SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:869–889
Published online 20 January 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.916

FLORA: a framework for decomposing software architecture to introduce local recovery

Hasan Sözer[1,*,†], Bedir Tekinerdoğan[2] and Mehmet Akşit[1]

[1]*Department of Computer Science, University of Twente, 7500 AE Enschede, The Netherlands*
[2]*Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey*

SUMMARY

The decomposition of software architecture into modular units is usually driven by the required quality concerns. In this paper we focus on the impact of local recovery concern on the decomposition of the software system. For achieving local recovery, the system needs to be decomposed into separate units that can be recovered in isolation. However, it appears that this required decomposition for recovery is usually not aligned with the decomposition based on functional concerns. Moreover, introducing local recovery to a software system, while preserving the existing decomposition, is not trivial and requires substantial development and maintenance effort. To reduce this effort we propose a framework that supports the decomposition and implementation of software architecture for local recovery. The framework provides reusable abstractions for defining recoverable units and the necessary coordination and communication protocols for recovery. We discuss our experiences in the application and evaluation of the framework for introducing local recovery to the open-source media player called MPlayer. Copyright © 2009 John Wiley & Sons, Ltd.

Received 4 June 2008; Revised 20 November 2008; Accepted 5 December 2008

KEY WORDS: local recovery; software architecture; availability; fault-tolerance

1. INTRODUCTION

One of the key principles in software architecture design is modularity that aims to decompose the system into separate, modular units [1]. The decomposition of a system into modules is usually

*Correspondence to: Hasan Sözer, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
†E-mail: sozerh@cs.utwente.nl, sozerh@ewi.utwente.nl

driven by the required quality concerns such as adaptability, reuse, and performance. In this paper we focus on the impact of recovery concern on the decomposition of the software system.

The context of this research is from the Television Related Architecture and Design to Enhance Reliability (TRADER) project [2], which is carried out together with NXP Semiconductors and several other academic and industrial partners. One of the key objectives of the project is to develop techniques for analyzing recovery at the architecture design level for digital TVs (DTVs). Recovery can be applied at different levels of granularity in the system. In case of *global recovery*, the system is recovered as a whole when errors are detected. For example, in case of a deadlock, restarting the whole system makes it completely unavailable until the system is in its normal operational mode again. This lack of availability can be avoided by applying *local recovery* in which only the erroneous parts of the system are recovered. To recover from a deadlock, for instance, only the modules that are involved in the deadlock need to be restarted, while the other parts can remain available. Local recovery has an additional benefit because it also decreases the mean time to recover [3]. Hence, for better availability and faster recovery, it is necessary to reduce the granularity of the parts in the system that can be recovered and as such realize local recovery. For achieving local recovery, the corresponding system needs to be separated into a set of isolated *recoverable units* (*RUs*) so that the propagation of errors can be prevented. However, it appears that this required decomposition for recovery is usually not aligned with the decomposition based on functional concerns.

To resolve this issue we can redesign the architecture and define the decomposition of the modules solely based on the recovery concern. Usually, this is not the desired approach, because the existing decomposition based on functional concerns supports other important quality concerns such as adaptability, reuse and extendibility. Another alternative for introducing local recovery is to preserve the existing modular structure but adapt the existing modules and introduce new modules. This alternative, however, is not trivial and requires a substantial maintenance effort. This is mainly because the interactions of a module with all the other parts of the system need to be captured and appropriately handled during recovery. The recovery actions need to be coordinated with the normal operation of the system at run-time, and new architectural elements and complex interactions need to be introduced for communication control and recovery coordination [3–5].

We propose the framework FLORA for supporting the decomposition and implementation of software architecture for local recovery. Using the framework we can preserve the existing decomposition while reducing the effort for introducing local recovery. The framework includes a set of reusable abstractions for defining RUs and introducing error detection, diagnosis, communication control between RUs. The framework can be extended for defining customized recovery properties. We discuss our experiences in the application and evaluation of the framework by introducing local recovery to an open-source media player called MPlayer [6].

The remainder of this paper is organized as follows. Section 2 outlines the requirements for local recovery. In Section 3, we discuss the impact of local recovery on the architectural decomposition using a case study. In Section 4, we introduce FLORA and its main elements. In Section 5, we illustrate its application. In Section 6, we evaluate the improvement of system availability that is achieved by decomposing the software architecture and introducing local recovery with FLORA. We discuss the applicability of FLORA in Section 7. In Section 8, we discuss the limitations and possible extensions of the approach. We provide a summary of previous related studies in Section 9 and the conclusions in Section 10.

DOI: 10.1002/spe

2. REQUIREMENTS FOR LOCAL RECOVERY

Recovery from errors is an essential step to achieve fault tolerance for reliability [7]. Introducing local recovery to a system imposes certain requirements to its architecture. Based on the literature and our experiences in the project we have identified the following three basic requirements:

- *Isolation*: An error occurring in one part of the system can easily propagate and lead to errors in other parts. To prevent this error propagation and support local recovery we need to be able to decompose the system into a set of units that can be isolated. We call each such unit an *RU*. Isolation is usually supported by either the operating system (e.g. process isolation [8]) or a middleware (e.g. encapsulation of Enterprise Java Bean objects) and the existing design (i.e. *crash-only design*) [9].
- *Communication control*: Although an RU is unavailable during its recovery, the other RUs might still need to access it in the mean time. Therefore, the communication between RUs must be captured to deal with the unavailability of RUs, for example, by queuing and retrying messages or by generating exceptions. In [3], for instance, the communication is mediated by an application server. In general, various alternatives can be considered for realizing the communication control like completely distributed, hierarchical or centralized approaches.
- *System-recovery coordination*: In case recovery actions need to take place while the system is still operational, interference with the normal system functions can occur. For this reason, the required recovery actions need to be coordinated. Similar to communication control, coordination can also be realized in different ways ranging from completely distributed to completely centralized solutions.

3. DECOMPOSITION FOR LOCAL RECOVERY

In this section we will discuss the adaptation of an architecture for local recovery. We will illustrate this using an example case, MPlayer [6]. The reason for selecting MPlayer is that it resembles DTV software with respect to its code size, the adopted components and the audio and video streaming. MPlayer is a media player, which supports many input formats, codecs and output drivers. It embodies approximately 700K lines of code (LOC) and it is available under the GNU General Public License. In our case study, we have used version v1.0rc1 of this software that is compiled on Linux Platform (Ubuntu version 7.04). Figure 1 presents a simplified view of the MPlayer software architecture with basic implementation units and direct dependencies among them. In the following, we briefly explain the important modules, which are shown in this view.

Stream reads the input media by bytes/blocks and provides buffering, seek and skip functions. *Demuxer* demultiplexers (separates) the input into audio and video channels. *Mplayer* connects the other modules, and maintains the synchronization of audio and video. *Libmpcodecs* embodies the set of available codecs. *Libvo* displays video frames. *Libao* controls the playing of audio. *Gui* provides the graphical user interface of MPlayer. We have derived the main modules of MPlayer from its package structure. For example, the source files that are related to the graphical user interface are collected under the folder ‘./Gui’.

To achieve local recovery, we need to fulfill the requirements that were mentioned in Section 2. First of all, we need to decompose the system into a set of RUs to support isolation. One possible

DOI: 10.1002/spe

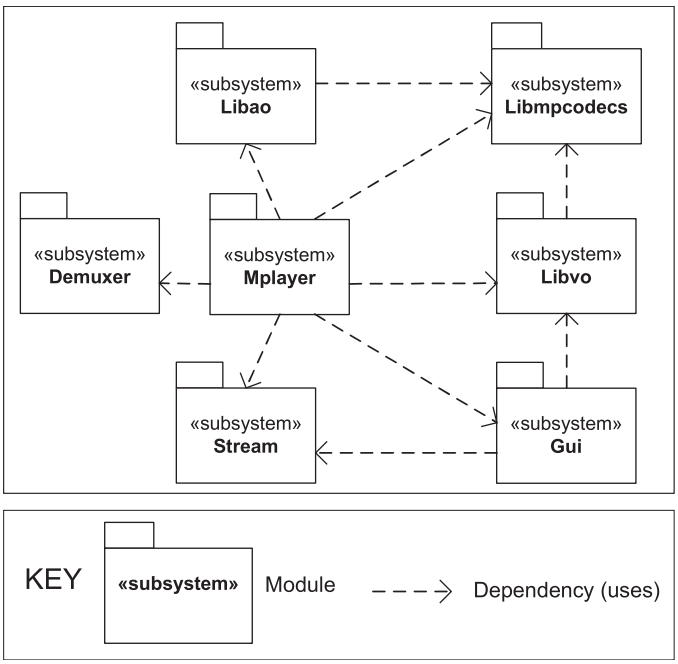

Figure 1. A simplified view of the MPlayer software architecture.

decomposition for the MPlayer case is to partition the system modules into 3 RUs: (1) *RU AUDIO*, which provides the functionality of *Libao* (2) *RU GUI*, which encapsulates the *Gui* functionality and (3) *RU MPCORE*, which comprises the rest of the system. Figure 2 depicts the boundaries of these RUs, which are overlayed on the MPlayer software architecture shown in Figure 1.

As we can see in Figure 2, introducing local recovery requires a different decomposition that is not aligned with the decomposition of the system based on functional concerns. The original architecture as displayed in Figure 1 separates the different modules based on functional concerns and as such supports quality concerns such as adaptability, reuse and extensibility. Obviously, we do not wish to break the modular structure and as such impede these quality concerns. We could therefore decide to keep the original structure and adapt the modules and if necessary introduce new modules to introduce local recovery. Unfortunately, this approach is also not viable because it requires a substantial development and maintenance effort. We need to meet the second and third requirements that are mentioned in Section 2. That is, the interactions of a module (in the original architecture) with all the other parts of the system need to be captured and appropriately handled during recovery. Accordingly, new architectural elements and complex interactions need to be introduced for communication control and system-recovery coordination.

We have also experienced these problems in our industrial research project [2]. Traditionally the recovery of TVs is based on restarting either the complete system or a large part of the system. However, current trends show that the size and complexity of software in future releases of TVs will increase dramatically and global recovery techniques will be less effective with respect to the desired availability and the mean time to recover. The size of software in TVs is doubled in

DOI: 10.1002/spe

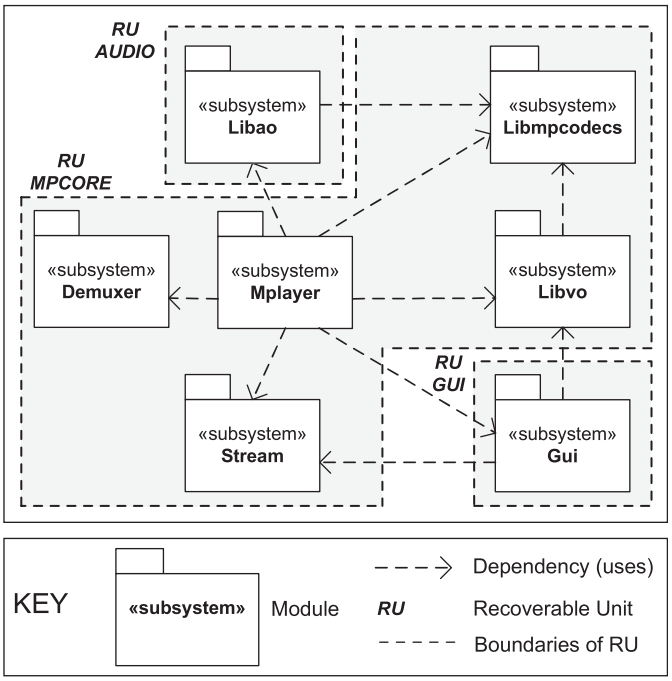

Figure 2. MPlayer software architecture with the boundaries of the recoverable units.

every product generation. Currently, a TV includes more than 1 million LOC and it takes several seconds to restart the system. Therefore, it is necessary to reduce the granularity of the parts in the system that can be recovered and as such realize local recovery. However, realizing local recovery without changing the modular structure of the original architecture is not trivial and requires lots of maintenance effort.

4. FLORA: A FRAMEWORK FOR LOCAL RECOVERY

To reduce the development and maintenance effort for introducing local recovery while preserving the existing decomposition we have developed the framework FLORA. The framework includes a set of reusable abstractions for introducing error detection, diagnosis, communication control between RUs. The framework can also be extended for defining customized recovery properties. Figure 3 shows a conceptual view of FLORA as a UML class diagram. The framework comprises three main components: *RU*, *Connector* and *Recovery Manager*.

Each *RU* uses the *Connector* to communicate with other *RU*s. The *Connector* mediates all inter-*RU* communication and employs a set of communication policies (e.g. drop, queue, retry messages). Note that a communication policy can be composed of a combination of other primitive or composite policies. The *Recovery Manager* uses the *Connector* to apply these policies based on the executed recovery strategy. The *Recovery Manager* also uses *RU*s to control them (e.g. kill,

DOI: 10.1002/spe

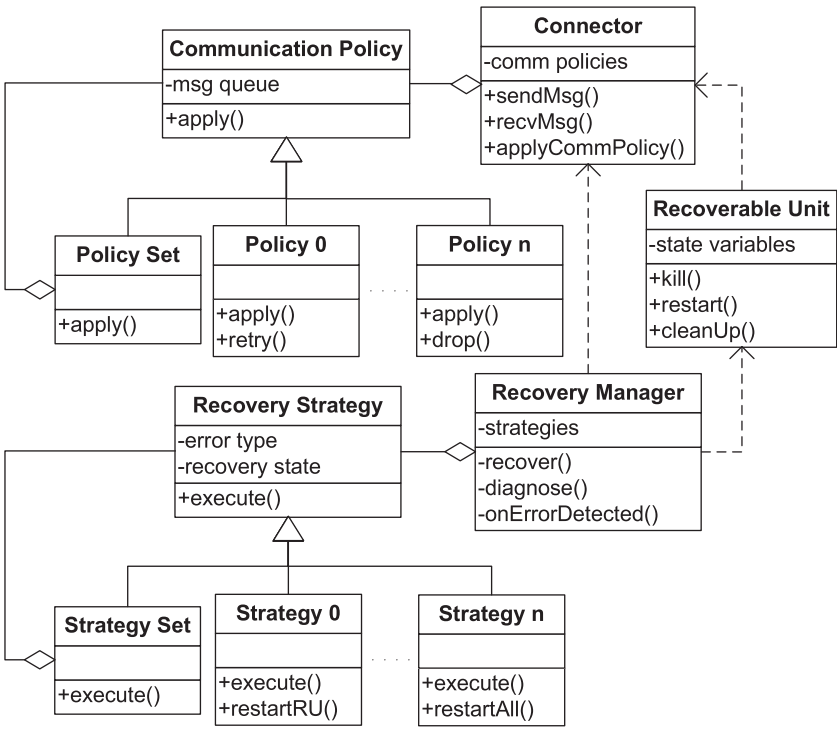

Figure 3. A conceptual view of FLORA.

restart) in accordance with the recovery strategy being executed. Recovery strategies are coupled with error types, from which they can recover. They can also be composed of a combination of other primitive or composite strategies and they can have multiple states.

FLORA comprises Inter-Process Communication (IPC) utilities, serialization/de-serialization primitives, error detection and diagnosis mechanisms, an RU wrapper template, one central recovery manager and one central communication manager (i.e. Connector) that communicate with one or more instances of RU. The framework has been implemented in the C language on a Linux platform. Currently, FLORA implements the detection of fatal and deadlock errors. Other error types can be implemented in due time but this does not impact the framework itself. In the following section, we explain how FLORA can be applied to adapt a given architecture for local recovery.

5. APPLICATION OF FLORA

Figure 4 depicts the overall process for applying FLORA, which consists of the steps *Architecture Design*, *Analysis* and *Realization*. In the *Architecture Design* step, we expect that a module view of the architecture is provided. The application of FLORA is agnostic to the architecture design method that is used in this step. The module view of the described architecture is provided to the *Analysis* step as an input. In the *Analysis* step, the system is analyzed to define the decomposition

DOI: 10.1002/spe

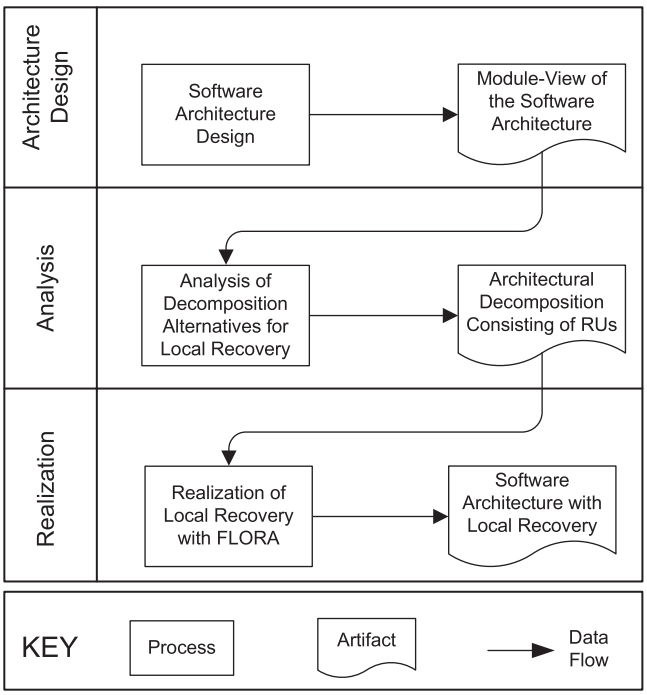

Figure 4. The overall process for the application of FLORA.

of the architecture into a set of RUs. In the *Realization* step, the local recovery is realized with FLORA according to this decomposition. In the following section, we discuss the *Analysis* and *Realization* steps in detail.

5.1. Analysis

There are several ways in which a system can be partitioned into a set of RUs. Each alternative may have both benefits and drawbacks. In the following subsections, we first discuss the design space. Then we outline the basic criteria to be considered and the analysis necessary to select an alternative systematically.

5.1.1. The design space

Figure 2 shows an alternative decomposition for decomposing the system into RUs and as such introducing local recovery. Obviously, the partitioning of modules can be done in many different ways. To reason about the number of decomposition alternatives, we first need to model the design space that defines the set of decomposition alternatives. In fact, the partitioning of architecture into a set of RUs can be generalized to the well-known *set partitioning problem* [10]. The total number of ways to partition a set of n elements into arbitrary number of nonempty sets is counted by the nth

DOI: 10.1002/spe

Bell number, B_n [10]. In theory, B_n is the total number of partitions of a system with n modules. B_n grows exponentially with n. For example, $B_1 = 1$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, $B_7 = 877$ (The MPlayer case), $B_{15} = 1\,382\,958\,545$.

However, not all theoretically possible decompositions are practically possible because modules of the architecture cannot be freely allocated to RUs due to domain constraints. Usually such constraints are specified with *mutex* and *require* relations [11]. For example, we have specified the following constraints for the MPlayer case.

- *Demuxer* ***requires*** *Stream*
- *Demuxer* ***requires*** *Libmpcodecs*

The constraints above simply specify that the *Demuxer* module must be in the same RU as the *Stream* and *Libmpcodecs* modules. We have used a recursive lexicographic algorithm [12] to generate all possible decomposition alternatives[‡]. There exist in total 877 decomposition alternatives for the MPlayer case. After applying the above constraints, 52 alternatives were left[§]. To select one of the remaining decomposition alternatives, we will assess them for availability and performance criteria, as discussed in the following subsections.

5.1.2. Selecting decomposition alternatives for availability

The main goal of local recovery is to make the system available to the user as much as possible. The total availability of a system depends on the availability of its individual RUs. Equation (1) shows the formula of availability.

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{1}$$

In Equation (1), MTTF and MTTR stand for the mean time to failure and the mean time to recover, respectively. To maximize the availability of the overall system, MTTF of RUs must be kept high and MTTR of RUs must be kept low. The MTTF and MTTR of RUs on their turn depend on the MTTF and MTTR values of the contained modules. As such, the overall value of MTTF and MTTR properties of the system depends on the decomposition alternative, that is, how we separate and isolate the modules into a set of RUs. As a general heuristic, system modules with high MTTR should be placed in different RUs than the ones with low MTTF [13]. Otherwise, frequent errors introduced by a module with low MTTF can trigger the slow recovery of modules with high MTTR. To improve the availability as much as possible, *mutex* relations can be used as constraints for enforcing that the system modules with high MTTR are placed in different RUs than the ones with low MTTF. For example, the audio and video processing is very critical for MPlayer and the recovery of the corresponding modules is slow (the whole audio–video streaming graph must be initialized). These modules must be isolated from the *Gui* module, which can be recovered fast and

[‡] In general, there exist combinatorial generation algorithms for the *set partitioning* problem [10].

[§] Note that these numbers are just examples. They illustrate the dramatic decrease in the design space that can be achieved with just a few constraints.

DOI: 10.1002/spe

which is subject to various errors based on the user interaction. Accordingly, we have specified the following constraints for the MPlayer case.

- *Gui* ***mutex*** *Libao*
- *Gui* ***mutex*** *Libvo*

The constraints above simply specify that the *Gui* module must be in a separate RU than the *Libao* and *Libvo* modules. After applying the above constraints together with the domain constraints as specified before, 27 alternatives were left.

5.1.3. Assessing performance overhead of decomposition alternatives

Decomposing the architecture into a set of RUs leads to a performance overhead due to the dependencies between the separated modules. We distinguish between two important types of dependencies that cause a performance overhead; (1) *function dependency* and (2) *data dependency*.

- *Function dependency*: By function dependency, we mean the amount of function calls between modules across different RUs. For transparent recovery, these function calls must be redirected, which leads to an additional performance overhead. For example, FLORA captures all function calls across different RUs and redirects them through IPC calls. For this reason, we should consider the amount of interactions between the chosen RU boundaries before selecting RUs.
- *Data dependency*: In the current literature on local recovery [3,4] it is assumed that the RUs do not contain shared state variables and as such are stateless. This assumption can hold, for example, for stateless components [3] and stateless device drivers in [4]. However, when we decompose an existing system into RUs, there might be shared state variables leading to data dependencies between RUs. Data dependencies complicate the recovery and increase the time to recover since such data need to be kept synchronized after recovery. This makes the amount of data dependency between RUs an important criteria for selecting RUs.

In the following subsections, we discuss our experiences and results with the analysis approaches that we utilized for assessing the decomposition alternatives with respect to the performance criteria.

- *Static analysis*: To assess the performance overhead introduced by a particular decomposition alternative, we need to analyze the function and data dependencies between the modules. For this aim, we have first tried to utilize static analysis approaches. Static analysis approaches inspect the source code of programs and perform the analysis without actually executing these programs. The sophistication of the analysis performed by tools varies depending on the granularity of the analysis (e.g. individual statements, individual source files) and the purpose (e.g. spotting potential errors, verifying specified properties) [14]. We have used several static source code analysis tools to derive both the function call graph and data dependency graph of the analyzed system. Our aim was to utilize these graphs for deriving the function and data dependencies among the modules of the system. However, static function call graph is not sufficient for function dependency analysis. This is because, for calculating the actual performance overhead introduced by a particular decomposition, we also need the frequency of function calls and their execution times. Moreover, also static data flow analysis turned out not to be practical and scalable. Even for the MPlayer code base with approximately 700K lines of C code, we were unable to obtain a data flow information with the state-of-the-art source code analysis tools [15,16]. The computation becomes even more expensive, and very

DOI: 10.1002/spe

soon intractable, when we employ detailed data analysis such as pointer analysis [17]. As a result, static analysis approaches appear to be inappropriate for our purpose.

- *Dynamic analysis*: Because of the problems we have faced with static analysis approaches, we have adopted a dynamic analysis approach and we have utilized existing profiling tools [18,19] for this purpose. Again, we have considered both function dependency and data dependency analysis.

  For the function dependency analysis, we have derived the function call graph and function call profile of the system (i.e. frequency of performed function calls and the execution time of functions) with the *GNU gprof* tool [18]. We have stored this information in a database for querying the number of function calls between the selected RU boundaries. We have calculated the function dependency overhead by taking the ratio of the number of function calls that pass the boundaries of RUs to the total number of function calls in the system.

  For the data dependency analysis, we have derived the memory access profile of the system modules (i.e. which modules accesses which memory locations of what size) by utilizing the Valgrind [19] tool. We have also stored this information in a database for querying the size of the memory that is shared by the modules from different RUs.

Figure 5 shows the function dependency overhead and data dependency size for the 27 decomposition alternatives that were within the previously specified constraints. In Figure 5 the third decomposition alternative corresponds to the decomposition of the system into the three recoverable units *RU AUDIO*, *RU GUI* and *RU MPCORE* as defined in Figure 2. This decomposition alternative has function dependency overhead and data dependency size calculated as approximately 5% and 5 KB, respectively.

As a result of the analysis step, we select a particular decomposition with a set of RUs. In this case, we have selected the third decomposition alternative shown in Figure 2. The next step is to realize local recovery for this decomposition by applying FLORA.

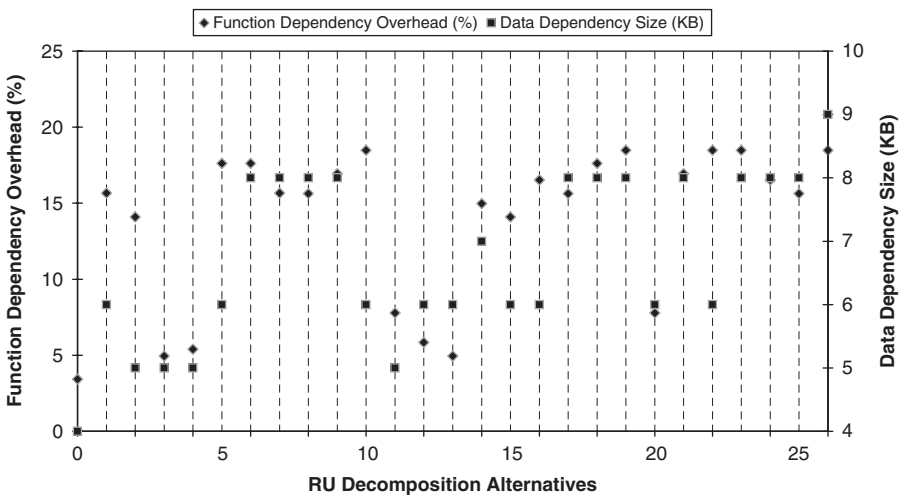

Figure 5. Function dependencies and data dependency sizes for decomposition alternatives that satisfy the decomposition constraints.

DOI: 10.1002/spe

```
 1: #include "util.h"
 2: #include "recunit.h"
 3: #include "rugui.h"
 4: ...
 5: #define STATE_VARS ... &guiIntfStruct
 6: ...
 7:
 8: void cleanUp() {
 9:   /* no component specific cleanup */
10: }
11:
12: void __ruGui(struct recunit info) {
13:   INIT_RU(SOCK_PATH_RECMGR, SOCK_PATH_CONN)
14:   ...
15:   PRESERVE_STATE
16:   ...
17:   processMsgs();
18: }
19:
20: void catchInterfaces() {
21:   BEGIN
22:     CATCH(INTERFACE_GUI, apOnMsgRcvd_gui)
23:   END
24: }
25:
26: void OnMsgRcvd_gui_guiInit() {
27:   guiInit();
28:   RETURN(INTERFACE_GUI, msg_gui_guiInit)
29: }
30: ...
```

Figure 6. RU Wrapper code for RU GUI.

5.2. Realization

Once the RUs have been defined, we can use FLORA to realize local recovery (See Figure 4). In this realization step, each RU is wrapped using the RU wrapper template as shown in Figure 6. The wrapper includes the necessary set of utilities for isolating and controlling an RU (lines 1–3). A set of state variables can be declared to be checkpointed (line 5). If needed, cleanup specific to the RU (e.g. allocated resources) can be specified (lines 8–10) as a preparation for recovery. Post-recovery initialization (lines 12–18) by default includes: (1) maintaining the connection with the *Connector* and the *Recovery Manager* (line 13), (2) obtaining the checkpointed state variables (line 15) and (3) start processing incoming messages from other RUs (line 17). Additional RU-specific initialization actions can also be specified here.

Each RU provides a set of interfaces, which are captured based on the specification in the wrapper (lines 20–24). Each interface defines a set of functions that are marshaled [20] and transferred through IPC. On reception of these calls, the corresponding functions are called and then the results are returned (lines 26–29). In all other RUs where this function is declared, function calls are redirected through IPC to the corresponding interface with C MACRO definitions. In principle we could also use aspect-oriented programming techniques [21] for this, provided that an appropriate

DOI: 10.1002/spe

```
1: #define guiInit() mpcore_gui_guiInit()
2: ...
3:
4: void mpcore_gui_guiInit() {
5:   CALL(INTERFACE_GUI, msg_gui_guiInit)
6: }
7: ...
```

Figure 7. Function redirection through RU interfaces.

weaver for the C language is available. In Figure 7, a code section is shown from one of the modules of *RU MPCORE*, where all calls to the function *guiInit* are redirected to the function *mpcore_gui_guiInit* (line 1), which activates the corresponding interface (*INTERFACE_GUI*) instead of performing the function call (lines 4–6).

Figure 8 depicts the design of the MPlayer after local recovery is introduced using the framework. Note that this is a separate architectural view [22] defined for local recovery in particular. The local recovery view is based on the *recovery style* that we have introduced earlier in [5]. In Figure 8, we can see the three RUs, *RU MPCORE*, *RU GUI* and *RU AUDIO*. In addition, the components *Connector* and *Recovery Manager* have been introduced by the framework. Each RU can detect deadlock errors[¶]. *Recovery Manager* can detect fatal errors[‖]. All error notifications are sent to *Connector*, which comprises the diagnosis facility. Diagnosis information is conveyed to *Recovery Manager*, which kills a set of RUs and/or restarts a dead RU. Messages that are sent from RUs to *Connector* are stored (i.e. queued) by RUs in case the destination RU is not available and they are forwarded when the RU becomes operational again.

6. EVALUATION

The main goal of local recovery is to increase the system availability. In this section we discuss the increase of availability as a result of applying FLORA. For this purpose, we have used FLORA to introduce local recovery to MPlayer for 3 different decomposition alternatives: (1) global recovery, where all the modules are places in a single RU ({ [*Mplayer*, *Libmpcodecs*, *Libvo*, *Demuxer*, *Stream*, *Gui*, *Libao*] }) (2) local recovery with two RUs, where the module *Gui* is isolated from the rest of the modules ({ [*Mplayer*, *Libmpcodecs*, *Libvo*, *Demuxer*, *Stream*, *Libao*] [*Gui*] }) (3) local recovery with three RUs, where the module *Gui*, *Libao* and the rest of the modules are isolated from each other ({ [*Mplayer*, *Libmpcodecs*, *Libvo*, *Demuxer*, *Stream*] [*Libao*] [*Gui*] }). The isolated modules were selected to have observable functionality and to be more relevant with respect to user perception. We could 'see' the restarting of Gui panel or 'hear' the lack of sound, while *Libao* is restarting.

To be able to measure the availability achieved with these three implementations, we have modified each module so that they fail with a specified failure rate (assuming an exponential

[¶] An RU wrapper detects if an expected response to a message is not received within a configured timeout period.
[‖] The recovery manager is the parent process of all RUs and receives and handles a signal when a child process is dead.

DOI: 10.1002/spe

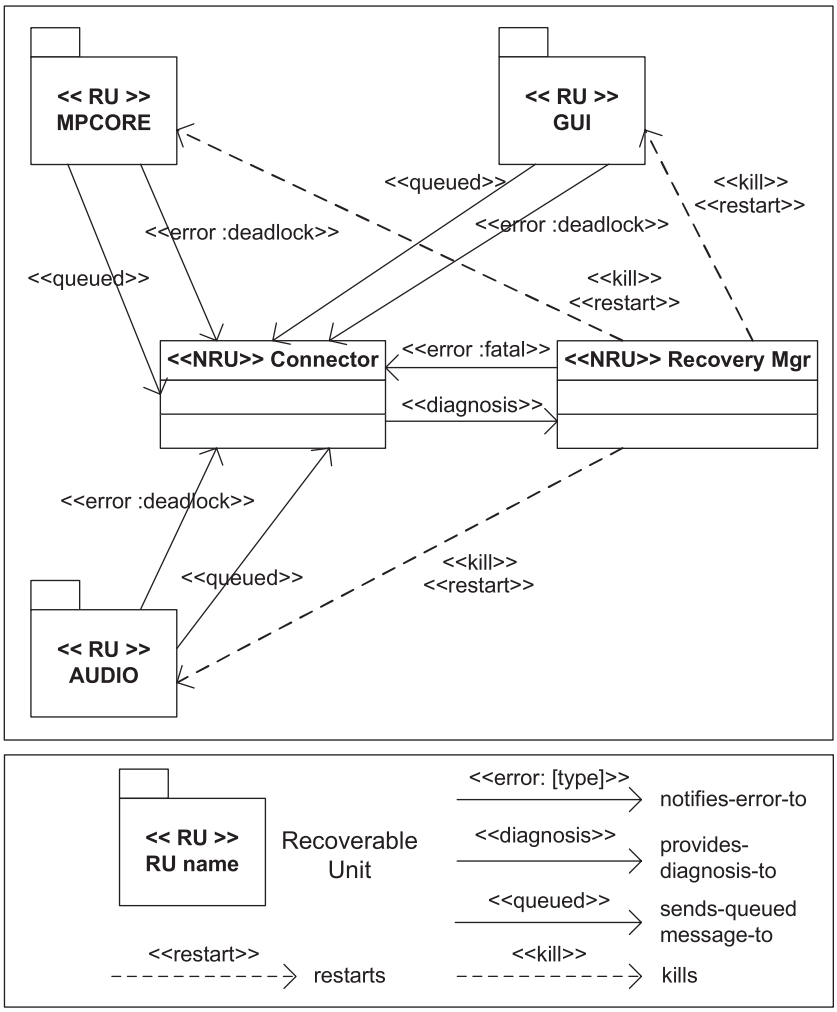

Figure 8. Application of FLORA for MPlayer.

distribution with mean MTTF). After a module is initialized, it creates a thread that is periodically activated every second to inject errors. The operation of the thread is shown in Algorithm 1.

The error injection thread first records the initialization time (line 1). Then, each time it is activated, the thread calculates the time elapsed since the initialization (line 3). The MTTF value of the corresponding module and the elapsed time is used for calculating the probability of error occurrence (line 4). In line 5, *random*() returns, from a uniform distribution, a sample value $r \in [0, 1]$. This value is compared with the calculated probability to decide whether or not to inject an error (line 6). Possibly an error is injected by basically creating a fatal error with an illegal memory operation. This error crashes the process on which the module is running (line 7).

DOI: 10.1002/spe

Algorithm 1 Periodically activated thread for error injection

```
1. time_init ← currentTime()
2. while TRUE do
3.    time_elapsed ← currentTime() − time_init
4.    p ← 1 − 1/e^(time_elapsed/MTTF)
5.    r ← random()
6.    if p ⩾ r then
7.       injectError()
8.       break
9.    end if
10. end while
```

Table I. Measured system availability for different decomposition alternatives.

Decomposition alternative	Availability (*all* MTTF = 1800 s)	Availability ($MTTF_{Libao}$ = 60 s, $MTTF_{Gui}$ = 30 s, *all other* MTTF = 1800 s)
All modules in 1 RU	97.57	83.59
Gui, *the rest*	97.58	93.25
Gui, *Libao*, *the rest*	97.75	97.75

The *Recovery Manager* component of FLORA logs the initialization and failure times of RUs to a file during the execution of the system. For each of the implemented alternatives, we have let the system run for 5 h. Then, we have processed the log files to calculate the time T_{avail} when the RU that contains the core system module, *Mplayer*, has been down. The whole system fails if and only if this RU fails. Hence, T_{avail} corresponds, by definition, to the system availability as a whole. We have calculated the steady-state availability of the system as the value $T_{avail}/5$.

The results of the measured system availability are shown in Table I. The first column lists the different decomposition alternatives. The second column shows the measured availability for the corresponding decomposition, where the MTTF value is specified as 1800 s for all the modules. The third column shows the availability, where the MTTF values for the modules *Libao* and *Gui* are specified as 60 and 30 s, respectively.

In Table I, we see that the more we decompose the architecture with FLORA, the more the availability of the system increases. The difference is even more significant when we decrease the MTTF values for the isolated modules. This is because, the isolation of relatively more unreliable modules of the RU increases the reliability of that RU significantly. A more reliable RU fails less often, which in turn increases the system availability. The availability measures at the last row of Table I are the same because the RU that determines the system availability contains the same set of modules (*Libmpcodecs*, *Demuxer*, *Mplayer*, *Libvo*, *Stream*) with the same MTTF values.

As such, the application of FLORA for introducing local recovery increases the availability of a system that comprises erroneous modules.

DOI: 10.1002/spe

Table II. LOC for the selected RUs (as shown in Figure 2), LOC for the corresponding wrappers and their ratio.

	LOC_{RU}	$LOC_{RU\ wrapper}$	Ratio (%)
RU MPCORE	214K	463	0.22
RU GUI	20K	345	1.72
RU AUDIO	8K	209	2.61
TOTAL	242K	1017	0.42

7. APPLICABILITY OF FLORA

If all the function calls that pass the boundaries of RUs are defined, FLORA guarantees the correct execution and recovery of these RUs. However, the specification of the RU boundaries with the RU wrapper template requires an additional effort. The main effort is spent due to the definition of the RU wrappers. For the decomposition shown in Figure 2, we have measured this effort based on the LOC written for RU wrappers and the actual size of the corresponding RUs**. Table II shows the LOC for each RU (LOC_{RU}), LOC of its wrapper ($LOC_{RU\ wrapper}$) and their ratio (($LOC_{RU\ wrapper}/LOC_{RU}$) $\times 100$).

As we can see from Table II, we had to write approximately 1K LOC to apply FLORA for the presented case study. The LOC written for wrappers is negligible compared with the corresponding system parts that are wrapped. The size of the wrapper becomes even less significant for bigger system parts. In fact, the wrapper size is independent of the size and internal complexity of the system part that is wrapped. This is because the wrapper captures only the interaction of an RU with the rest of the system.

To be able to estimate the LOC to be written for wrappers, we have used the following equation:

$$LOC_{total} = 30 \times |RU| + 15 \times \sum_{r \in RU} calls(r \rightarrow \not{r}) \tag{2}$$

Equation (2) estimates that the LOC needs to be written for wrappers based on the following assumptions. There should be a wrapper for each RU with some default settings (Figure 6). Therefore, the equation includes a fixed amount of LOC (30) times the number of RUs ($|RU|$). In addition, all function calls between RUs must be defined in the corresponding wrappers. For each such function call we add a fixed amount of LOC (15) taking into account the code for redirection of the function, capturing and processing its arguments and return values. To calculate Equation (2), we used the function dependency analysis for calculating the number of calls between the selected RU boundaries. We have generated all the set of possible partitions for varying number of RUs. We have calculated Equation (2) for each possible partition and we have determined the minimum and maximum LOC estimations with respect to the number of RUs. The results can be seen in Figure 9.

In Figure 9, we can see the range of LOC estimations with respect to the number of RUs. When the number of RUs is equal to 1 or 7 (i.e. equal to the number of modules), there exists logically

**We have excluded the source code for the various supported codecs, which are encapsulated mostly in *Libmpcodecs*.

DOI: 10.1002/spe

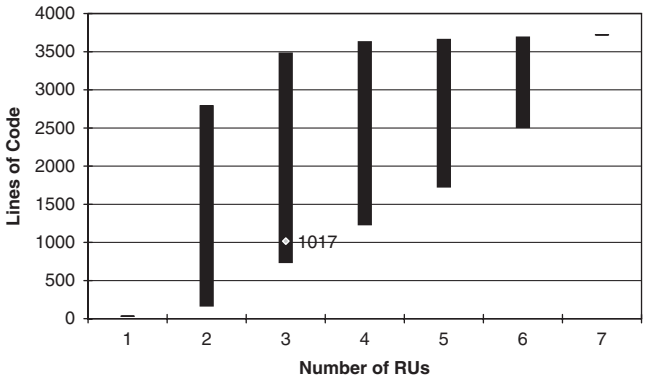

Figure 9. Estimated LOC to be written for wrappers with respect to the number of RUs.

only one possible partition. Therefore, the minimum and the maximum values are equal for these cases. Figure 9 also marks the LOC written in the actual implemented case with 3 RUs as presented in Table II. FLORA itself includes source code of size 1.5K LOC, which was reused as is. Thus, we can provide an approximate prediction of the effort for utilizing the framework before the actual implementation.

Depending on how homogeneous the coupling between system modules is, this analysis can point out exceptional decompositions. For instance, in the analysis of the MPlayer case, we can see that several decompositions with 6 RUs require less overhead compared with the maximum overhead caused by a decomposition with 2 RUs. This means that there are certain modules that are exceptionally highly coupled (e.g. *Libvo* and *Libmpcodecs*) compared with the other modules.

8. DISCUSSION

8.1. Specifying MTTF and MTTR values

In our case study, we have measured the MTTR values from the actual implementation by calculating the mean time it takes to restart a process and the corresponding modules over 100 runs. Initially, we have used the same MTTF value for all the modules. In the second step of the evaluation, we have decreased the MTTF values for the modules that are isolated in different RUs to amplify the effect of local recovery on system availability and to better observe the difference with global recovery.

In principle, there are three strategies that can be used for determining the MTTF values:

- *Using fixed values*: It can be assumed that all modules have the same MTTF. Accordingly, MTTF values can be fixed to a certain value just to investigate the analysis results.
- *What-if analysis*: A range of values can be considered, where the values are varied and their effect is observed.
- *Estimation*: The MTTF values can be estimated based on historical data.

DOI: 10.1002/spe

We do not commit to any of these strategies in our approach. The estimation of actual failure rates is usually the most accurate way to define the MTTF values. However, the historical data (e.g. a problem database) can be missing or not accessible. In that case, either fixed values and/or a what-if analysis can be used.

8.2. Specifying requires/mutex relations

In our case study, we have defined requires and mutex relations manually. In fact, requires relations can be set automatically based on the function and data dependencies between the modules, i.e. if the dependencies between two modules exceed a threshold, they are kept together by defining a requires relation. The specification of mutex relations depends more on the application domain. In principle, the more mutex relations are defined, the better system availability can be achieved. However, some mutex relations can be specified explicitly, for instance, to isolate a third-party software or to protect a critical part of the system from the rest.

8.3. Refactoring the architecture

Depending on the required effort, it is possible to consider refactoring the architecture instead of trying to define RUs with minimal dependencies. The effort needed to refactor the architecture depends very much on the nature and semantics of the architecture. Restructuring for decoupling shared variables can be considered in case the architecture is already being refactored to improve certain qualities like reusability, maintainability or evolvability. In fact, this is a viable approach if the architects/developers have very good insight in the system. Otherwise, it would be better to treat modules as black boxes and wrap them with FLORA.

8.4. State preservation

State variables that are critical and need to be saved can be declared in RU wrappers. Declared state variables are check-pointed and automatically restored after recovery by FLORA. For instance, in the wrapper of *RU GUI* (see Figure 6), the memory location of variable *guiInfStruct* is declared (line 5). This variable is a C structure (*struct* [23]), which stores various information like the name of the media file, volume level and the current position in the stream. These values are restored after the *Gui* module is restarted. However, particular states that are critical for modules are application dependent. Such states must be known and explicitly declared in the corresponding wrappers by developers.

8.5. Partial failures

We have calculated the availability of the system according to the proportion of time that the core system module, *Mplayer* has been down. There are also partial failures, where *RU GUI* and *RU AUDIO* are restarted. In these cases, GUI panel vanishes and audio stops playing until the corresponding RUs are recovered. These are also failures from the user's perspective but we have neglected these failures to have a common basis for comparison with global recovery. Depending on the application, functional importance of the failed module and the recovery time, the user might get annoyed differently from partial failures. In addition, different users might have different

DOI: 10.1002/spe

perceptions. Hence, to take into account different types of partial failures, experiments are needed to be conducted to determine their actual effect on users [24]. In any case, the failure of the whole system would be the most annoying of all.

8.6. Frequency of data access

For data dependency analysis, we have evaluated the size of shared data among modules. The shared data size is important as such data should be synchronized back after each recovery. The number of data access, on the other hand, is important due to redirection of the access through IPC. This is mostly covered by he function dependency analysis, where data are accessed through function calls. The correlation between function dependency overhead and data dependency size (Figure 5) shows this. In fact, the integration with FLORA requires that all data access must be performed through explicit function calls. For each direct data access without a function call, if there are any, the corresponding parts of the system are refactored to fulfill this requirement. After this refactoring, the function dependency analysis can be repeated to get more accurate results with respect to the expected performance overhead.

8.7. Limitations of dynamic analysis

An important advantage of dynamic analysis approaches is their ability to capture detailed interactions at run-time (e.g. pointer operations, late binding). This leads to more accurate information about the analyzed program compared with what might be obtained with static analysis. On the other hand, dynamic analysis also has limitations and drawbacks.

First, instrumentation or probing has an effect on the execution of the target program. As a potential risk, this effect can influence the program behavior and analysis results. We have not addressed this issue since we are performing comparative analysis and all the functions are affected by instrumentation. If there is a feeling that there is a big variance in how different functions are affected, modules can be analyzed separately and the difference introduced by the instrumentation can be measured.

Second, collected data for analysis depends on the usage scenarios and program input. In our case study, we have performed our measurements for the video-playing scenario, which is a common usage scenario for a media player application. In principle, it is possible to take different types of usage scenarios into account. The results obtained from several system runs are statistically combined by the tools [18]. If there is a high variance among the execution of scenarios, where statistically combining the results would be wrong, multiple scenarios can be repeated in a period of time and the overhead can be calculated based on the profile information collected during this time period. However, this would require selection and prioritization of a representative set of scenarios with respect to their importance from the user point of view (user profile) [24]. The analysis process will remain the same although the input profile data can be different.

9. RELATED WORK

We have implemented FLORA basically using macro-definitions in the C language. We could also implement FLORA using aspect-oriented programming techniques [21] in which usually a

DOI: 10.1002/spe

distinction is made between *base code* on which additional so-called *crosscutting concerns* are *woven*. In particular the function redirection calls to IPC could be automatically woven using aspects. We consider this as our future work.

A software architecture is usually represented using more than one architectural view [22]. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. The fundamental reason for modeling different views of the architecture is that current software systems are too complex to represent all the concerns in one model. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. For utilizing FLORA actually we had to define a separate view that makes the recovery concern explicit. We have defined a *recovery style* in [5] to define local recovery views.

In [25], a survey of approaches for application-level fault-tolerance is presented. According to the categorization of this survey, FLORA falls into the category of *single-version software fault-tolerance libraries* (*SV libraries*). SV libraries are said to be limited in terms of separation of concerns, syntactical adequacy and adaptability [25]. On the other hand, they provide a good ratio of cost over improvement of the dependability, where the designer can reuse existing, long-tested and sophisticated pieces of software [25]. An example SV library is *libft* [26], which collects reusable software components for recovery. However, like other SV libraries [25] it does not support local recovery.

Candea *et al*. introduced the *microreboot* [3] approach, where local recovery is applied to increase the availability of Java-based Internet systems. Microreboot aims at recovering from errors by restarting a minimal subset of components of the system. Progressively larger subsets of components are restarted as long as the recovery is not successful. To employ microreboot, a system has to meet a set of architectural requirements (i.e. *crash-only design* [9]), where components are isolated from each other and their state information is kept in state repositories. Unfortunately, designs of many existing systems do not have these properties. Such systems should be either redeveloped from scratch or they should be modified to support the necessary requirements. It is usually too costly to redesign and implement the whole system from the start. On the other hand, maintenance costs for modifying a system can be also very high depending on its design. Moreover, dedicated solutions for a particular application cannot be reused for other systems. FLORA provides the necessary set of abstractions and mechanisms that can be reused to introduce local recovery without a need for redesigning the whole system.

In [4], a micro-kernel architecture is introduced, where device drivers are executed on separate processes at user space to increase the failure resilience of an operating system. In case of a driver failure, the corresponding process can be restarted without affecting the kernel. The design of the operating system must support isolation between the core operating system and its extensions to enable such a recovery [4]. Mach kernel [27] also provides a micro-kernel architecture and flexible multiprocessing support, which can be exploited for failure resilience and isolation. Singularity [8] proposes multiprocessing support in particular to improve dependability and safety by introducing the concept of *sealed process architecture*. This architecture limits the scopes of processes and their capabilities with respect to memory alteration for better isolation. To be able to exploit the multiprocessing support of the operating system for isolation, the application software must be partitioned to be run on multiple processes. FLORA supports this process and reduces the re-engineering effort, while making use of the multiprocessing support of the Linux operating system.

DOI: 10.1002/spe

Erlang/OTP [28] is used by Ericsson to achieve highly available network switches by enabling local recovery. Their framework is used for restructuring Erlang programs using Erlang/OTP Design Principles. FLORA has the same goal as Erlang/OTP but it is used for restructuring C programs.

Aurora Management Workbench (AMW) [29] uses model-centric development to integrate a software system with an high availability middleware. AMW generates code based on the desired fault-tolerance behavior that is specified with a domain specific language. By this way, it aims at reducing the amount of hand-written code and as such reducing the developer effort to integrate fault-tolerance mechanisms provided by the middleware with the system being developed. Developers should manually write code only for component-specific initialization, data maintenance and invocations of check-pointing APIs similar to those specified within RU wrapper templates. AMW allows software components (or servers) to be assigned to separate RUs (*capsule* in AMW terminology) and be restarted independently. However, AMW currently does not support the restructuring and partitioning of legacy software to introduce local recovery.

10. CONCLUSION

In this paper, we have discussed our experiences in introducing local recovery to a system. Local recovery is an effective approach for increasing availability. However, it appears that the required decomposition for local recovery is usually not aligned with the decomposition based on functional concerns. Moreover, the realization of local recovery requires substantial development and maintenance effort. We have presented the framework FLORA that provides reusable abstractions to preserve the existing structure and support the realization of local recovery. We have illustrated FLORA to define three recoverable units (RUs) in the open-source media player called MPlayer. These three RUs were overlaid on the existing structure without adapting the individual modules. In addition, the realization effort for applying the framework and introducing local recovery appears to be relatively negligible. We have also seen that introducing local recovery by decomposing the software architecture increases the availability of the system. The increase of availability is more significant when there is more difference between the reliability of the system modules. An important part of the approach is the dependency analysis for identifying RUs based on the performance criteria. We have seen that the dynamic analysis approaches are more practical and scalable than static analysis approaches with respect to function and data dependency analysis applied to large code bases. The application of the framework, as such, provides a reusable and practical approach to introduce local recovery to software architectures.

ACKNOWLEDGEMENTS

We acknowledge the feedback from the discussions with our TRADER project partners from NXP Research, NXP Semiconductors, Philips TASS, Philips Consumer Electronics, Design Technology Institute, Embedded Systems Institute, IMEC, Leiden University and Delft University of Technology.

This work has been carried out as a 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.

REFERENCES

1. Parnas DL. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
2. Trader project. ESI, 2009. http://www.esi.nl.
3. Candea G, Kawamoto S, Fujiki Y, Friedman G, Fox A. Microreboot: A technique for cheap recovery. *OSDI*, USENIX, 2004; 31–44.
4. Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS. Failure resilience for device drivers. *DSN'07*. IEEE Computer Society: Silver Spring, MD, 2007; 41–50.
5. Sozer H, Tekinerdogan B. Introducing recovery style for modeling and analyzing system recovery. *WICSA'08*. IEEE Computer Society: Silver Spring, MD, 2008; 167–176.
6. MPlayer official website. 2008. http://www.mplayerhq.hu/.
7. Avizienis A, Laprie J-C, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing* 2004; **1**(1):11–33.
8. Hunt GC, Aiken M, Fähndrich M, Hawblitzel C, Hodson O, Larus J, Levi S, Steensgaard B, Tarditi D, Wobber T. Sealing OS processes to improve dependability and safety. *SIGOPS* 2007; **41**(3):341–354.
9. Candea G, Fox A. Crash-only software. *Proceedings of the Workshop on Hot Topics in Operating Systems* (*HotOS*), Hawaii, U.S.A., 2003; 67–72.
10. Harris JM, Hirst JL, Mossinghoff MJ. *Combinatorics and Graph Theory*. Springer: Berlin, 2000.
11. Czarnecki K, Eisenecker U. *Generative Programming*: *Methods*, *Tools*, *and Applications*. Addison-Wesley Professional: 2000.
12. Ruskey F. Simple combinatorial gray codes constructed by reversing sublists. *Proceedings of the 4th International Symposium on Algorithms and Computation* (*ISAAC 1993*) (*Lecture Notes in Computer Science*, vol. 762). Springer: Berlin, 1993; 201–208.
13. Candea G, Cutler J, Fox A, Doshi R, Garg P, Gowda R. Reducing recovery time in a small recursively restartable system. *Proceedings of the International Conference on Dependable Systems and Networks* (*DSN*), Washington, DC, U.S.A., 2002; 605–614.
14. Binkley D. Source code analysis: A road map. *Future of Software Engineering* (*FOSE'07*), Washington, DC, U.S.A., 2007; 104–119.
15. Necula GC, McPeak S, Rahul SP, Weimer W. CIL: Intermediate language and tools for analysis and transformation of c programs. *Proceedings of the Conference on Compiler Construction*, Grenoble, France, 2002; 213–228.
16. Teitelbaum T. Codesurfer. *SIGSOFT Software Engineering Notes* 2000; **25**(1):99.
17. Hind M. Pointer analysis—Haven't we solved this problem yet? *Program Analysis for Software Tools and Engineering* (*PASTE'01*). ACM: New York, 2001; 54–61.
18. Fenlason J, Stallman R. *GNU gprof*: *The GNU Profiler*. Free Software Foundation, 2009. http://www.gnu.org/.
19. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices* 2007; **42**(6):89–100.
20. Dollimore J, Kindberg T, Coulouris G. *Distributed Systems*: *Concepts and Design*. Addison-Wesley: Reading, MA, 2005.
21. Elrad T, Fillman R, Bader A. Aspect-oriented programming. *Communications of the ACM* 2001; **44**(10):29–32.
22. Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J. *Documenting Software Architectures*: *Views and Beyond*. Addison-Wesley: Reading, MA, 2002.
23. Kernighan B, Ritchie D. *The C Programming Language*. Prentice-Hall: Englewood Cliffs, NJ, 1988.
24. de Visser I. Analyzing user perceived failure severity in consumer electronics products. *PhD Thesis*, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2008.
25. De Florio V, Blondia C. A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys* 2008; **40**(2):1–37.
26. Huang Y, Kintala C. Software fault tolerance in the application layer. *Software Fault Tolerance*, Lyu MR (ed.), Chapter 10. Wiley: New York, 1995; 231–248.
27. Rashid R, Julin D, Orr D, Sanzi R, Baron R, Forin A, Colub D, Jones M. Mach: A system software kernel. *Proceedings of the 34th Computer Society International Conference* (*COMPCON*), San Francisco, CA, U.S.A., 1989; 176–178.
28. Erlang/OTP design principles, 2009. http://www.erlang.org/doc/.
29. Buskens R, Gonzalez OJ. Model-centric development of highly available software systems. *Architecting Dependable Systems IV* (*Lecture Notes in Computer Science*), de Lemos R, Gacek C, Romanovsky A (eds.). Springer: Berlin, 2007; 409–433.

DOI: 10.1002/spe