Detecting Mode Inconsistencies in Component-Based Embedded Software*

Hasan Sozer, Christian Hofmann, Bedir Tekinerdogan, Mehmet Aksit
Department of Computer Science, University of Twente
P.O. Box 217 7500 AE Enschede, The Netherlands
{sozerh, hofmannc, bedir, aksit}@cs.utwente.nl

Abstract

To deal with increasing size and complexity, component-based software development has been employed in embedded systems. These systems comprise a set of components each of which implements a particular functionality. The system utilizes the components to provide the functionalities that are required in a set of working modes. Components can also be considered to have a set of working modes. They should work in harmony and consistent with the working mode of the system. Due to several errors that remain undetected during the design and implementation phases, components can make wrong assumptions about the working mode of the system and the working modes of the other components. These errors may lead to severe failures. Fault tolerance is required to prevent these failures at run-time. The first step to achieve fault tolerance is error detection. To detect mode inconsistencies at run-time, we propose a "lightweight" error detection mechanism, which can be integrated with component-based embedded systems. We define three dependent levels of abstractions: the run-time behavior of components, the working mode specifications of components and the specification of the working modes of the system. We define explicit links among these levels by specifying a mutual consistency condition. This allows us to detect the user observable run-time errors. The effectiveness of the approach is demonstrated by implementing a software monitor integrated into a TV system.

1. Introduction

Software technology is an integral part of today's consumer electronics. In fact most home equipments today such as television sets are full-fledged embedded systems (ES). An evident problem in the ES domain is the exponential growth of software size and complexity [7]. Since mid 1980s, the size of software in digital television sets (DTVs), for example, has doubled with every new DTV generation [20]. This is a consequence of the increasing number of new requirements and constantly changing hardware technologies. The designers on one hand have to deal with these complex demands; on the other hand they have to provide the desired qualities such as low cost, reliability and performance. Re-use and effective modularization are essential to deal with these challenges. Recently, component-based development has been recognized as a feasible approach for ES development, that helps to improve reuse and that eases the creation of variants of products [1, 8]. One advantage that componentization brings into the development process is the possibility of testing functionalities in isolation. If the system comprises an extensive number of components, however, a proper integration of components can be difficult [6].

We consider here ES like DTVs, where software is mainly developed using component technology. A considerable portion of the architecture is developed by third party companies, which deliver software conforming to a specific component model. Usually each component has to deliver a set of well defined services in a set of working modes. Components can correctly work together in the integrated system, only if their working modes are consistent with each other. A large set of faults, however, leads to mode inconsistencies at run-time. Similar to the mode confusion problem in Human-Computer interaction [13], wrong assumptions about the internal state of a component often lead to the execution of actions that conflict with the real state of the current system.

We observed that mode inconsistencies between components can cause severe errors that lead to user-perceived failures. To detect and recover such errors, dedicated fault tolerance mechanisms are required. Instead of tolerating faults, one may try to avoid them by adopting theorem prov-

*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 Netherlands Ministry of Economic Affairs under the Bsik program.

ing and model checking techniques at the design time. Although these techniques are proven to be valuable for many practical applications, the existing tools do not scale-up easily. Moreover, some faults may simply remain undetected and/or new faults may be introduced during the implementation. Since it is considered hard to remove all possible faults in a complex system, we propose an error detection mechanism that can detect mode inconsistencies regardless of the cause.

In our approach, we use an algorithm that verifies the mutual consistency of the interacting components according to the system-level and component-level working modes. We define three dependent levels of abstractions: the run-time behavior of the components, the working mode specifications of components and the specification of the working modes of the system. We establish explicit links among these levels by specifying a mutual consistency condition. This allows us to detect user observable run-time errors caused by inconsistent working modes. The effectiveness of the approach is demonstrated with a software monitor integrated into a TV system.

The remainder of this paper is organized as follows. Section 2 introduces the problem using an industrial case. The proposed solution approach is described in Section 3 together with a prototype implementation. Section 4 proposes diagnosis and recovery techniques for complementing the detection. Section 5 discusses the effectiveness and the limitations of the solution approach. In Section 6, related previous studies are summarized. Finally, some future work issues are discussed and the paper is concluded in section 7.

2. Industrial Case

In this section, we illustrate the problem using DTV as an industrial case. Although the problem is introduced in a specific context, it can be generalized to other component-based ES. The software of DTVs is composed of many components working in coordination. Software components and their interconnections are specified by the Koala Component Model [17]. Because of high diversity, both in features provided and hardware configurations used in different products, compatibility, reusability and configurability are required. The Koala Component Model was designed to provide an explicit description of the components and how they are connected to form a configuration. Each component has its own internal behavior and a set of working modes. These modes should be consistent with each other to provide the functionality that is required in a working mode of the system. If synchronization between the components is lost (by loosing a notification message, data corruption etc.) inconsistent behavior occurs and component interactions do no longer work in the anticipated way.

Consider for example the Teletext sub-system of a television. A component, the *Teletext Page Handler* is responsible for requesting a teletext page, display the status line during acquisition, and reveal it after it has arrived. Another component, the *Display Manager* is used to render teletext pages to the screen. Figure 1 shows the software component layout of a typical TV product. It can be seen that *Display Manager* and *Teletext Page Handler* are part of different top-level components that belong to different levels in the component hierarchy. Additionally, both components are developed by different teams at different development sites. To work correctly together, the assumptions about the current working mode of the TV, made by each component, should be correct and consistent.

Wrong assumptions can lead to errors, even though each component works perfectly given its local context. If, for example, the *Display Manager* correctly assumes that the TV is in Teletext mode whereas the *Teletext Page Handler* assumes that the TV should display the video stream, the combined behavior of both components leads to a user visible failure. No Teletext page is rendered leaving the user with a blank screen. Even though the fault has only a negligible impact on the run-time behavior of the system, it is unacceptable for a consumer electronics product.

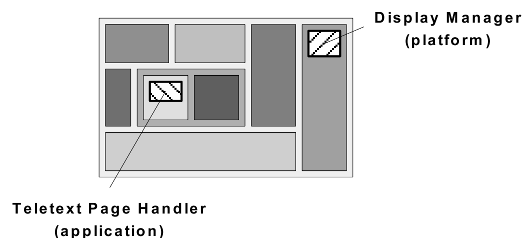

Figure 1. Illustrating the Problematic Components in the SW-Component Layout Notation

Detection of inconsistent modes is quite challenging. The large number of components and connectors makes determining which components should be checked in a given execution context, very complex. Another challenge is the cost sensitivity of the consumer electronics domain due to market pressure and high volume production. Because of complexity, resource usage and cost constraints, it is not feasible to check at system level whether the component modes are correct and consistent . We propose in the following an approach to detect mode inconsistencies at component level.

3. Error Detection

Fault tolerance requires the detection of the error(s) leading to a failure. Then the detected error(s) should be recovered, which may require the diagnosis of the cause(s). In

this work, we focus on error detection. We propose an error detection mechanism, for which we provide a prototype implementation. Incorporating diagnosis and recovery to our approach is discussed in Section 4.

To detect errors, we utilize the information regarding the working modes of the system and the component modes (See Figure 2). We map the models regarding the component working modes to the implementation of the corresponding component for observing the component modes at run-time. Note that this mapping is not even necessary if the modes are explicitly specified by component interfaces [9]. We also map all component modes to the system modes, which reveals inter-dependencies among them.

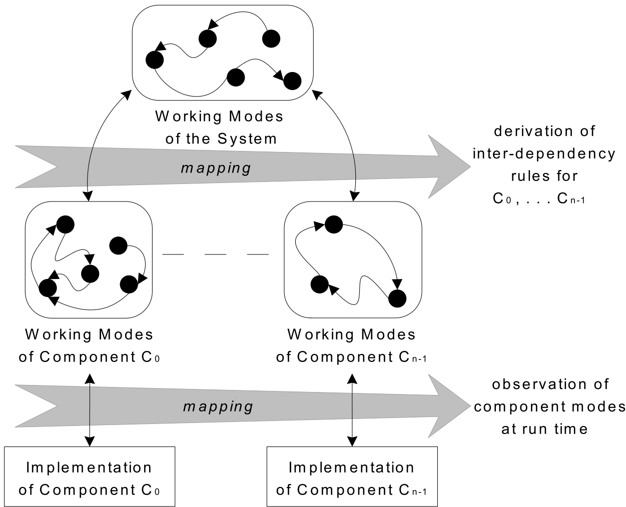

Figure 2. Mode Inconsistency Detection

The mappings between modes specify the mutual consistency condition, which is checked by monitoring the system at run-time to detect mode inconsistencies. This can be implemented as an observer that collects the current modes from the components of the system and checks the consistency condition. An error is issued whenever an inconsistency is detected. We present in Section 3.1, a prototype implementation of the approach. Section 3.2 generalizes this implementation and provides a formal definition of the mode consistency condition.

3.1. Implementation: A Prototype

We developed a prototype and integrated it into a real TV system to test the effectiveness of our solution approach. The prototype is an observer (i.e. monitor of the system) that checks the consistency between the working modes of the *Teletext Page Handler* and the *Display Manager* components we introduced in Section 2.

We modelled the working modes using the state machine formalism. The system working modes are represented by

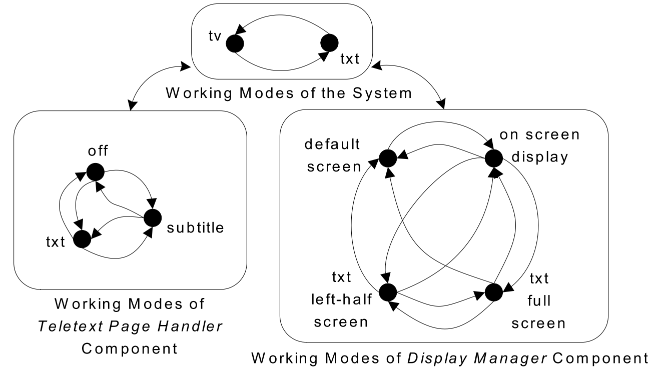

Figure 3. Error Detection Mechanism for Mode Consistency of Two Components

a simple state machine composed of two states: *tv* and *txt* (See Figure 3). *Txt* represents the mode in which the system is when the user is browsing Teletext. It is also possible to watch TV while browsing Teletext. This case is covered by the *txt* mode as well. If the user is not making use of the Teletext browsing functionality, then the system is in *tv* mode. We also constructed state machines to represent the modes of the *Teletext Page Handler* and *Display Manager* components. As can be seen in Figure 3, each of these components has different concerns[1] and its own set of modes. For the mapping of modes to the implementation, we were able to use the variables that hold the current mode information.We did not map the transitions but only the modes of the models. We represent modes of a component using a bit vector, where each bit corresponds to a mode of the system. In this case, each mode is represented by two bits. The value 1 is assigned to the first bit of a mode, if the corresponding component can be in that mode while the system is in *tv* mode. Similarly, the second bit is set to 1, if the component can be in that mode while the system is in *txt* mode (See Table 1).

Our error detection mechanism is designed such that it polls the system periodically. Error checking is performed simply by comparing the values of the mode variables to observe whether the components have a consensus on the system mode. In brief, the observer periodically takes snapshots of the running system and checks whether the mode information is consistent.

We injected four different faults, which lead to mode inconsistencies of the *Teletext Page Handler* and *Display Manager* components and eventually end up in lock-up failures in the Teletext functionality. These faults are systematically activated, and in all cases our detection mechanism was able to detect the errors before the associated failure was observed by the user. This is an important feature, be-

[1] Due to confidentiality, we present a representative set of modes.

Table 1. Mapping component modes to working modes of the system

Txt. Page Handler	map	Display Mgr.	map
off	10	*default screen*	10
txt	01	*on screen display*	10
subtitle	11	*txt left-half screen*	01
		txt full screen	01

cause it allows to recover from an error before a failure is perceived.

3.2. Generalization of the Prototype

We presented a prototype implementation of our solution approach for a specific case comprising two components. In this section, we generalize and formalize the way that our solution approach is realized in the prototype. The definitions and notation introduced in this section will be further used in the remainder of the paper.

We model the working modes using state machines, where each state corresponds to a mode. A state machine M can be specified as a tuple (S, E), where S is a finite set of modes and $E \subseteq S \times S$ is the set of transitions between these modes. We don't make here any assumptions about conditions or initial modes, since we only want to specify the mode consistency relationship. To distinguish between state machines that model the working modes of a specific component, we introduce a finite set of components C, where for each component $C_i \in C$, there exists a state machine $M_i = (S_i, E_i)$ that models its working modes. Furthermore there exists a state machine $M_G = (S_G, E_G)$ that models the working modes of the system. For each mode couple (s, g) *s.t.* $s \in S_i$ and $g \in S_G$, we define a mapping function,

$$map(s,g) = \begin{cases} 0, & g \Rightarrow A(\neg sW\neg g) \\ 1, & otherwise \end{cases}$$

where $g \Rightarrow A(\neg sW\neg g)$ is a formula in Computational Tree Logic (CTL) [3] denoting that when g occurs, s should never occur until the mode of the system changes. For every mode $s \in S_i$ of a component C_i, we define a bit-vector $v_{i,s}$ of length $|S_G|$, where each bit refers to a mode in S_G. Using the mapping function, we assign to every bit of the bit-vector $v_{i,s}$ a value that specifies if the mode s can occur, or should not occur, in a specific mode $g \in S_G$. The consistency condition is defined as follows.

$$\bigwedge_{0\leq i<|C|} v_{i,s_{current,i}}$$

There exists an error due to a mode inconsistency if the consistency condition is violated (i.e. the result is 0). The existence of an error means that there is no consistent assumption of each component about the current mode of the system. In a correctly functioning system, there exists at least one working mode, where the current mode $s_{current,i}$ of each component C_i can occur.

4. Diagnosis and Recovery

As mentioned in Section 3, error detection is the first necessary step for the prevention of failures. As another essential step, detected errors must be recovered. Backward recovery [12] is a generally applicable approach, in which the system state is set back to a previous state that is known to be correct. The cheapest backward recovery is to automatically reset the whole system. This action will put the component modes back to a consistent combination. Local-recovery is a more effective approach, in which the recovery procedure take actions concerning only the erroneous components. For instance, resetting can be applied to a set of components instead of the whole system (see "micro-reboot" in [5]). However, for this technique to be applied, the design of the system should permit re-initializing the components independently.

Algorithm 1 Diagnosis Procedure

```
1:  systemmode ← ∅
2:  maximumvotesum ← 0
3:  for 0 ≤ j < |S_G| do
4:     votesum ← 0
5:     for 0 ≤ i < |C| do
6:        votesum ← votesum + v_{i,s_{current,i}}[j]
7:     end for
8:     if maximumvotesum < votesum then
9:        systemmode ← j
10:       maximumvotesum ← votesum
11:    end if
12: end for
13: for i such that 0 ≤ i < |C| do
14:    if v_{i,s_{current,i}}[systemmode] ≠ 1 then
15:       mark the component C_i
16:    end if
17: end for
```

Additionally, to make local-recovery possible, a diagnosis step should be introduced beforehand, which localizes the error. In our case, the existence of an error means that the components do not have a consensus on the current system mode. We can apply a voting mechanism to pinpoint the problematic components in $O(|S_G| \times |C|)$ time as shown in Algorithm 1 (Recall the notation introduced in Section 3.2).

The algorithm keeps track of two variables; the most likely system mode and its vote, which are initialized as *null*

and 0, respectively (lines 1-2). The vote for each system mode is computed by summing up the values of the corresponding bits of the bit vectors of all components (lines 3-7). After each computation, the result is compared with the current maximum vote and the variables are updated if necessary (lines 8-12). As the last step, the algorithm iterates over all components and marks the ones that have incompatible modes with respect to the system mode voted by the majority (lines 13-17). The algorithm assumes that there is only one possible system mode, on which the majority of the components agrees. It might also be the case that there are multiple modes having the same maximum number of votes. A slight modification to the algorithm can handle those cases as well by storing a set of possible system modes instead of one. In the last step of the algorithm, the comparison would take place for each member of this set. The component would be marked if its mode is not compatible with any member of the set of modes.

5. Discussion

In the following, we discuss the design choices that we made for the implementation of the prototype. In the prototype, we ignored the transitions and we only mapped the modes. We represented each mode of a component with a bit vector where each bit corresponds to a mode of the system. The binary values indicate the compatibility of the modes. This representation was chosen for simplicity and efficiency; *Effort to map the modes:* Each bit has to be set to 0 or 1 based on mutual mode consistency. *Space complexity:* A space of size $(|C| \times |S_G|)$ bits must be allocated. *Time complexity:* Error checking is done with a single *AND* operation over the bit-vectors.

Monitoring at run time is problematic with respect to timing (i.e. when to collect the mode information). Furthermore, monitoring can intervene with the system functionality. To avoid these problems, we assigned the monitor to the lowest priority task, which can proceed after all other tasks become idle. This provides a safe point of time to perform the error checking since the system reaches to a stable state before the mode information is collected. Furthermore, no performance degradation is introduced. The monitor behaves as an external observer because it does not intervene with other tasks and functions. The only drawback is that the error detection might be late. Nevertheless, this did not emerge as a problem during test executions. All lock-ups were detected before they were observed by the user. This could be improved using an event-driven implementation. However, specifying a safe error checking point is not straightforward. False positives can be introduced if the error checking is done while there are still some concurrent actions in progress. We can put a timeout value to wait until when we expect the actions related to the mode changes are to be completed. If the timeout value is too short, false positives can occur. If it is too long, detection will be late. The exact duration may even change depending on the load of the system. Error checking may never have a chance to execute if the system is continuously busy or deadlocked. Such errors can be detected by other mechanisms like watchdog ([10]).

6. Related Work

Schroeder defines in [19] basic concepts of on-line monitoring and explains its characteristics. Accordingly, a monitoring system is an external observer that monitors a fully functioning application and is generally intended to be permanent. The error detection mechanism presented in this paper fits this definition. Based on the classification scheme provided, we can classify our approach as follows: The purpose of our system is reliability. Sensors are the mapping of the component modes to the implementation and they are always enabled. The event interpretation specification is built into the monitoring system. Sensing is based on sampling. Action specification is a recovery attempt, which can be achieved by partially or fully restarting the observed system and it is executed when an inconsistent mode combination is detected.

An outline of important features of real-time system monitoring and a classification is provided in [18]. Based on the two common design philosophies mentioned, our system can be considered as time-driven in the sense that the error checking is performed periodically. However, it can also be considered as event-driven since the monitoring system becomes active only during system idle time. So, there is an implicit event (i.e. system becomes idle), which triggers error checking. Fidge elaborates on observation of distributed systems in [4]. The focus of this work, however, is timing with the aim to determine the order of events.

In [18], the behavior of a real-time system is monitored based on its formal specification. In practice, it is hard to provide formal specifications for the total behavior of complex ES. Nevertheless, there have been several proposals regarding formal specification of behavior. For example, in [14] requirements are used to define a set of acceptable system behavior. A model based on Hoare Logic is proposed in [22] and [2] proposes the use of Petri Nets.

Behavior protocols for software components are proposed in [15]. Essentially, the basic approach is similar to the usage of contracts as proposed in [21]. Behavioral contracts are specified as regular expressions for both horizontal (client-service) and vertical (nesting) cooperation of components. They are utilized mainly for checking the compliance of the components at design-time but they can also be used for run-time monitoring. To enforce large scale synchronization, however, specifications must be defined

strictly for all components of the system. In contrast, our mechanism can be introduced to an existing system incrementally. The consistency between indirectly cooperating components can be checked without modifying the other components.

The scheme proposed by [11] checks state consistency for detecting errors. The decision about whether there exists an error or not is made statistically. The outcome is according to the ratio between checks that passed and the total number of checks executed. In our case, we have a deterministic approach, in which an error is issued whenever a check does not pass.

Basically, our work can be considered to be aligned with the approach defined as "acceptability-oriented computing" in [16]. According to this approach, the designer identifies key properties that the execution must satisfy to be acceptable to its users. Instead of developing a perfect program, the idea is to integrate mechanisms to the program that keeps it satisfy the acceptability properties. In this work we aim at detecting errors that end up with severe failures. *partially* defined properties are monitored at run-time. These properties identify crucial requirements for a failure-free execution.

7. Conclusion and Future Work

In this paper, we pointed out a problem associated with component-based software that constitutes a serious challenge for reliability of ES. Either because of implicit assumptions at the design level or faults introduced during the implementation, mode inconsistencies can occur, which end up with the failure of the system. Our aim was to detect and recover such errors independent of the fauls. Accordingly, we proposed a error detection mechanism that can detect mode inconsistencies at run-time. It can be utilized independent of the component technology. We implemented a prototype of our solution and integrated it into a TV system. Together with useful insights into the design of such a system, we obtained initial results that are promising.

As a future work, we would like to enhance the prototype by utilizing an extensive number of modes of the system and several components. In addition, we are planning to incorporate diagnosis and recovery mechanisms as discussed to test their effectiveness.

Acknowledgments

We thank all members of the TRADER project. In particular we thank Rob Golsteijn from NXP Semiconductors, Paul L. Janson from Philips Research, Pierre van de Laar from ESI for their help during the implementation of the prototype. We also thank Jozef Hooman and Teun Hendriks from ESI, Ben Pronk from NXP Semiconductors and the anonymous reviewers for reviewing and providing useful feedback to this paper.

References

[1] I. Crnkovic. Component-based approach for embedded systems. In *WCOP*, Oslo, Norway, 2004.

[2] M. Diaz, G. Juanole, and J. Courtiat. Observer - a concept for formal on-line validation of distributed systems. *IEEE Trans. Software Eng.*, 20(12):900–913, 1994.

[3] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[4] C. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

[5] G. Candea et al. Microreboot: A technique for cheap recovery. In *OSDI*, San Francisco, CA, 2004.

[6] D. Garlan, R. Allen, and J. OckerBloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE'95*, 1995.

[7] M. Grimheden and M. Torngren. What is embedded systems and how should it be taught. *TECS*, 4(3):633–651, 2005.

[8] A. Guerrouat and H. Richter. A component-based specification approach for embedded systems using FDTs. *SIGSOFT Software Engineering Notes*, 31(2):14, 2006.

[9] H. Hansson et al. SaveCCM - a component model for safety-critical real-time systems. In *EUROMICRO*, pages 627–635, Washington, DC, 2004.

[10] Y. Huang and C. Kintala. Software fault tolerance in the application layer. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 10, pages 231–248. John Wiley & Sons, 1995.

[11] J. Thai et al. Detection of errors using aspect-oriented state consistency checks. In *ISSRE*, pages 29–30, 2001.

[12] M. Elnozahy et al. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, 1996.

[13] Nancy G. Leveson et al. Analyzing software specifications for mode confusion potential. In *Workshop on Human Error and System Development*, pages 132–146, 1997.

[14] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Software Eng.*, 28(2):146–158, 2002.

[15] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11), 2002.

[16] M. Rinard. Acceptability-oriented computing. *SIGPLAN Not.*, 38(12):57–75, 2003.

[17] Rob C. van Ommering et al. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[18] U. Schmid. Monitoring distributed real-time systems. *Real-Time Systems*, 7(1):33–56, July 1994.

[19] B. A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, pages 72–78, January 1995.

[20] Trader project, ESI, 2007. http://www.esi.nl.

[21] Y. Berbers et al. CoConES: An approach for components and contracts in embedded systems. *LNCS*, 3778:209–231, 2005.

[22] M. Zulkernine and R. Seviora. Towards automatic monitoring of component-based software systems. *JSS ACBSE Special Issue*, 74(1):15–24, January 2005.