying Model Driven Engineering (MDE). Many institutes have recognized the benefits of MDE and consequently launched tools that can be very helpful for deriving design documentation from a model, generating source code, verifying model properties, and/or generating test cases. Selecting a suitable tool for prolonged usage requires a thorough evaluation about featuring, costs, support, training, intuitive usage, etc. Most of these characteristics are important for an engineer to shorten the learning curve and to increase productivity. This paper addresses modeling tools that include verification of state machines since this key feature accounts for a productivity boost. Almost all such verifying tools provide formal verification based on a formal language such as CSP *[1]*, SDL *[2]*, or LOTOS *[3]*. However, applying these tools can hamper the productivity of a software developer who is only familiar with common design techniques and programming languages. For this reason we propose an alternative approach based on exhaustive runtime verification which closely relates to a developer's knowledge and experience. The tool TismTool *[4]* supports this new approach.
The setup of this paper is as follows. Firstly, the rationale for using a verifying tool is given. Then, the 2 considered techniques of formal verification and runtime verification are characterized. Finally, the differences between the techniques are discussed.

Tool rationale

The suitability of a verifying tool depends on the application domain to be modeled. A model with a simple state machine for a *door* with 2 states (*open* and *closed*) hardly needs tool support. Application source code generated from a model relieves a developer from some burden, but this profit can be nullified by the additional effort to glue that code in the rest of the software system. The *door* model becomes more interesting if asynchronous activities happen when receiving events from door sensors. Stimuli from both a client and lower level components may cause race conditions which have to be solved carefully. But, the investment in a verifying tool should still be questioned for occasionally modeling a simple controller.
Without any doubt a verifying tool is valuable for modeling complex controllers. The MediaPlayer, elaborated in the Appendix, is a typical example of such a controller based on practical experience. It has dozens of (hierarchical and orthogonal) states, many

asynchronous and/or autonomous events, multiple subcomponents and a thread. The correctness of such a controller with several (sub)components in a multi-threaded environment is hard to prove without tool support. Verifying this complex controller demands a tool with the ability of executing an aggregated verification session encompassing multiple components and multiple threads.
The choice of a suitable tool should be made after mature consideration since it concerns a long term investment. The existing technique of formal verification will be examined thanks to its years of merit. Furthermore, a novel technique relying on exhaustive runtime verification will be investigated. One of the few tools supporting this approach can be found at *[4]*.

Formal verification

A generic toolchain for model verification is shown in Figure 1.

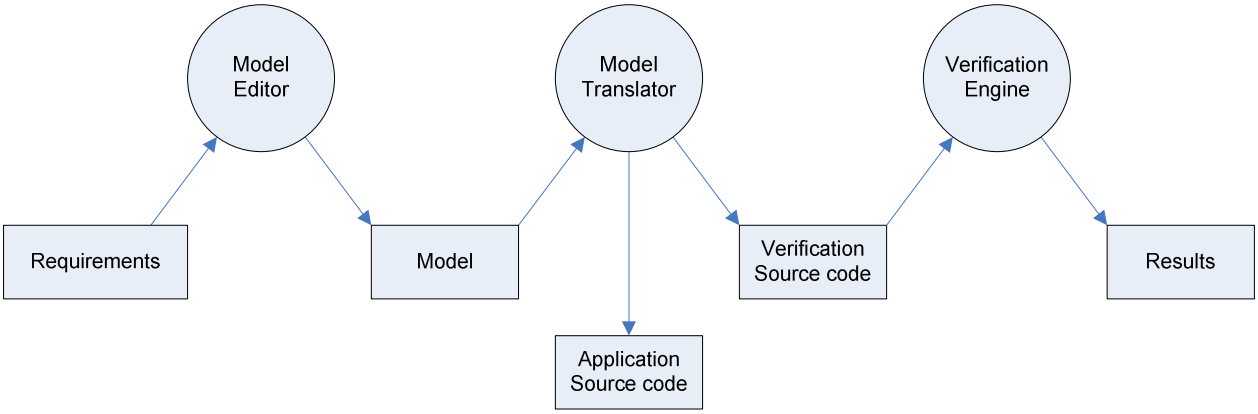

Figure 1 Toolchain template

The ModelEditor facilitates the construction of a model based on the input requirements; this program can be a graphical editor (e.g. *[5]*), or a tabular editor (e.g. *[6]*), or even an ASCII editor. The ModelTranslator generates verification source code from the model; it also generates source code for application purpose in one or more (programming) languages. The VerificationEngine performs the verification and reports its findings. This program acts as an interpreter, or occasionally it must be compiled and linked first (e.g. SPIN *[7]*).
Currently there are already dozens of suppliers offering toolchains for formal verification *[8]*, both commercially as well as in the open source domain. Of course there is a strong relationship between the format of the model and the verification source code on the one side and the individual tools of a chain on the other side. Despite the variation in formats and tools the following general observations can be made about the formal verification technique.
1. The model is described by one of the (many) formal languages. Some of these languages have been officially standardized, such as SDL *[2]*, or LOTOS *[3]*. CSP *[1]* is a

well-known example serving as de facto standard arising from the academic world, but also adopted for commercial purposes. It has been extended in the past 30 years leading to many variations. Promela *[7]* is a proprietary language for the popular SPIN toolchain.
2. The mentioned formal languages have been successfully applied for algorithm proving. Some of them originate from the data communication world for verifying protocols.
3. For observing real-time behavior some of the formal languages incorporate the notion of time. (Timing aspects will not be considered in this paper.)
4. The verification takes into account not only states of the state machines but also values of variables. These variables are restricted to discrete types: integers (with a finite range), booleans, or enumerations.
5. The so called *state space explosion* is caused by the orthogonality of state machines and the usage of (many) integer variables with big ranges.
6. The syntax of formal languages has limitations, such as absence of floating/double variables. Also, library functions of the target programming language cannot be called in a formal model.
7. Syntax statements that are not supported by the formal language must be realized via foreign components. In such a way a distinction can be made between control processing by the model and data processing by foreign components.
8. The model to be verified can be categorized as a Platform Independent Model (PIM). The generated verification source code is only intended for verification purpose. The ModelTranslator can generate application source code for the supported programming languages. Many tools in the academic world support only a single application language, and some of them even none if the intention is only algorithm proving. The big advantage of PIM is that the verification engine can be optimized for its dedicated purpose.
9. The ModelTranslator is assumed to generate correct application source code; a successful verification can't give any guarantees since it acts on differing (verification) code. Also, deploying the application source code correctly is not enforced by the verification step.
We will go deeper into some of the above observations. Comparing two floating variables must be done in a foreign component. And at some toolchains a foreign component is even needed for comparing 2 integer variables. Not surprisingly the imposed drop of productivity might keep a software developer away from applying formal verification. Furthermore, the usage of foreign components may introduce side-effects, as is illustrated in Figure 2.

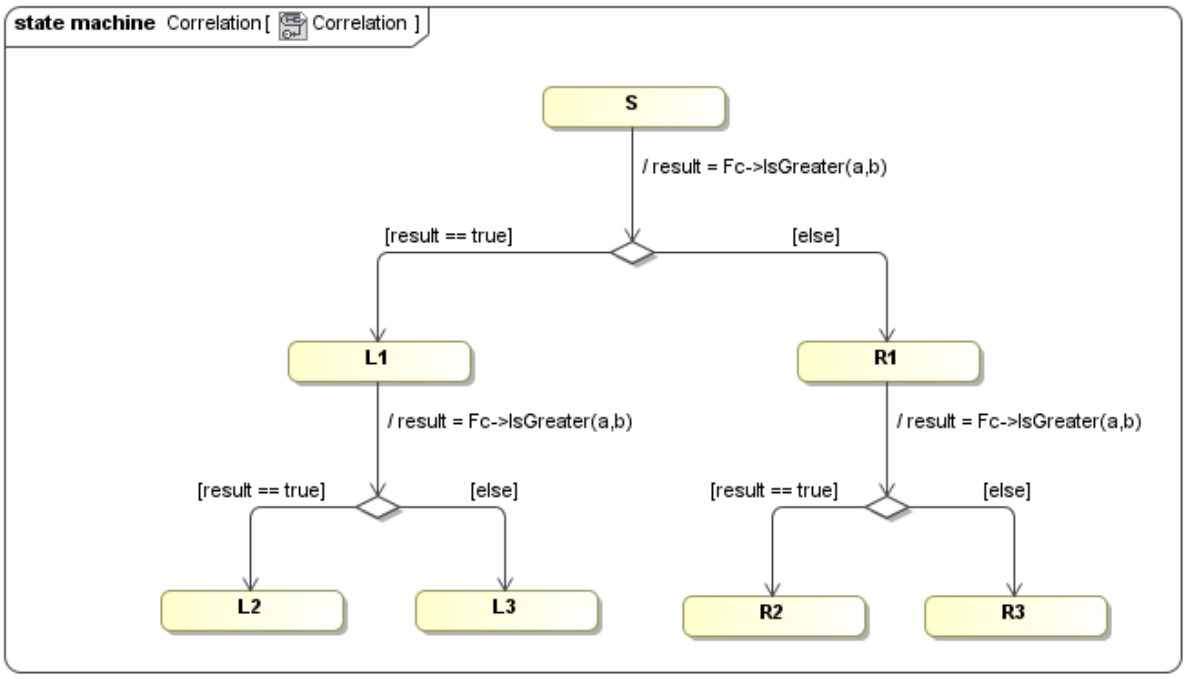

Figure 2 Fragment of foreign component usage

Calling Fc->IsGreater(a,b) at the foreign component Fc is always followed by branching into two states due to the 2 possible boolean results. So, the formal verification has to deal with states L3 and R2, and their sequel. This simplified example shows that these states will never be visited in the actual application. The point is that utility functions of a foreign component have no state behavior, so that correlation between related calls is lost. Of course the problems in this artificial example could be prevented by removing the second comparison, but that might not be a solution for more complex situations.

Exhaustive runtime verification

According to reference *[9]* the term Runtime verification refers to "*any mechanism for monitoring an executing system*". We consider this definition also to be valid for a certain part of a system of our interest. Nevertheless, the corollary in the reference can be argued concerning: "*Runtime verification avoids the complexity of traditional formal verification techniques … by analyzing only one or a few execution traces and …*". The dispute concerns the fact that our approach entails an exhaustive execution of all traces for the verified system.
The analogy with the toolchain sketched in Figure 1 will be drawn for the tool TismTool *[4]* in Figure 3.

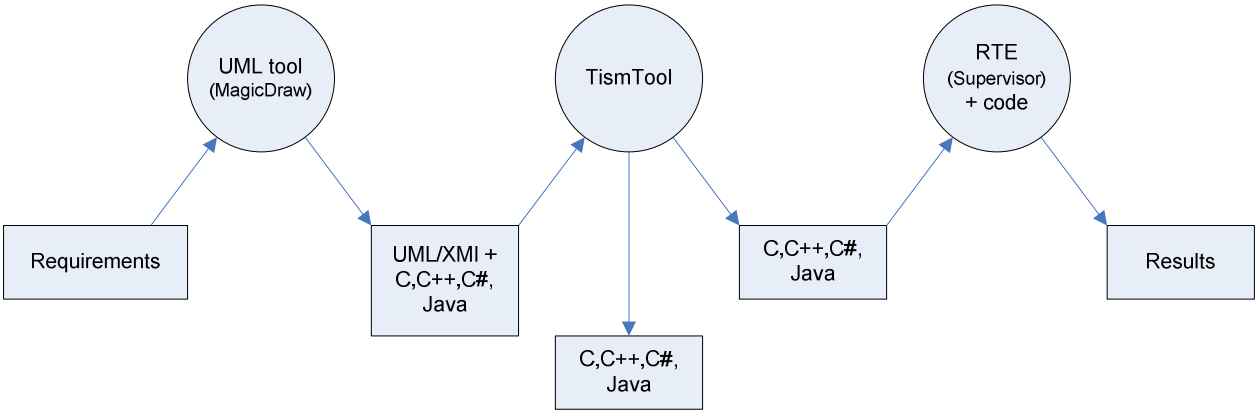

Figure 3 Toolchain of TismTool

The UML format *[10]* has been chosen for model construction because of its widespread usage. Software developers are familiar with this notation, and the diagrams always serve as documentation. Since the target programming language (C, C++, C#, or Java) is known upfront, code snippets in the chosen programming language can be included in the model. The UML model must be saved in the standardized XMI format *[11].* TismTool reads the XMI file and generates source code in the target language for verification of the interface protocols and verification of the specified architecture(s). Also source code can be generated for the final application. The verification code is almost the same as the application code: some additional verification code administers the visiting of states and transitions. The generated (and optionally foreign) source code must be compiled for the target platform and linked with a Runtime Environment (RTE). Two types of RTE's exist, namely one for the normal application and one for verification purpose. The latter includes a supervisor with the verification engine. Implementation of such an engine for each programming language can be considered as a major disadvantage of this approach. However, reuse of the verification algorithm reduces the implementation effort. And, since the RTE's have already been implemented for the 4 mentioned programming languages no additional effort is needed by a user of the tool.
The source code of the verification engine is freely available because a developer must be able to compile it for the specific target platform. The resulting verification executable runs on the target platform. This explains the term runtime verification since the instrumented application source code is verified within application circumstances.
The verification engine performs an **exhaustive search** of all possible execution scenarios. This distinguishes our approach from other Runtime verification methods that are based on a selected set of test cases. During the verification all scenarios are remembered, and also all (intermediate) state configurations must be stored. The involved memory resources might be a bottleneck on the target system, especially for small embedded systems. The processing power of the target system might be another limiting factor. If resources on a target host are insufficient for verification, then one can resort to another host system provided that no target specific system calls are made.

The reported model defects deal with the following properties:

- Deadlock
- Livelock
- Unhandled function
- Rendez-vous error
- Queue overflow
- Unvisited state
- Unvisited transition

Most of these behavioral properties correspond to those checked by formal verification methods. The verification output optionally includes a log file from which TismTool can construct sequence diagrams. These diagrams aid in understanding and solving possible defects.

Discussion

Formal verification tools distinguish from other verification tools by the ability of performing an exhaustive search among all possible scenarios. In particular runtime verification tools are known for processing only a limited set of scenarios. However, the toolchain of TismTool is able to master an exhaustive search of runtime scenarios on the target platform. Comparing both techniques is meaningful now since this hurdle has been taken. The verified properties are similar for both techniques emphasizing on deadlock detection and livelock detection.
Some formal verification engines incorporate integer variables with a restricted range in their verification state administration. TismTool handles purely states in its administration; variables can be taken into account for verification by simply modeling them in a state machine, which will be part of the verified architecture.
The application domain of formal verification is focused on data communication protocols and mathematical algorithms. Most formal verification tools originate from academic institutes interested in theoretical problems. On the other hand TismTool is suited for much wider application domains including complex software controllers. TismTool has been developed based on a solid practical experience from industry. The UML notation is applied for modeling because of its widespread usage among software developers. Finite state machines should be modeled in a graphical language for decent maintenance and reviewing by colleagues. Anyway, the UML diagrams will automatically serve as design documentation.
The main distinction between the two verification techniques is the fact that a runtime verification session is executed on the target machine. The differences between verification source code and application source code are minimal in case of runtime verification. The immediate feedback from execution of the generated verification source code creates confidence in the generated application code. The relationship between the formally verified model and the corresponding application code is less tight because completely different tools are involved.
Runtime verification allows a developer to add source code to the model in the target programming language without any restrictions. This powerful feature means ease of

use accounting for a large increase in productivity. The developer does not need to design artificial foreign components. As mentioned earlier foreign components included in a formal model may cause models with states and transitions that will never be visited in the actual application. The formal verification will not notice such unvisited elements; on the other hand TismTool keeps track of (un)visited states and transitions during its exhaustive verification. In fact this tool can be used as a facility for analyzing a model, since the designer is able to manipulate input stimuli for the model.
The compared items above lead to the conclusion that exhaustive runtime verification with TismTool prevails over formal verification techniques.

Appendix: Complex software controller

A complex software controller is a component controlled by a client component and controlling several lower-level components. The architecture of a complex controller is illustrated by an example of the MediaPlayer *[12]*; refer to Figure 4 for its context diagram. For gaining a better understanding of the involved complexity this controller is described in more detail.

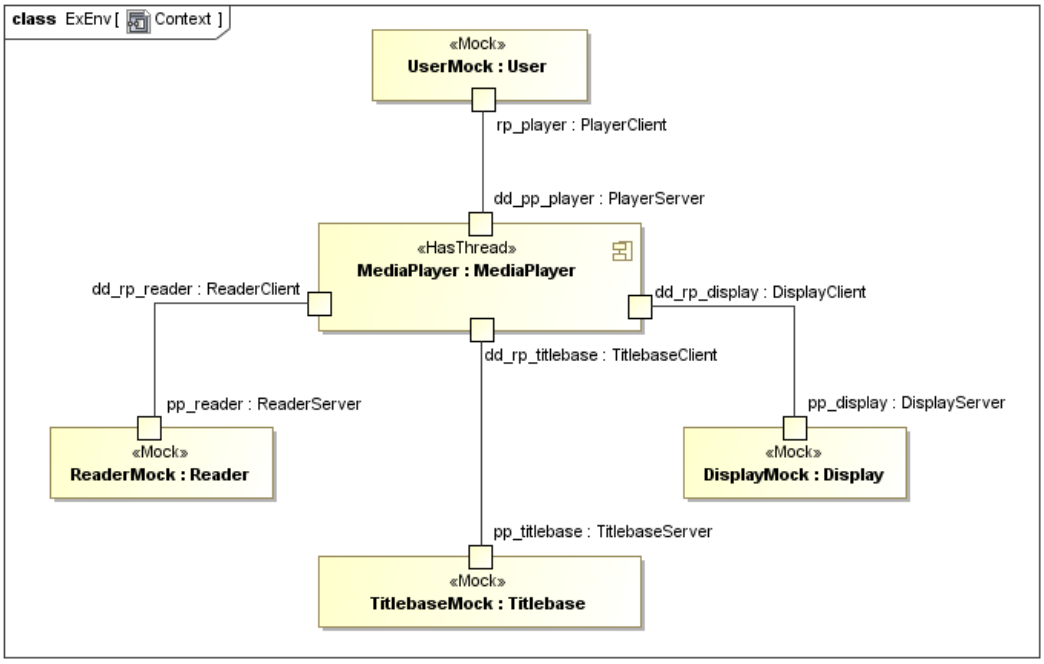

Figure 4 MediaPlayer context

The Titlebase component contains title information, especially the sector number at the beginning of the title, and sector numbers in the middle of the title needed for recovery after an error. The Reader component handles reading of the media data; for reasons of simplicity a single range suffices for reading (the remaining part of) the title. The Display component takes care of the presentation of the media data. In order to limit the example only the play modes *normal play* and *pause* are modeled. Errors can occur in reading data from the medium, and in decoding data due to authoring failures. The

MediaPlayer is able to recover from such errors. This functionality complicates the internal state machine to a big extent. Playback aborts in case of too many consecutive errors. The MediaPlayer executes in its own thread so that it is decoupled from hardware related activities and from the user interface. As a consequence the function calls and callbacks have to be marshaled into messages, which will be stored in the thread's queue.
The communication link between a pair of interacting components is considered as a protocol with one component having a server role and the other one having a client role. The client does synchronous function calls to the server, and the server invokes asynchronous callbacks to the client. The access to a server as well as to a client is shielded by a port. At each port a Protocol State Machine (PSM) describes the state behavior. Modeling a state machine at the client role is needed for verification purposes, and furthermore it helps the developer to resolve race conditions at the implementation of the client component.
The MediaPlayer is decomposed in several subcomponents: a Backend for the main activity and subcomponents acting as proxies for the controlled components. All subcomponents will have an internal design state machine. A callback to a subcomponent will be marshaled to a message that is queued in an internal message queue. The run-to-completion mechanism inside the thread is realized by servicing all messages inside the internal queue before accessing the external queue. A subcomponent's call to another subcomponent is executed immediately without marshaling or queuing. So, such a call executes differently compared with the marshaled call to the threaded component.

References

[1] Hoare, C. A. R., Communicating Sequential Processes, Prentice Hall International, 1985,.ISBN 0131532715.
[2] Specification and Description Language (SDL), http://www.itu.int/rec/T-REC-z
[3] The Formal Description Technique LOTOS, P.H.J. van Eijk et al., editors, N-H, 1988, ISBN 0444872671
[4] http://www.tismtool.com
[5] http://www.uppaal.org
[6] Sequence Based Specifications, http://community.verum.com/Files/Starting_ASD_Slides/40_background_sbs.pdf
[7] Holzmann, G. J., The SPIN Model Checker - Primer and Reference Manual, 2003, ISBN 0-321-22862-6
[8] http://en.wikipedia.org/wiki/List_of_model_checking_tools
[9] http://en.wikipedia.org/wiki/Runtime_verification
[10] Unified Modeling Language, http://www.omg.org/spec/UML/
[11] http://www.omg.org/spec/XMI/2.1.1/
[12] http://mediaplayer.tismtool.com