Model-Based Run-Time Error Detection*

Jozef Hooman [1,2] and Teun Hendriks [1]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
jozef.hooman@esi.nl, teun.hendriks@esi.nl
[2] Radboud University Nijmegen, The Netherlands

Abstract. We discuss the use of models for run-time error detection to improve user-perceived reliability of consumer electronics products. The aim is to apply the approach in industrial products and to embed error detection into a general run-time awareness concept. To study this concept, an awareness framework has been developed in which an application and a model of its desired behaviour can be inserted. It allows both time-based and event-based error detection at run-time.

1 Introduction

Modern consumer electronics devices, such as TVs or smart phones, contain vast amounts of intelligence encoded in either software or dedicated hardware. Hundreds of engineers develop and improve these "computers in disguise" for global markets but facing plenty of local variations. Complexity and open connectivity make it exceedingly difficult to guarantee total product correctness under all operating conditions. The final aim of our work is to improve user-perceived reliability of these devices by run-time awareness, i.e., allow a device to correct at run-time important, user-noticeable, failure modes. This paper presents an approach to insert run-time error detection as a first step towards awareness.

The work described here is part of the Trader project in which academic and industrial partners collaborate to optimize the reliability of high-volume products, such as consumer electronic devices. The main industrial partner of this project is NXP (formerly Philips Semiconductors), with a focus on audio/video equipment (e.g., TVs and DVD players). NXP provides the problem statement and relevant case studies which are taken from the TV domain. A current high-end TV is a very complex device which can receive analog and digital input from many possible sources and using many different coding standards. It can be connected to various types of recording devices and includes many features such a picture-in-picture, child lock, teletext, sleep timer, child lock, TV ratings, emergency alerts, TV guide, and advanced image processing. Similar to other domains, we see a convergence to additional features such as photo browsing, MP3 playing, USB, games, databases, and networking. Correspondingly, the amount

* This work has been carried out as part of the Trader project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the Bsik program.

of software in TVs has seen an exponential increase from 1 KB in 1980 to 64 MB in current high-end TVs. Also the hardware complexity is increasing rapidly to support, for instance, real-time decoding and processing of high-definition (HD) images for large screens, large data streams, and multiple tuners. Correspondingly, a TV is designed as a system-on-chip with multiple processors and dedicated hardware accelerators, to meet stringent real-time requirements of, for instance, HDTV-quality input at rates up to 120 Hz.

In addition, there is a strong pressure to decrease time-to-market, i.e., the increasing complexity of products has to be addressed in shorter innovation cycles. To realize many new features quickly, components developed by others have to be incorporated. This includes so-called third-party components, typically realizing audio and video standards, but also in-house developed components supplied by other business units. Moreover, there is a clear trend towards the use of downloadable components, to increase product flexibility and to allow new business opportunities (selling new features, games, etc.).

Given these trends, the complexity of hardware and software, and the large number of possible user settings and types of input, exhaustive testing is impossible. Moreover, the product has to tolerate certain faults in the input (e.g., deviations from coding standards or bad image quality). Hence, it is extremely difficult to continue producing products at the same reliability level. The cost of non-quality, however, is high, because it leads to many returned products, it damages brand image, and reduces market share.

The main goal of the Trader project is to prevent faults in high-volume products from causing customer complaints. Hence, the focus is on run-time error detection and correction, minimizing any disturbance of the user experience of the product. The main challenge is to realize this without increasing development time and, given the domain of high-volume products, with minimal additional hardware costs and without degrading performance.

This paper is structured as follows. In Section 2 the main approach is described. We list the main research questions in Section 3. Section 4 contains current results. Concluding remarks can be found in Section 5.

2 Approach

In observing failures of current products, it is often the case that a user can immediately observe that something is wrong, where the system itself is completely unaware of the problem. Inspired by other application domains, such as the success of helicopter health and usage monitoring [1], the main approach in Trader is to give the system a notion of run-time awareness that its customer-perceived behavior is (or is likely to become) erroneous. In addition, the aim is to provide the system with a strategy to correct itself in line with customer expectations. Important part of this approach is the use of models at run-time. Below, we list the main ingredients needed for the realization of such run-time awareness and correction, (illustrated in Figure 1) while giving examples from the TV domain:

– *Observation*: observe relevant inputs, outputs and internal system states. For instance, for a TV we may want to observe keys presses from the remote control, internal modes of components (dual/single screen, menu, mute/unmute, etc), load of processors and busses, buffers, function calls to audio/video output, sound level, etc.
– *Error detection*: detect errors, based on observations of the system and a model of the desired system behaviour. For a TV, this could be done using a state machine which describes mode changes in response to remote control commands. An alternative is to use a model of expected load and memory usage and compare this with the actual system behaviour.
– *Diagnosis*: in case of an error, find the most likely cause of the error, e.g. using architectural models of the system. Examples are diagnosis techniques that record data about executed parts of the system and the (non)occurrence of errors or techniques that use architectural models that include faulty behaviour.
– *Recovery*: correct erroneous behaviour, based on the diagnosis results and information about the expected impact on the user. Possible corrections include restarting particular components, resetting internal modes/variables, rescheduling software components, etc.

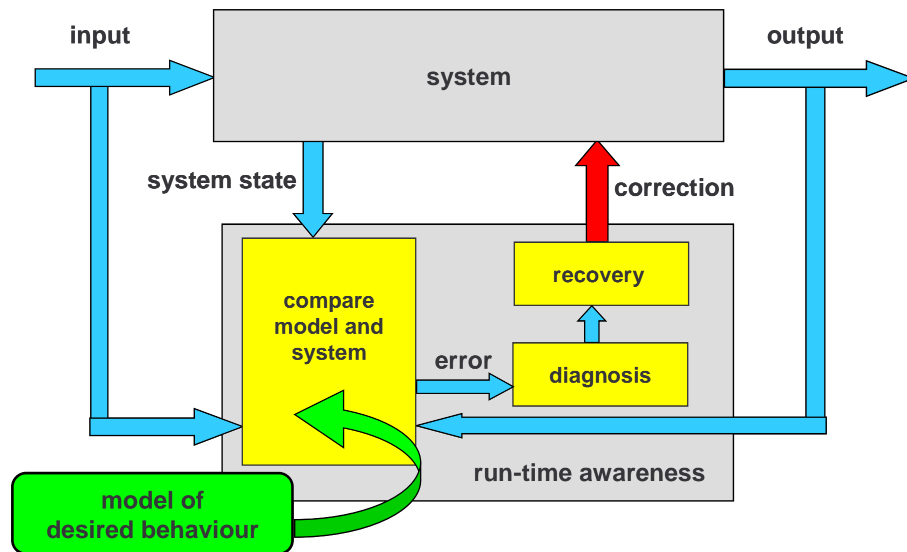

Fig. 1. Adding awareness at run-time

Note that the approach depicted in Figure 1 allows the use of partial models of desired system behaviour. We can apply this approach hierarchically and incrementally to parts of the system, e.g., to third-party components. In the remainder of this paper we focus on the error detection part, and refer to [2] for results on the other parts.

3 Research Questions

The research on embedding error detection in concrete industrial products consists of two main parts: (1) how to get suitable models, and (2) how to use these models for run-time error detection. We list a number of research questions for each part and briefly mention current results in Section 4.

Modeling

- Which part of the system has to be modeled? For a complex device such as a TV, it is cost-inhibitive to check the complete system behaviour at run-time. Hence, a choice has to be made based on the likelihood of errors and the impact on the user. Moreover, it is relevant to take into account which errors can be treated by the diagnosis and recovery parts of the awareness framework.
- Which models are most suitable for run-time error detection? For instance, which type of models is convenient and what is the right level of abstraction? Although we focus on user-perceived behaviour, some architectural modeling will be relevant to enable early detection of errors, i.e. before a user observes a failure. Another question is which models can be implemented efficiently with respect to performance overhead and memory.
- How to obtain suitable models? Typically, in the area of embedded systems, the number of models available in industry is limited and, hence, models have to be reconstructed.
- How to increase the confidence in the model; how to evaluate model quality and fidelity?

Using models at run-time

- How to avoid detecting non-existing errors? The concept is to compare system observations (e.g., output, states, load) with the values specified in the model of desired system behaviour, henceforth also called the specification model. This might lead to incorrect results for a number of reasons such as i) the use of an incorrect model (see the question about model quality above), ii) an incorrect implementation of the model, iii) a comparison at a wrong moment in time when the system is not stable, leading to the next questions.
- How to preserve model semantics in the implementation at run-time?
- When to compare system observations with the model? When the system is unstable, e.g., it is performing an action which takes some time, comparison may lead to wrong results. Should comparison be done time-driven, event-driven, or by a combination?
- When to report an error exactly? Should system and specification match exactly, or is a certain tolerance allowed? How much difference is allowed? Should a single deviation lead to an error or are a few consecutive deviations needed before an error is generated?

4 Results on Model-Based Error Detection

We present the current results of the Trader project on model-based error detection. First, we discuss work on obtaining a model of desired system behaviour, and next describe current research on a framework for run-time model-based error detection.

Experiences with modeling desired system behaviour

Since the TV domain is our source of inspiration and the focus is on user-perceived reliability, the first aim was to make a model that captures the user view of a particular type of TV in development. The model should capture the relation between user input, via the remote control, and output, via images on the screen and sound. Such a model did not exist. Neither could it be derived easily from the TV requirements, which, in common industrial practice, were distributed over many documents and databases.

Concerning the control behaviour of the TV, a few first experiments indicated that the use of state machines leads to suitable models. But it also revealed that it was very easy to make modeling errors. Constructing a correct model was more difficult than expected. Getting all the information was not easy, and many interactions were possible between features. Examples are relations between dual screen, teletext and various types of on-screen displays that remove or suppress each other. Hence, we aim at executable models to allow quick feedback on the user-perceived behaviour and to increase the confidence in the fidelity of the model. In addition, we exploit the possibilities of formal model-checking and test scripts to improve model quality.

Besides the control behaviour, a TV also has a complex streaming part with a lot of audio and video processing. Typically, this gets most attention in the requirements documentation. We would like to model this on a more abstract level, with emphasis on the relation with the control part.

These considerations led to the use of Matlab/Simulink [3]. Stateflow is used for the control part and the Image and Video Processing toolbox for the streaming part. A snapshot of a simulation is depicted in Figure 2. The Simulink model is shown in the middle, at the top, with on the left a (blue) Stateflow block called “TVbehaviour” and on the right, an image processing block called “Video”. The Stateflow block is a hierarchical and parallel state diagram. It is partly shown on the bottom, where the active states are dark (blue). External events are obtained by clicking on a picture of a remote control, shown on the left. Output is visualized by means of Matlab’s video player and a scope for the volume level, shown on the bottom right side in Figure 2.

The visualization of the user view on input and output of the model turned out to be very useful to detect modeling errors and undesired feature interactions. Since the model was changed frequently, we experimented with the tool Reactis [4] to generate test scripts to check conformance after model changes. This tool can also be used to validate model properties. Related functionality is provided by the Simulink Design Verifier.

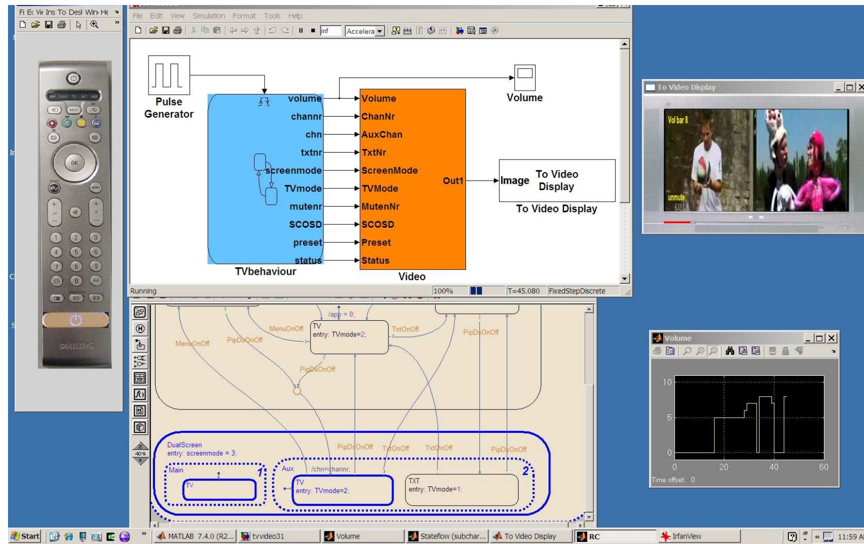

Fig. 2. Simulation of model of TV behaviour

A framework for run-time model-based error detection

To foster quick experimentation with the use of models at run-time inside real industrial products, e.g. a TV where the control software is implemented on top of Linux, we have developed a Linux-based framework for run-time awareness. A particular System Under Observation (SUO) can be inserted, needing only minimal adaptations to provide certain observations concerning input, output, and internal states to the awareness monitor. The specification model of the desired system behaviour is included by using the code generation possibilities of Stateflow. Hence, it is easy to experiment with different specification models. The awareness part also contains a comparator that can be adapted to include different comparison and detection strategies.

Before implementing the framework, it has been modeled in Matlab/Simulink to investigate the main concepts. A high-level view is depicted in Figure 3, illustrating the comparison of the volume level. To simulate the comparison strategy, we also made a second model for the SUO, this time a more detailed architectural model which also includes timing delays to simulate the execution time of internal actions. A few observations based on simulations:

- Our initial specification models had to be adapted to include best-case and worst-case execution times. To capture uncertainties in the system behaviour, we added intermediate states to represent that the system might be in transition from one mode to another.
- Part of the comparison strategy is included in the specification model, to be able to use domain knowledge about processing delays and intermediate

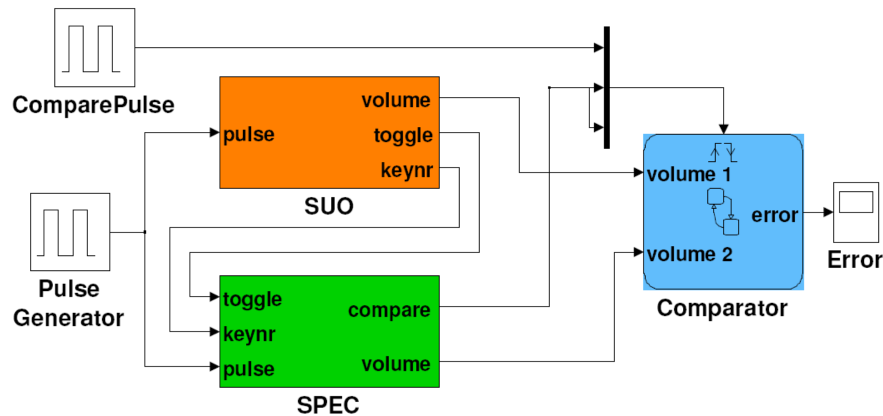

Fig. 3. Model of model-based error detection

states. To this end, the specification generates events to start and to stop the comparison (modeled by the "compare" signal in Figure 3).

– It became also clear that the comparator should not be too eager to report errors; small delays in system-internal communication might easily lead to differences during a short amount of time. Hence, current comparators only report an error if differences persist during a certain amount of time or occur a consecutive number of times. Observe that we have to make a trade-off between taking more time to avoid false errors and reporting errors fast to allow quick repair. This also influences the frequency with which we want to compare (modeled by the "ComparePulse" in Figure 3).

The design of the awareness framework is shown in Figure 4. The SUO and the awareness monitor are separate processes and Unix domain sockets are used for inter-process communication. The SUO has to be adapted slightly, to send messages with relevant input and output events (which may also include internal states) to Input and Output Observers. The Stateflow Coder of Simulink is used to generate C-code from a Stateflow model of the desired behaviour. This code is executed by the Model Executor, based on event notifications from the Input Observer. Information about relevant input and output events is stored in the Configuration component.

The Comparator compares relevant model output with system output which is obtained from the Output Observer. For each observable value, the user of the framework can specify

– a threshold for the allowed maximal deviation between specification model and system, and
– a limit for the number of consecutive deviations that are allowed before an error will be reported.

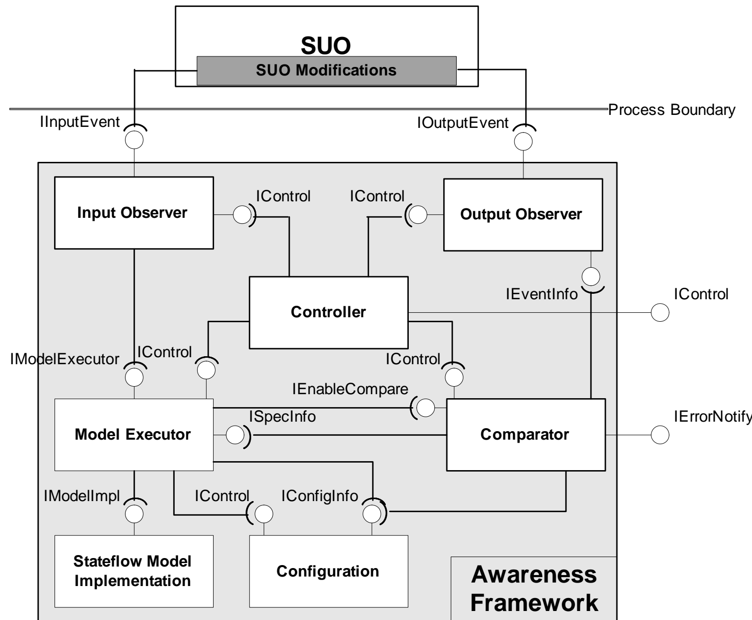

Fig. 4. Design of awareness framework in Linux

Another parameter is the frequency with which time-based comparison takes place. This can be combined with event-based comparison by specifying in the specification model when comparison should take place and when not (e.g., when the system is in an unstable state between certain modes). The Model Executor obtains this information from executing the implementation of the model and uses it to start and stop the Comparator. The Controller initiates and controls all components, except for the Configuration component which is controlled by the Model Executor.

5 Concluding remarks

Traditional fault-tolerance techniques such as Triple Modular Redundancy and N-version programming are not applicable in our application domain of high-volume products, because of the cost of the required redundancy. Related work that also takes cost limitations into account can be found in the research on fault-tolerance of large-scale embedded systems [5]. They apply the autonomic computing paradigm to systems with many processors to obtain a healing network. Similar to our approach is the use of a kind of controller-plant feedback loop, state machines, and simulation in Simulink/Stateflow.

Also related to our work are assertion-based approaches such as run-time verification [6]. For instance, monitor-oriented programming [7] supports run-time monitoring by integrating specifications in the program via logical annotations.

In our approach, we aim at minimal adaptation of the software of the system, to be able to deal with third-party software and legacy code. Moreover, we also monitor timing properties which are not addressed by most techniques described in the literature. Closely related in this respect is the MaC-RT system [8] which also detects timeliness violations. Main difference with our approach is the use of a timed version of Linear Temporal Logic to express requirements specifications, whereas we use executable timed state machines to promote industrial acceptance and validation.

To validate our Linux-based framework, we have experimented with model-to-model comparisons. That is, we have compared a specification model with code generated from models of the SUO. Currently, the framework is used for awareness experiments with the open source media player MPlayer [9], investigating both correctness and performance issues. In addition to error detection, this also includes connections with diagnosis [10] and recovery research in Trader. Next, the approach will be applied in the TV domain. Current activities in this domain focus on an evaluation of our techniques at development time, especially by using it for model-based testing.

Acknowledgments Many thanks goes to Chetan Nair for his work on the implementation of the awareness framework in Linux. The members of the Trader project are gratefully acknowledged for many fruitful discussions on reliability and the awareness concept.

References

1. Cronkhite, J.D.: Practical application of health and usage monitoring (HUMS) to helicopter rotor, engine, and drive systems. In: AHS, Proc. 49th Annual Forum. Volume 2. (1993) 1445–1455
2. Embedded Systems Institute: Trader project. (2007) http://www.esi.nl/trader/.
3. The Mathworks: Matlab/Simulink. (2007) http://www.mathworks.com/.
4. Reactive Systems: Model-Based Testing and Validation with Reactis. (2007) http://www.reactive-systems.com/.
5. Neema, S., Bapty, T., Shetty, S., Nordstrom, S.: Autonomic fault mitigation in embedded systems. Engineering Applications of Artificial Intelligence **17** (2004) 711–725
6. Colin, S., Mariani, L.: Run-time verification. In: Proceedings Model-Based Testing of Reactive Systems. Volume 3472 of LNCS., Springer-Verlag (2005) 525–555
7. Chen, F., D'Amorim, M., Rosu, G.: A formal monitoring-based framework for software development and analysis. In: Proceedings ICFEM 2004. Volume 3308 of LNCS., Springer-Verlag (2004) 357–372
8. Sammapun, U., Lee, I., Sokolsky, O.: Checking correctness at runtime using real-time Java. In: Proc. 3rd Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'05). (2005)
9. MPlayer: Open source media player. (2007) http://www.mplayerhq.hu/.
10. Zoeteweij, P., Abreu, R., Golsteijn, R., van Gemund, A.: Diagnosis of embedded software using program spectra. In: Proc. 14th Conference and Workshop on the Engineering of Computer Based Systems (ECBS'07). (2007) 213–220