Runtime Verification of Component-Based Embedded Software

Hasan Sözer, Christian Hofmann, Bedir Tekinerdoğan and Mehmet Akşit

Abstract To deal with increasing size and complexity, component-based software development has been employed in embedded systems. Due to several faults, components can make wrong assumptions about the working mode of the system and the working modes of the other components. To detect mode inconsistencies at runtime, we propose a "lightweight" error detection mechanism, which can be integrated with component-based embedded systems. We define links among three levels of abstractions: the runtime behavior of components, the working mode specifications of components and the specification of the working modes of the system. This allows us to detect the user observable runtime errors. The effectiveness of the approach is demonstrated by implementing a software monitor integrated into a TV system.

1 Introduction

An evident problem in the embedded systems (ES) domain is the increasing software size and complexity. As a solution, component-based development has been recognized as a feasible approach to improve reuse and to ease the creation of

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.

H. Sözer (✉)
Özyeğin University, Istanbul, Turkey
e-mail: hasan.sozer@ozyegin.edu.tr

C. Hofmann · M. Akşit
University of Twente, Enschede, The Netherlands

B. Tekinerdoğan
Bilkent University, Ankara, Turkey

E. Gelenbe et al. (eds.), *Computer and Information Sciences II*,
DOI: 10.1007/978-1-4471-2155-8_60,

variants of products [1]. Hereby, 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; however, several faults can lead to mode inconsistencies at runtime.

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 proving and model checking techniques at design time. Although these techniques have showed their value for many practical applications, the existing tools do not scale-up easily. Moreover, some faults may simply remain undetected during design and/or new faults may be introduced during the implementation.

In our approach, we define 3 levels of abstractions: the runtime 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. This allows us to detect runtime errors caused by inconsistent working modes of components. 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. Our solution approach is described in Sect. 3. Section 4 proposes diagnosis and recovery techniques. Section 5 discusses the effectiveness and the limitations of the solution approach. In Sect. 6, related previous studies are summarized. Finally, the paper is concluded in Sect. 7.

2 Industrial Case

In this section, we illustrate the problem using digital TV (DTV) as an industrial case. The DTV software is composed of many components working in coordination [1]. Each component has a set of working modes. These modes should be mutually consistent to provide the functionality that is required by a working mode of the system. If the synchronization between the component's working modes is lost (by loosing a notification message, data corruption etc.) inconsistent behavior occurs and component interactions no longer work in the anticipated way.

Consider for example the *Teletext Page Handler* component, which is responsible for requesting and acquiring a teletext page. Another component, *Display Manager* renders teletext pages on the screen. If *Display Manager* correctly assumes that the TV is in Teletext mode whereas *Teletext Page Handler* assumes that the TV should display the video stream, the combined behavior leads to a failure: no Teletext page is rendered leaving the user with a blank screen.

Due to the large number of components and the cost sensitivity of the ES domain, it is not feasible to check mode consistencies at the system level. Therefore, we propose to detect mode inconsistencies at component level as follows.

Table 1 Mapping between the component modes and the 4 system working modes

Application manager		Txt page handler		Display manager	
Mode	Map	Mode	Map	Mode	Map
On screen display	1000	On	111	On screen display	1000
Txt	0111	Off	1000	Txt full screen	0100
Off	0000	subtitle	0111	Txt left-half screen	0010
TV	1000			Default screen	1000

3 Error Detection

Our approach is based on models of the working modes of the system and its components. We map the models describing the component working modes to the implementation of the corresponding component in order to observe the component modes at runtime. We also map all component modes to the system modes. This makes the inter-dependencies between component modes explicit. The mappings between modes specify the mutual consistency condition, which is checked by monitoring the system at runtime. An error is detected whenever an inconsistency has been observed. In the following, we will discuss a prototype implementation of this approach in more detail. Then, we generalize this implementation and provide a formal definition of the mode consistency condition.

3.1 Implementation: A Prototype

The number of errors that can be detected by our approach depends on the number of working modes of the system that is considered and the number of component modes that are monitored. We developed a prototype and integrated it into a real TV system, where 4 system working modes are considered and 3 components are monitored. System working modes are *TV*, *Txt*, *dual screen*, and *transparent teletext*. The monitored components are *Teletext Page Handler*, *Display Manager*, and *Application Manager*. The *Application Manager* component controls the execution of applications in the system. Each component mode is represented by a bit vector, $\langle b_0 b_1 b_2 b_3 \rangle$, where each bit corresponds to a system working mode, e.g., b_2 corresponds to *dual screen*. The mapping between the system working modes and the component modes is shown in Table 1.

Application Manager, *Teletext Page Handler* and *Display Manager* components can be in one of 4, 3 and 4 different modes, respectively. In total, there are $3 \times 4 \times 4 = 48$ different mode combinations. `AND` ing the bit vector representations of the component modes leads to the value 0 in 40 of the cases. In 8 cases the result is non-zero, i.e., there is at least one bit where all of the component modes have the value 1. Therefore, our prototype can detect 40 error cases. The error detection mechanism polls the system periodically, where the current state values of the components are `AND`ed and an error is issued if the result is 0.

We injected 4 different faults in the TV system. These lead to mode inconsistencies between the *Teletext Page Handler* and *Display Manager* components, which eventually end up in lock-up failures in the Teletext functionality. We systematically activated the faults with a key combination from the remote control. In all cases the detection mechanism was able to detect the errors (notified by blinking the TV status LED) even before we observed the associated failure. This allows to (possibly) recover from an error before a failure is perceived by the user.

3.2 Generalization of the Prototype

In this section, we generalize and formalize our approach. We define a finite set of components C, where for each component $C_i \in C$, there exists a set of working modes M_i. Furthermore, there exists a set of working modes of the system M_S. For each mode couple *(m,s) s.t.* $m \in M_i$ and $s \in M_S$, we define a mapping function,

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

where $s \Rightarrow A(\neg mW \neg s)$ is a Computational Tree Logic [2] formula denoting that when s occurs, m should never occur until the mode of the system changes. For every mode $m \in M_i$ of a component C_i, we define a bit-vector $v_{i,m}$ of length $|M_S|$. Each bit in $v_{i,m}$ refers to a mode s in M_S and its value is assigned according to the mapping function. The consistency condition is defined as $\bigwedge_{0 \leq i < |C|} v_{i,m}$, where $v_{i,m}$ corresponds to the current mode of component C_i. There exists an error if this condition is evaluated to 0, meaning that there is no consistent assumption of each component about the current mode of the system.

3.3 Performance Overhead

To measure the performance overhead introduced by the error detection mechanism, we used an existing feature of the system that measures the load in terms of CPU cycles. We have made load measurements during two scenarios; watching TV (*TV*) and reading a Teletext page (*TXT*). For each scenario, we calculated the maximum, minimum and average CPU load of the system with and without the error detection mechanism. The results are presented in Table 2.

In Table 2, we see that the CPU load during the *TV* scenario did not differ at all. For the *TXT* scenario, the average CPU loads was increased by 1,2% on average. This shows that the overhead introduced by the error detection mechanism can vary depending on the usage scenario. In the case of the *TV* and *TXT* scenarios, the overhead is at acceptable levels. In general, our approach provides several advantages in terms of simplicity and efficiency. Error checking is performed with a single `AND` operation over bit-vectors and a space of size ($|C| \times |S_G|$) bits must

Table 2 CPU load of the system with and without error detection

Scenario	Without error detection			With error detection		
	Min.(%)	Avg.(%)	Max.(%)	Min.(%)	Avg.(%)	Max.(%)
TV	34	36.9	51	34	36.9	51
TXT	39	42.9	46	41	43.4	47

be allocated only. Also note that the modeling effort is limited to assigning binary values that indicate the mode compatibility.

4 Diagnosis and Recovery

Error detection is the main focus in this paper. Another essential step of fault tolerance is recovering from the detected errors. We have recently developed a local recovery framework [3] for this purpose. Local recovery is an effective approach, in which the recovery procedure takes actions concerning only the erroneous components. To make local recovery possible, an additional diagnosis step should be introduced, which identifies the components that do not have a consensus on the current system mode. We can apply a voting mechanism to pinpoint such components in $O(|S_G| \times |C|)$ time as shown in Algorithm 4.

Algorithm 1 Diagnosis Procedure

1. $systemmode \leftarrow \emptyset$
2. $maximumvotesum \leftarrow 0$
3. **for** $j = 0 \rightarrow |S_G|$ **do**
4. $votesum \leftarrow 0$
5. **for** $i = 0 \rightarrow |C|$ **do**
6. $votesum \leftarrow votesum + v_{i,m_{current}}[j]$
7. **end for**
8. **if** $maximumvotesum < votesum$ **then**
9. $systemmode \leftarrow j$
10. $maximumvotesum \leftarrow votesum$
11. **end if**
12. **end for**
13. **for** $i = 0 \rightarrow |C|$ **do**
14. **if** $v_{i,m_{current}}[systemmode] \neq 1$ **then**
15. mark the component C_i
16. **end if**
17. **end for**

5 Discussion

We assigned the monitor to the lowest priority task, which can proceed after all the other tasks become idle. This provides a safe point in time to perform the error checking: (i) the monitor does not intervene with other tasks and functions, (ii) the system reaches to a stable state before the mode information is collected, and (iii) the introduced performance degradation is negligible. The only drawback of this approach is that error detection might be late. Error checking may never have a chance to execute in case the system is continuously busy or deadlocked. Such errors can be detected by other mechanisms like watchdog [4].

6 Related Work

There have been several proposals regarding formal specification of behavior [5–7]. Behavior protocols [8] and contracts [9] have been mainly used to formalize component interaction and utilized for design-time verification.

The scheme proposed by Thai et al. in [10] detects errors by checking consistency of states. 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. Our approach is deterministic in the sense that an error is issued whenever a check does not pass.

Classification schemes have been provided for on-line monitoring [11] and real-time system monitoring [12]. We can classify our work as a monitoring approach for *fault tolerance*. It is based on *time-driven* sampling of component modes and *built-in event interpretation* that triggers *recovery* actions.

7 Conclusion

We pointed out a problem associated with component-based software that constitutes a challenge for reliability of ES. Either because of implicit assumptions at the design level or faults introduced during the implementation, mode inconsistencies can occur between components, which end up with the failure of the system. Such errors can be detected and recovered, (possibly) before a failure is observed. In this paper, we proposed an error detection mechanism that can detect mode inconsistencies at runtime. It can be adopted independent of the utilized component technology. We implemented a prototype of our solution and integrated it into a TV system. We obtained promising results.

References

1. van Ommering, R.C., et al.: The Koala component model for consumer electronics software. IEEE Comput. **33**(3), 78–85 (2000)
2. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982)
3. Sozer, H., Tekinerdogan, B., Aksit, M.: FLORA: a framework for decomposing software architecture to introduce local recovery. Softw. Pract. Exper. **39**(10), 869–889 (2009)
4. Huang, Y., Kintala, C.: Software fault tolerance in the application layer. In: Lyu, M.R. (ed.) Software Fault Tolerance, pp.231–248. John Wiley & Sons, Chichester (1995)
5. Peters, D.K., Parnas, D.L.: Requirements-based monitors for real-time systems. IEEE Trans. Softw. Eng. **28**(2), 146–158 (2002)
6. Zulkernine, M., Seviora, R.: Towards automatic monitoring of component-based software systems. JSS ACBSE Special Issue **74**(1), 15–24 (2005)
7. Diaz, M., Juanole, G., Courtiat, J.: Observer—a concept for formal on-line validation of distributed systems. IEEE Trans. Softw. Eng. **20**(12), 900–913 (1994)
8. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11), 1056–1076 (2002)
9. Berbers, Y. et al.: CoConES: an approach for components and contracts in embedded systems. LNCS **3778**, 209–231 (2005)
10. Thai, J., et al.: Detection of errors using aspect-oriented state consistency checks. In: ISSRE, pp. 29–30 (2001)
11. Schroeder B.: On-line monitoring: a tutorial. IEEE Comput. **46**(25), 72–78 (1995)
12. Schmid, U.: Monitoring distributed real-time systems. Real-Time Syst. **7**(1), 33–56 (1994)