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. The reliability of high-volume products, such as consumer electronic devices, is threatened by the combination of increasing complexity, decreasing time-to-market, and strong cost constraints. As an approach to maintain a high level of reliability and to avoid customer complaints, we present a run-time awareness concept. Part of this concept is the use of models for run-time error detection. We have implemented a general awareness framework 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. This method, coupled to local recovery techniques, aims to minimize any user exposure to product-internal technical errors, thereby improving user-perceived reliability.

To appear in: Proceedings MoDELS 2007 Workshops, Lecture Notes in Computer Science (LNCS), Vol. 5002, pp. 225-236, Springer-Verlag, 2008

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 provide 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 Semiconductors (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

* 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.

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, 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 of software in TVs has seen an exponential increase from 1 KB in 1980 to 24 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 Sect. 2 the main approach is described. We list the main research questions in Sect. 3. Section 4 contains current results. In Sect. 5 we discuss related work. Concluding remarks can be found in Sect. 6.

2 Approach

In observing failures of current products, it is often the case that a user can immediately observe that something is wrong, whereas 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. The concept of run-time awareness and correction as pursued by the Trader project is depicted in Fig. 1.

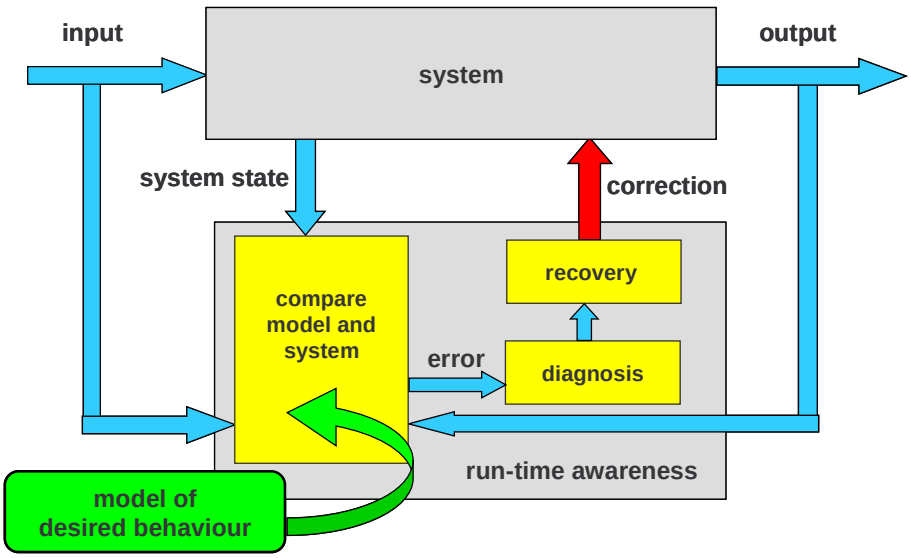

Fig. 1. Adding awareness at run time

We list the main ingredients of our run-time awareness approach, 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. These techniques can be applied at various levels of granularity, from fine-grained fault-localization in blocks of C-code to course-grained diagnoses of large components.

– *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.

The approach depicted in Fig. 1 can be applied to the complete system, but run-time awareness can also be added hierarchically and incrementally to parts of the system, e.g., to third-party components.

An important part of our approach is the use of models at run time. These models need not be complete descriptions of the full system behaviour; they could concentrate on a high-level abstract view of part of the system behaviour, depending on what is most useful in view of user-perceived reliability. To be able to detect and correct errors before the user notifies them, models will usually also describe certain aspects of internal system behaviour, such as the maximum load or memory usage in certain modes or crucial internal variables.

We briefly mention the current status of a number of research activities under the umbrella of the Trader project that contribute to run-time awareness:

– *Observation*: To observe relevant aspects of the system, hardware-related work in Trader currently aims at exploiting mechanisms already available in hardware, such as the on-chip debug and trace infrastructure. Software behaviour is observed by code instrumentation using aspect-oriented techniques, partly based on results from the ESI-project Ideals [2]. A specialized aspect-oriented framework called AspectKoala [3] has been developed on top of the component model Koala which is used at NXP to modularize the TV software.
– *Error detection*: Various techniques for error detection are investigated such as hardware-based deadlock detection and range checking. An approach which checks the consistency of internal modes of components turned out to be successful to detect teletext problems due to a loss of synchronization between components [4].
– *Diagnosis*: The diagnoses techniques developed within Trader are based on so-called program spectra [5]. The first applications in the TV domain are encouraging and the technique is currently refined by using it for debugging.
– *Recovery*: To allow independent recovery of parts of the system, a framework for local recovery has been developed [6]. A few first experiments in the multimedia domain show that, after some refactoring of the system, independent recovery of parts of the system is possible without large overhead. Another part of the recovery research concentrates on task migration. This includes, for instance, the migration of tasks from one processor to another to improve image quality in case of overload situations (e.g., due to intensive error correction on a bad input signal).

In addition, there are controlled experiments with TV users to capture user-perceived failure severity, that is, to get an indication of the level of user-irritation caused by a product failure. In the remainder of this paper we focus on model-based error detection, and refer to [7] for more information on other research within Trader.

3 Research Questions

We list a number of research questions concerning the embedding of error detection in concrete industrial products. In Sect. 3.1 we address the problem of getting suitable models. Section 3.2 discusses the use of these models for run-time error detection.

3.1 Modeling

There are several questions related to the models to be used at run time:

- Which part of the system has to be modeled? For a complex price-sensitive 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 the 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.
- How to obtain suitable models? Typically, in the area of embedded systems, the number of models available in industry is limited. The required overall system behaviour is usually not modeled and models have to be reconstructed using a lot of implicit domain knowledge (comparable to the experiences described Sect. 4 of [8]).
- How to increase the confidence in the model; how to evaluate model quality and fidelity?

3.2 Using Models at Run Time

The use of models at run time for error detection raises a number of questions:

- How to obtain the relevant observations from the system? Note that this involves both hardware and software parts of the system. Typically, there are several processors and state information is often distributed. Hence, a question is how to get a consistent snapshot of the global state of the system.
- 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. False errors might occur due to a number of reasons such as i) the use of an incorrect model, ii) an incorrect implementation of the model, iii) a comparison at a wrong moment in time when the system is not stable. This leads to the questions mentioned in the following points.
- 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. How to decide when the system is in a stable state? Is it possible to get notifications from the system or should stability be deduced indirectly, e.g., by observing return values or changing data structures? 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?

Observe that many of these questions are related. For instance, the decision what to model depends on the type of errors one wants to detect, which errors are recoverable, and what can be observed about the system in an effective way (without too much costs or performance loss). In our context, an important factor is also the user perception of which failures are irritating and which type of recovery is acceptable for users.

4 Results on Model-Based Error Detection

We present the current results of the Trader project on model-based error detection. First, in Sect. 4.1, we discuss work on obtaining a model of desired system behaviour, related to the questions in Sect. 3.1. Next, in Sect. 4.2, we present a framework for run-time model-based error detection to obtain more insight in the issues mentioned in Sect. 3.2.

4.1 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 [9]. The Stateflow toolbox of Simulink 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 Fig. 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 Fig. 2.

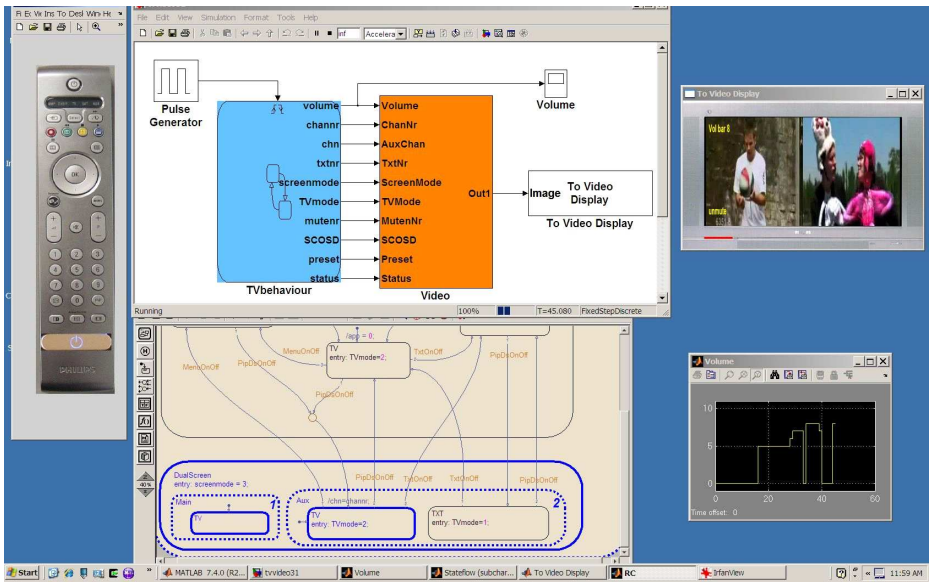

Fig. 2. Simulation of model of TV behaviour

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 [10] 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.

4.2 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

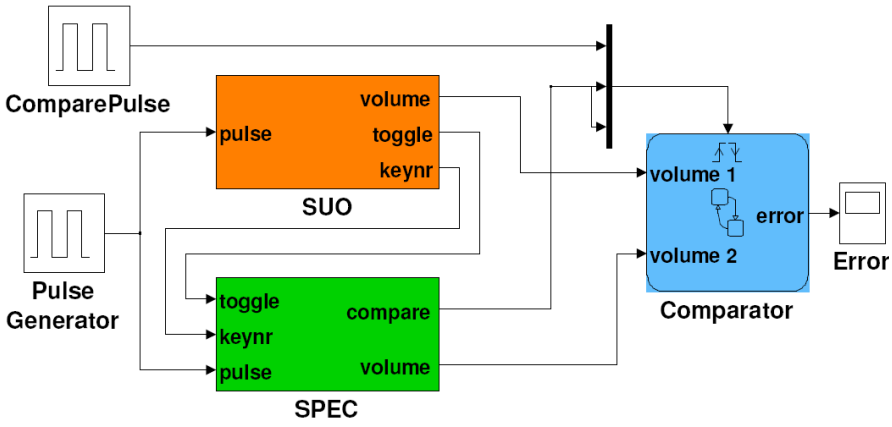

Fig. 3. Model of model-based error detection

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 are listed below:

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

– 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. A trade-off has to be made between taking more time to avoid false errors and reporting errors fast to allow quick repair. This also influences the frequency with which comparisons take place (modeled by the "ComparePulse" in Fig. 3).

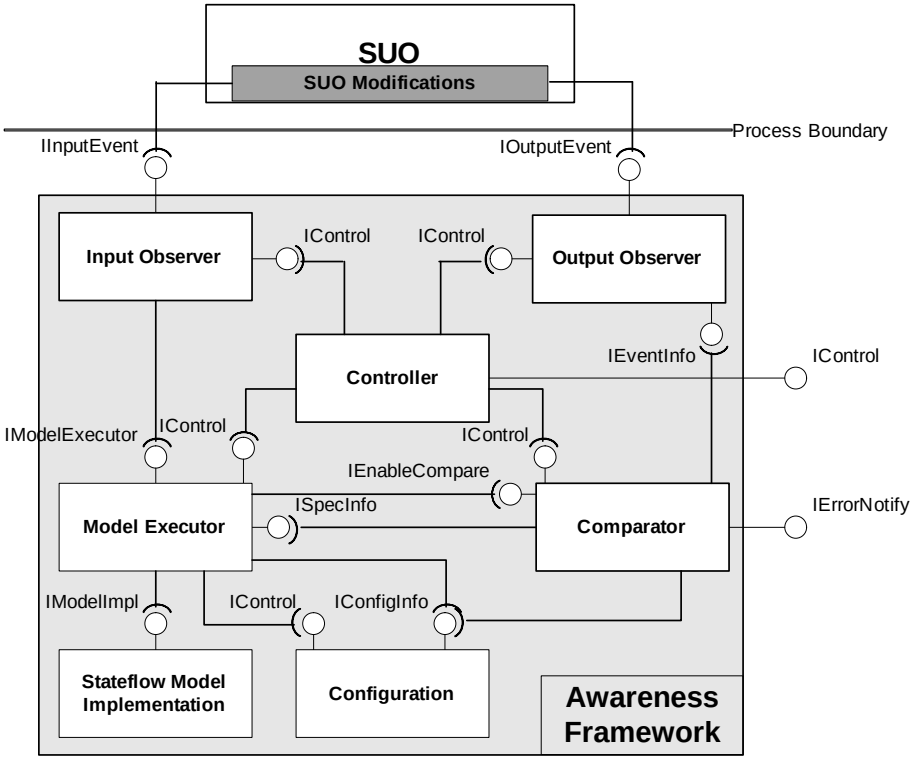

Fig. 4. Design of awareness framework in Linux

The design of the awareness framework is shown in Fig. 4. The SUO and the awareness monitor are separate processes communicating via Unix domain sockets. The SUO has to be adapted slightly, to send messages with relevant input and output (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 included in the Stateflow Model Implementation component and executed by the Model Executor. Based on event notifications from the Input Observer, the Model Executor provides in-

put to the code of the model. It also receives output from the model. Information about relevant input and output is stored in the Configuration component.

The Comparator compares model output with system output which is obtained from the Output Observer. For each observable value, the user of the framework can specify (1) a threshold for the allowed maximal deviation between specification model and system, and (2) a limit for the number of consecutive deviations that are allowed before an error will be reported. 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 Related Work

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 [11]. 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. Related work on adding a control loop to an existing system is described in the middleware approach of [12] where components are coupled via a publish-subscribe mechanism. A method to wrap COTS components and monitor them using specifications expressed as a UML state diagrams is presented in [13]. The analogy between self-controlling software and control theory has already been observed in [14]. Garlan et al [15] have developed an adaptation framework where system monitoring might invoke architectural changes. Using performance monitoring, this framework has been applied to the self-repair of web-based client-server systems.

Other related work consists of assertion-based approaches such as run-time verification [16]. For instance, monitor-oriented programming [17] 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 [18] 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.

Our approach to model-based error detection is also related to on-the-fly testing techniques which combine test generation and test execution [8,19]. The main difference is that these testing techniques generate input to the system based on the model, whereas we consider normal input during system operation and forward this input to the awareness component. Hence, our approach is more related to so-called passive testing. An additional difference is that testing methods concentrate on testing the input/output interface, whereas our focus is on fast error detection (preferably before output failures occur) which often leads to the monitoring of internal implementation details such as internal variables or load.

6 Concluding Remarks

Clearly, we have not yet answered all research questions mentioned in Sect. 3. Concerning the modeling questions of Sect. 3.1 we have mainly followed the well-known state machine approach to model the control behaviour of embedded systems. To increase both the industrial acceptance and the confidence in the correctness of the models, model execution and an intuitive visualization of input/output behaviour turned out to be essential. Convenient tool support has been obtained by using Matlab/Simulink/Stateflow which allows efficient code generation from models.

To investigate the questions in Sect. 3.2 about the use of models at run time, we developed a framework which allows quick experiments with run-time awareness. Currently, the framework is used for awareness experiments with the open source media player MPlayer [20]. It was easy to insert both the MPlayer and an abstract high-level Stateflow model of its desired behaviour in the framework, without degrading the performance of the MPlayer. Although some injected errors could be detected, more work is needed to investigate which types of errors can be detected, how false errors can be avoided, and how the approach scales to larger models. Moreover, we also intend to investigate the use of more architectural information in the model to detect errors earlier, before they affect the user-perceived behaviour.

Current work also includes connections with diagnosis and recovery techniques. The first results indicate that more research is needed to clarify the relation between the types of errors that can be detected and those that can be corrected by the local recovery techniques developed within Trader. Moreover, future work will address issues concerning the synchronization between the techniques, e.g., to avoid error detection during recovery and to ensure a re-synchronization between system and model after recovery.

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. We thank the anonymous reviewers for many useful comments and suggestions for improvement.

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: Ideals project. (2007) `http://www.esi.nl/ideals/`.
3. van de Laar, P., Golsteijn, R.: User-controlled reflection on join points. Journal of Software **2**(3) (2007) 1–8
4. Sözer, H., Hofmann, C., Tekinerdogan, B., Aksit, M.: Detecting mode inconsistencies in component-based embedded software. In: DSN Workshop on Architecting Dependable Systems. (2007)
5. 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
6. Sözer, H., Tekinerdogan, B.: Introducing recovery style for modeling and analyzing system recovery. In: Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA). (2008)
7. Embedded Systems Institute: Trader project. (2007) http://www.esi.nl/trader/.
8. Larsen, K., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: 5th ACM Int. Conf. on Embedded Software (EMSOFT'05), ACM Press New York (2005) 299 – 306
9. The Mathworks: Matlab/Simulink. (2007) http://www.mathworks.com/.
10. Reactive Systems: Model-Based Testing and Validation with Reactis. (2007) http://www.reactive-systems.com/.
11. Neema, S., Bapty, T., Shetty, S., Nordstrom, S.: Autonomic fault mitigation in embedded systems. Engineering Applications of Artificial Intelligence **17** (2004) 711–725
12. Parekh, J., Kaiser, G., Gross, P., Valetto, G.: Retrofitting autonomic capabilities onto legacy systems. Cluster Computing **9**(2) (2006) 141–159
13. Shin, M.E., Paniagua, F.: Self-management of COTS component-based systems using wrappers. In: Computer Software and Applications Conference (COMPSAC 2006), IEEE Computer Society (2006) 33–36
14. Kokar, M.M., Baclawski, K., Eracar, Y.A.: Control theory-based foundations of self-controlling software. IEEE Intelligent Software (1999) 37–45
15. Garlan, D., Cheng, S., Schmerl, B.: Increasing System Dependability through Architecture-based Selfrepair. In: Architecting Dependable Systems. Springer-Verlag (2003)
16. Colin, S., Mariani, L.: Run-time verification. In: Proc. Model-Based Testing of Reactive Systems. Volume 3472 of LNCS., Springer-Verlag (2005) 525–555
17. Chen, F., D'Amorim, M., Rosu, G.: A formal monitoring-based framework for software development and analysis. In: Proc. ICFEM 2004. Volume 3308 of LNCS., Springer-Verlag (2004) 357–372
18. 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)
19. van Weelden, A., Oostdijk, M., Frantzen, L., Koopman, P., Tretmans, J.: On-the-fly formal testing of a smart card applet. In: Int. Conf. on Information Security (SEC 2005). (2005) 565–576
20. MPlayer: Open source media player. (2007) http://www.mplayerhq.hu/.