

Introducing Recovery Style for Modeling and Analyzing System Recovery *

Hasan Sözer, Bedir Tekinerdoğan
Department of Computer Science, University of Twente
P.O. Box 217 7500 AE Enschede, The Netherlands
{sozerh, bedir}@cs.utwente.nl

Abstract

An analysis of the existing approaches for representing architectural views reveals that they focus mainly on functional concerns and are limited when considering quality concerns. We introduce the recovery style for modeling the structure of the system related to the recovery concern. The recovery style is a specialization of the module viewtype in the Views&Beyond approach. It is used to communicate and analyze architectural design decisions and to support detailed design with respect to recovery. We illustrate the style for modeling the recovery views for the open-source software, MPlayer.

1. Introduction

A software architecture for a program or computing system consists of the structure or structures of that system, which comprise elements, the externally visible properties of those elements, and the relationships among them [17]. Like this definition implies, the software architecture of a system does not consist of a single structure but is represented using more than one architectural view. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern [17]. 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.

Architectural views conform to viewpoints that represent the conventions for constructing and using a view [14].

*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.

In the literature initially some authors have prescribed a fixed set of views to document the architecture. For example, the Rational's Unified Process [12], which is based on Kruchten's 4+1 view approach [11] utilizes the logical view, development view, process view and physical view. Another example is the Siemens Four Views model [8] that uses conceptual view, module view, execution view and code view to document the architecture. Because of the different concerns that need to be addressed for different system, the current trend recognizes that the set of views should not be fixed but multiple viewpoints might be introduced instead. This is also implicit in the IEEE 1471 standard [14] for architectural description which takes a multi-view approach by not committing to any view. IEEE 1471 indicates in an abstract sense that an architecture description consists of a set of views, each of which conforms to a viewpoint [14] realizing the various concerns of the stakeholders. The Views and Beyond (V&B) approach as proposed by Clements et al. is another multi-view approach [17] that appears to be more specific with respect to the viewpoints. To bring order to the proposed views in the literature the V&B approach holds that a system can be generally viewed from so-called three different viewtypes: module viewtype, component & connector viewtype and allocation viewtype. Each viewtype can be further specialized or constrained in so-called architectural styles. An architectural style is a specialization of architectural element types and relationship types, along with any constraints. The notion of styles makes the V&B approach adaptable since the architect may in principle define any style needed.

Certainly, existing multi-view approaches are important for representing the structure and functionality of the system and are necessary to document the architecture systematically. Yet, an analysis of the existing multi-view approaches reveals that they still appear to be incomplete when considering quality properties. The IEEE 1471 standard is not specific with respect to concerns that can be addressed by views. Thus, quality properties can be seen as separate concerns. In the V&B approach quality con-

cerns appear to be implicit in the different views but no specific style has been proposed for this yet. One could argue that for addressing quality properties software architecture analysis approaches have been introduced. The difficulty here is that these approaches usually apply a separate quality model, such as queuing networks or process algebra, to analyze the quality properties. Although these models represent often precise calculations they do not depict the decomposition of the architecture and an additional translation from the evaluation of the quality model needs to be performed. To solve the problem preferably an architectural view is required to model the decomposition of the architecture based on the required quality concern.

The context of this research is in the TRADER (Television Related Architecture and Design to Enhance Reliability) project [19], which is carried out together with NXP Semiconductors and several other academic and industrial partners. One of the key objectives in the project is to develop techniques for analyzing recovery at the architecture design level. Hereby modules in a TV can be composed in various ways and each alternative decomposition might lead to a different recovery properties. To represent the functionality of the system we have developed different architectural views including module view, component and connector view and deployment view. None of these views however directly shows the decomposition of the architecture based on recovery concern. On the other hand using separate quality models such as fault trees helps to provide a thorough analysis but is separate from the architectural decomposition. A practical and easy-to-use approach coherent with the existing multi-view approach was required to understand the system from a recovery point of view.

Based on these industrial experiences we introduce the recovery style as an explicit style for depicting the architecture from the recovery viewpoint. The recovery style is a specialization of the module viewtype in the V&B approach. Similar to the module viewtype, component viewtype and allocation viewtype, which define units of decomposition of the architecture, the recovery style also provides a view of the architecture. Unlike conventional analysis techniques that require different models, recovery views directly represent the decomposition of the architecture and as such help to understand the structure of the system related to the recovery concern. The recovery style considers *recoverable units* as first class elements, which represent the units of isolation, error containment and recovery control. The style defines basic relations for coordination and application of recovery actions. As a further specialization of the recovery style, the local recovery style is provided, which is supported with a framework for introducing local recovery to a system. The usage of the recovery style is illustrated by introducing local recovery to the open-source software, MPlayer [16].

The remainder of this paper is organized as follows. Section 2 provides background information on fault tolerance and recovery. Section 3 illustrates the problem in the context of a case study, which is also used to illustrate the usage of the recovery style. In Section 4, we introduce the recovery and the local recovery styles, related definitions, notations and properties. In Section 5, we utilize the recovery style for the refactored MPlayer software. In Section 6, we provide a generic method to realize the recovery style for a system. In Section 7, related previous studies are summarized. Finally, in section 8 we discuss some future work issues and provide the conclusions.

2. Fault Tolerance and Recovery

Reliability is the ability of a system to perform its required functions under stated conditions for a specified period of time [1]. That is the ability of the system to function without a *failure*, which is as a deviation of the delivered service of a system from the a correct service [1]. A correct service is delivered when the service implements the required system function. An *error* is defined as the system state that is liable to lead to a failure and the cause of an error is called a *fault* [1]. The following figure depicts the fundamental chain of these concepts that leads to a failure.

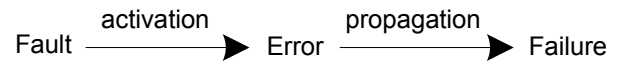


Figure 1. The Fundamental Chain of Reliability Threats Leading to a Failure

In fact, the chain of threats can include multiple errors. An error can propagate to other errors and finally lead to a failure. In order to prevent a failure, this chain must be broken. This is possible through 1) preventing occurrence of faults, 2) removing existing faults, or 3) preventing errors and their propagation [1].

The last approach is called fault tolerance in which it is accepted that faults exist and they might get activated but their consequences (i.e. errors) are taken care of, if possible before they lead to a failure. Error detection is the first necessary step for fault tolerance. As another essential step, detected errors must be recovered. If the recovery affects the whole system, then the system becomes unavailable during the recovery time. For example, in case of a deadlock, restarting the whole system makes it completely unavailable until the system is again in its normal operational mode. This problem can be overcome by applying recovery only to the erroneous parts of the system when possible. To recover from a deadlock, for instance, only the components that are involved in the deadlock need to be restarted, while

the other parts can remain available. Moreover, applying recovery to a subset of the system components rather than the whole system decreases the mean time to recover [6]. 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. However, application of local recovery introduces additional requirements for the system. First of all, cause(s) of errors must be localized (i.e. diagnosis). In addition, to prevent the propagation of errors, the system should be separated into a set of units with clear boundaries and isolation between them. Isolation is usually supported by either the operating system (e.g. process isolation [5]) or a middleware (e.g. encapsulation of Enterprise Java Bean objects [6]). The connection and communication between the isolated units must be managed by a third entity. Recovery actions take place, while the system is operational. Consequently, there is interference between the recovery mechanism and the core system functions. This requires the coordination of recovery actions. As the case with the communication control, coordination can be realized in different ways ranging from completely distributed to completely centralized solutions.

3. Problem Statement

3.1. Case Study: MPlayer

MPlayer [16] is a media player, which supports many input formats, codecs and output drivers. It embodies approximately 700K lines of code 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 2 presents a simplified module view [17] of the MPlayer software architecture with basic implementation units and direct dependencies among them. In the following, we briefly explain the important modules that are shown in this view.

Stream reads the input media by bytes, or blocks and provides buffering, seek and skip functions. *Demuxer* demultiplexes (separates) the input to audio and video channels, and reads them from buffered packages. *Mplayer* connects all the other modules, and maintains the synchronization of audio and video. *Libmcodecs* 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 introduced a local recovery mechanism to MPlayer, which can recover from transient faults. As one of the requirements of local recovery, the system had to be splitted into several units that are isolated from each other. We have utilized operating system processes [5] for isolation. The communication between multiple processes had to be controlled and recovery actions had to be coordinated

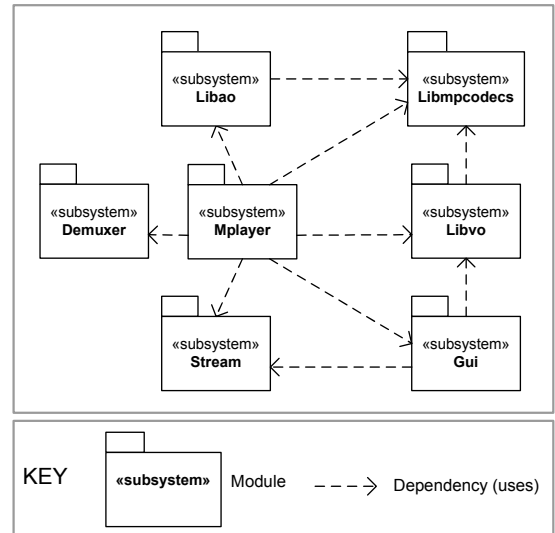


Figure 2. A Simplified Module View of the MPlayer Software Architecture

so that the processes can be restarted independently, while the other processes are operational. Note that the splitting of the system for local recovery is not necessarily, and in general will not be, aligned one-to-one with the functional decomposition of the system. Multiple modules can be wrapped and recovered together.

3.2. Problems

Designing a system with recovery or refactoring it to introduce recovery requires the consideration of several design choices related to; the types of errors that will be recovered, architectural elements that embody error detection and diagnosis mechanisms, the way that the information regarding error detection and diagnosis is conveyed, error containment boundaries, mechanisms for the control of the communication during recovery and the coordination of recovery actions, architectural elements that embody these mechanisms and the way that they interact with the rest of the system.

These design choices should take part in the architecture description of the system to be communicated. Effective communication is one of the fundamental purposes of an architectural description that comprise several views. However, viewpoints that capture functional aspects of the system are limited for representing recovery mechanisms and it is inappropriate for understandability to populate these views with elements and complex relations related to recovery.

Architecture description should also enable analysis of recovery design alternatives. For example, splitting the sys-

tem for local recovery on one hand increases availability of the system during recovery but on the other hand, it leads to a performance overhead due to isolation. This leads to a trade-off analysis. Architecture description of the system should comprise information not only regarding the functional decomposition but also decomposition for isolation to support such analysis.

Realizing a recovery mechanism usually requires dedicated architectural elements and relations that have system-wide impact. Architectural description related to recovery is required also for providing a roadmap for the implementation and supporting the detailed design.

In short, we need a architectural view for recovery to 1) communicate recovery design decisions 2) analyze design alternatives 3) support the detailed design. For this purpose, we introduce the recovery and the local recovery styles.

4. Recovery Style

In this section, we present the recovery style, by means of which we can represent the architectural properties and concepts related to recovery. We derived these concepts and common properties from the literature on recovery [1, 13] and previously introduced systems that employ fault tolerance mechanisms [9] and local recovery [6, 10]. The recovery style is a specialization of the module viewpoint [17]. First, the properties and elements of the style are introduced with related definitions and a visual notation. Then, we introduce the local recovery style, which is a specialization of the recovery style.

4.1. Basic Definitions and Properties

The key elements and relations of the recovery style are listed below. A visual notation based on UML [18] is shown in Figure 3 that can be used to document a recovery view.

- **Elements:** recoverable unit (RU), non-recoverable unit (NRU)
- **Relations:** applies-recovery-action-to, conveys-information-to
- **Properties of elements:** Properties of RU: set of system modules together with their functional importance and reliability, types of errors that can be detected, supported recovery actions, type of isolation (process, exception handling, etc.). Properties of NRU: types of errors that can be detected (i.e. monitoring capabilities).
- **Properties of relations:** Type of communication (synchronous/asynchronous) and timing constraints if there are any.

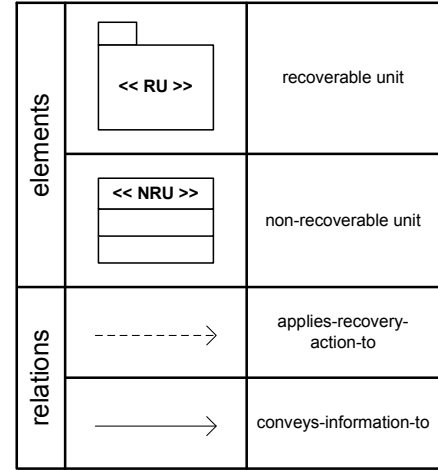


Figure 3. Recovery style notation

- **Topology:** The target of a *applies-recovery-action-to* relation can only be a *RU*.

The recovery style introduces two types of elements; *RU* and *NRU*. A *RU* is a unit of recovery in the system, which wraps a set of modules and isolates them from the rest of the system. It provides interfaces for conveying information about detected errors and responding to triggered recovery actions. A *RU* has the ability to recover independently from other *RU* and *NRUs* it is connected to. A *NRU*, on the other hand, can not be recovered independent from other *RU* and *NRUs*. It can only be recovered together with all other connected elements. This can happen in the case of global recovery or a recovery mechanism at a higher level in the case of a hierarchic decomposition.

conveys-information-to relation is about communication between elements for guiding and coordinating the recovery actions. Conveyed information can be related to detected errors, diagnosis results or a chosen recovery strategy. *applies-recovery-action-to* relation is about the control imposed to a *RU* and it affects the functioning of the target *RU* (suspend, kill, restart, roll-back, etc.)

4.2. Local Recovery Style

In the following, we introduce the local recovery style, in which the elements and relations of the recovery style are further specialized. Figure 4 provides a notation for specialized elements and relations based on UML.

- **Elements:** *RU*, recovery manager, communication manager
- **Relations:** restarts, kills, notifies-error-to, provides-diagnosis-to, sends-queued message-to, provides-synch. info-to

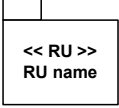
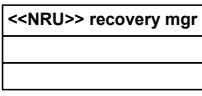
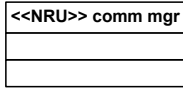
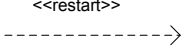
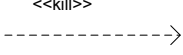
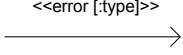
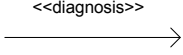
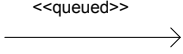
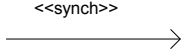
elements		recoverable unit
		recovery manager
		communication manager
relations		restarts
		kills
		notifies-error-to
		provides-diagnosis-to
		sends-queued message-to
		provides-synch. info-to

Figure 4. Local recovery style notation

- *Properties of elements:* Each RU has a *name* property in addition to those defined in the recoverability viewpoint.
- *Properties of relations:* *notifies-error-to* relation has a *type* property to indicate the error type if multiple types of errors can be detected.
- *Topology:* *restarts* and *kills* relations can be only from a *recovery manager* to a RU. *sends-queued message-to* can be between a RU and a communication manager.

Communication manager connects RUs together, routes messages and informs connected elements about the availability of another connected element. *Recovery manager* applies recovery actions on RUs.

Figure 5 depicts a simple instance of the local recovery style with one communication manager, one recovery manager and two RUs, A and B. Errors are detected by the communication manager and notified to the recovery manager. Recovery manager restarts A and/or B. B provides synchro-

nization information to A after A is restarted so that it can re-synchronize with the system. The messages that are sent from the communication manager to A are stored in a queue by the communication manager while A is unavailable (i.e. being restarted) and sent (retried) after A becomes available.

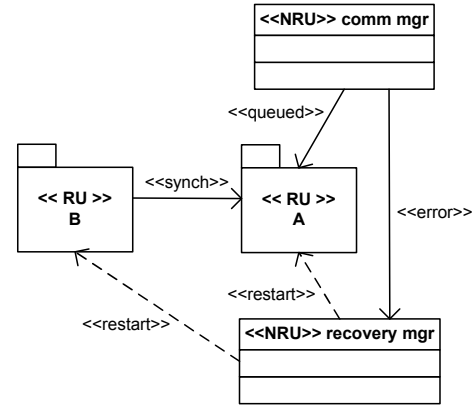


Figure 5. A simple recovery view based on the local recovery style (KEY: Figure 4)

5. Utilization of the Recovery Style

Our aim is to utilize the recovery style for MPlayer (Section 3.1) to choose and realize a recovery design among several design alternatives. One such alternative is to introduce a local recovery mechanism comprising 3 RUs; 1) *RU AUDIO*, which provides the functionality of *Libao 2*) *RU GUI*, which encapsulates the *Gui* functionality and 3) *RU MP-CORE* which comprises the rest of the system. Figure 6 depicts the boundaries of these RUs, which are overlayed on the module view of the MPlayer software architecture. In Figure 7(b) the recovery design corresponding to this RU selection is shown. Here, we can also see two new architectural elements that are not recoverable; a *communication manager* that mediates and controls the communication between the RUs and a *recovery manager* that applies the recovery actions on RUs.

Note that Figure 7(b) shows just one possible design for the recovery mechanism to be introduced to the MPlayer. We could consider many other design alternatives. One alternative would be to have a global recovery (See Figure 7(a)). We could also have more than 3 recoverable units (See Figure 7(c)). On the other hand, there could be more than one element that controls the recovery actions and communication (e.g. distributed or hierarchical control). These elements could be recoverable units as well.

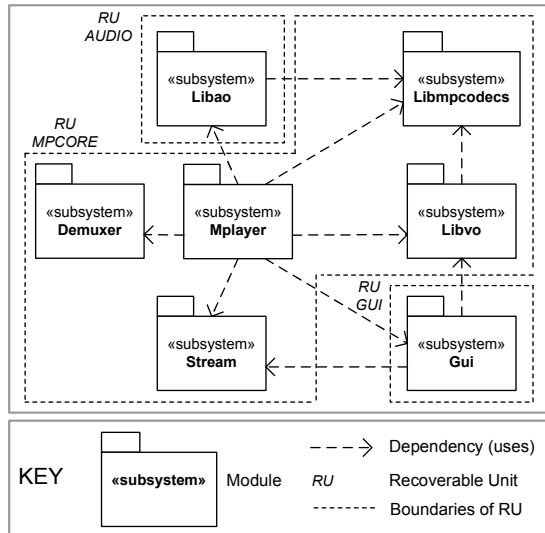
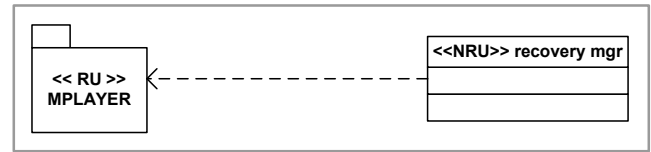


Figure 6. The Module View of the MPlayer Software Architecture with the Boundaries of the Recoverable Units

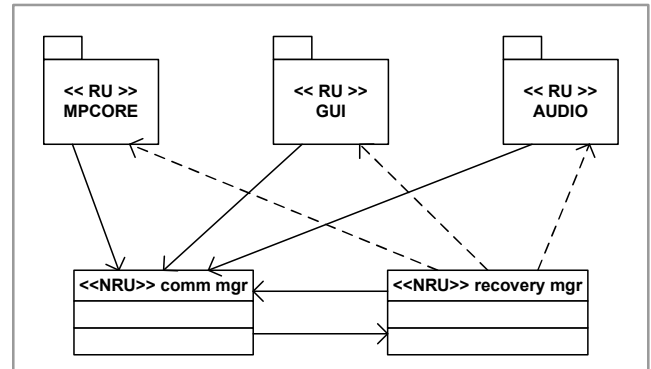
5.1. Analysis based on the Recovery Style

In principle, each style supports a set of analysis. In this section, we present analysis alternatives that can be performed based on the recovery style. As mentioned in the previous section, there are several design choices for a recovery mechanism. The impact of these choices can be analyzed to evaluate design alternatives and do the selection accordingly. As one of the most important design decisions, if not the most important one, it has been decided to split MPlayer into three RUs. There are several other alternatives for the selection of RUs. At one extreme, we could have only 1 RU, which means we do not do any splitting and apply global recovery (See Figure 7(a)). At the other extreme, we could decide to split each and every module of the system and encapsulate them in separate RUs (See Figure 7(c)).

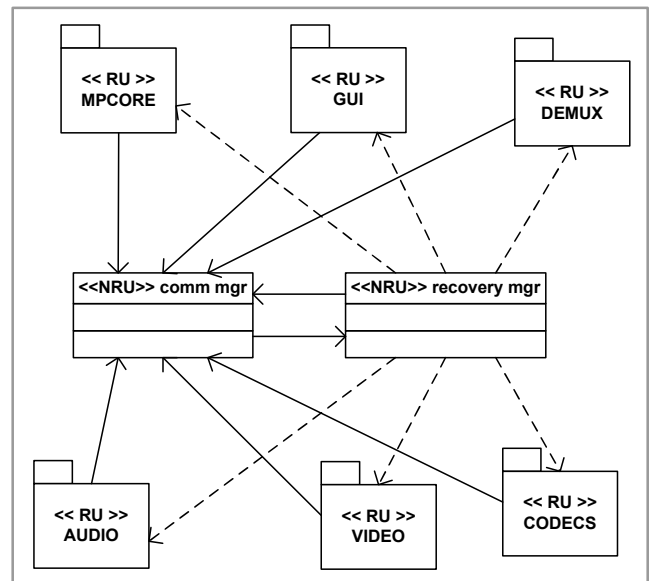
These alternatives have both pros and cons. The more we split the MPlayer and decrease the granularity of recovery, the more we gain with respect to the availability of the system. The probability that all RUs fail at the same time decreases by increasing the number of RUs. Thus, some functionality will always be available to the user during recovery of the system. On the other hand, all function calls that pass the boundaries of RUs are captured and directed through Inter-Process Communication (IPC) calls. This results in a performance overhead depending on the amount of interactions between the chosen RU boundaries. As a result, availability during recovery time and performance during operational time turns out to be conflicting factors



(a) global recovery



(b) local recovery with 3 RUs



(c) local recovery with 7 RUs

Figure 7. Architectural alternatives for the recovery design of MPlayer software (KEY: Figure 3)

Table 1. Comparison of recovery design alternatives

recovery design	global recovery	recovery with 3 RUs	recovery with 7 RUs
PO	0	0,8	4,8
PF_S	0,63	0,2	0

leading to a trade-off.

Performance overhead: To measure this overhead we run the original MPlayer software with a profiler (GNU *gprof*) to derive the function control flow together with information regarding the frequency of function calls. We have developed an analysis tool, *Module Dependency Graph (MDG) Analyzer*, which parses the output of *gprof* and *nm* (a GNU tool that lists the symbols - including function names - defined in object files). Based on the parsed data, MDG Analyzer creates a MDG, which comprises a module control flow graph (each module is an object file). This information is stored in a database so that interactions between a set of modules with the rest of the system can be queried. We queried this information for the modules that are part of the chosen RUs. We calculate the performance overhead with the following metric.

$$PO = \frac{\sum_{r \in RU} \sum_{f \in F_r} \sum_{\substack{h \in F_S \wedge \\ h \notin F_r}} calls(h \rightarrow f) \times t_{IPC}}{\sum_{f \in F_S} \sum_{h \in F_S} calls(h \rightarrow f) \times time(f)} \times 100 \quad (1)$$

In Equation 1, F_r denotes the set of functions provided in RU r and F_S denotes the set of functions in the whole system. Thus, $\bigcup_{r \in RU} F_r = F_S$. In the denominator of Equation 1 we sum up for all function pairs in the system, the number of times a function is called by another function multiplied by the average time (in *ms*) spent in that function. In the nominator we sum up for all RUs and for all function calls to a RU from outside the boundaries of that RU, the number of calls to a function multiplied by the inter-process communication overhead (t_{IPC}). We multiply the result with 100 to obtain the percentage. Based on our measurements we fixed t_{IPC} to a worst case value (100*ms*). PO for three alternatives can be seen at the first row of Table 1. Here, we see that PO for global recovery is 0% since we do not do any splitting. Local recovery with 3 RUs and 7 RUs lead to PO of 0,8% and 4,8%, respectively.

Availability: As the other trade-off factor, availability is traditionally measured as the percentage of time that the system is available. System up-time is considered as a binary property (The system is either up or down). Degraded system states and partial availability introduced by local re-

covery is not taken into account by this metric. Our aim is to analyze the degree of availability of the system during its recovery. We want this degree at least to be greater than 0 meaning that some parts of the system will always be available even if the system is being recovered. That means that not all RUs of the system should fail at the same time. This can be calculated as the probability of system failure as follows.

$$PF_{RU}(r) = \sum_{i=0}^{n_r} (-1)^{n_r} \times (n_r - i) \times PF_m^i \quad (2)$$

$$PF_S = \prod_{r \in RU} PF_{RU}(r) \quad (3)$$

In Equation 2 above the assumption is that RUs encapsulate a set of type of errors to provide isolation and therefore their failures are independent events. For that reason, the probability of system failure, PF_S , is equal to the multiplication of probabilities of failure for each RU (PF_{RU}). Failure of a RU, on the other hand, is a result of joint probability of failures of modules (PF_m) comprised by the RU. n_r denotes the number of modules comprised by RU r . In this paper, we assume equal failure probabilities for modules and fix PF_m to (0.1). Based on these assumptions, approximate PF_S values for three alternatives can be seen at the second row of Table 1. Here, we see that the probability that the whole system will be unavailable, PF_S is 0,63 for global recovery. It decreases to 0,2 with 3 RUs and approximately 0 for local recovery with 7 RUs.

5.2. Realization of the Local Recovery

We have decided to realize the recovery design presented in Figure 7(b) for MPlayer. Figure 8 depicts a more detailed design based on the local recovery style notation (See Figure 4). Here, we can see the communication manager, the recovery manager and the three RUs, MPCORE, GUI and AUDIO. MPCORE provides information to GUI for synchronization after GUI is restarted. Each RU can detect deadlock errors. The recovery manager can detect fatal errors. All error notifications are send to the communication manager, which comprises the diagnosis facility. Diagnosis information is conveyed to the recovery manager, which kills a set of RUs and/or restarts a dead RU. Messages that are sent from RUs to the communication manager 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.

We have implemented a reusable *fault tolerance library (Libft)*, which comprises IPC communication utilities, serialization/deserialization primitives, error detection and diagnosis mechanisms, a RU wrapper template, a recovery

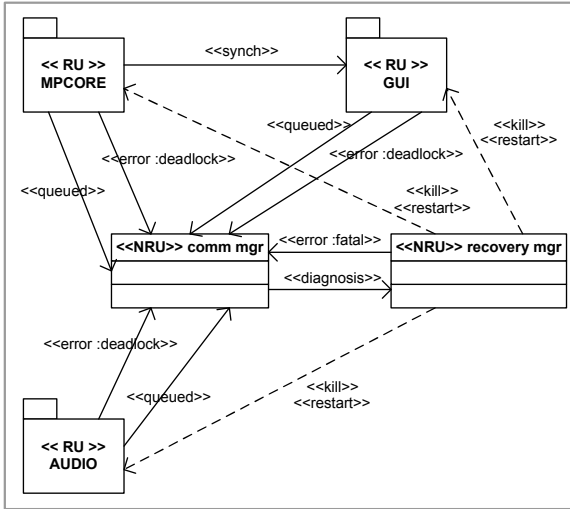


Figure 8. Realized MPlayer Software Architecture with 3 RUs (KEY: Figure 4)

manager and a communication manager. The library is developed in C language for Linux platform and it is used to realize the recovery view presented in Figure 8. *Libft* currently supports the detection of two error types: 1) Fatal error: The recovery manager detects if a process is dead¹ 2) Deadlock: A RU wrapper detects if an expected response to a message is not received within a period of time (The timeout period can be configured). The detected errors are reported to the communication manager, which employs the diagnosis facility.

Each RU is wrapped with a template as shown in Figure 9 so that it runs on a separate process at run-time. A set of state variables can be declared and they can be preserved after recovery. If needed, cleanup specific to the RU (i.e. allocated resources) can be specified.

Each RU provides an interface, through which it accepts serialized function calls together with its arguments. On reception of these calls, the corresponding functions are called and then the results are returned (See Figure 9). In all other RUs where this function is declared, function calls are redirected through IPC to the corresponding interface with C MACRO definitions (See Figure 10).

In Figure 10 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*, which activates the corresponding interface (*INTERFACE_GUI*) instead of performing the function call.

¹The recovery manager is the parent process of all RUs and receives and handles a signal when a child process is dead.

```
#include "util.h"
#include "recunit.h"
#include "rugui.h"
...
#define STATE_VARS ... &guiIntfStruct
...
void cleanUp()
{
    /* no component specific cleanup */
}
void __ruGui(struct recunit info)
{
    INIT_RU(SOCK_PATH_RECMGR, SOCK_PATH_CONN)
    ...
    PRESERVE_STATE
    ...
    processMsgs();
}
void catchInterfaces()
{
    BEGIN
        CATCH(INTERFACE_GUI, apOnMsgRcvd_gui)
    END
}
void OnMsgRcvd_gui_guiInit()
{
    guiInit();
    RETURN(INTERFACE_GUI, msg_gui_guiInit)
}
...
```

Figure 9. RU Wrapper code for RU AUDIO

6. A Method for the Realization of the Local Recovery Style

In the previous section, we have showed the utilization of the local recovery style to document, analyze and realize a local recovery design. To refactor an existing system to make it support local recovery involves several steps. In this section, we introduce a generic method, which is supported by the analysis tool (MDG Analyzer) and the reusable library (*Libft*) introduced before. Figure 11 shows the fundamental steps of the method.

The first step is *Analysis* where the basic design choices are made. The information required as an input to this phase

```
#define guiInit() mpcore_gui_guiInit()
...
void mpcore_gui_guiInit()
{
    CALL(INTERFACE_GUI, msg_gui_guiInit)
}
...
```

Figure 10. Function indirection through RU interfaces

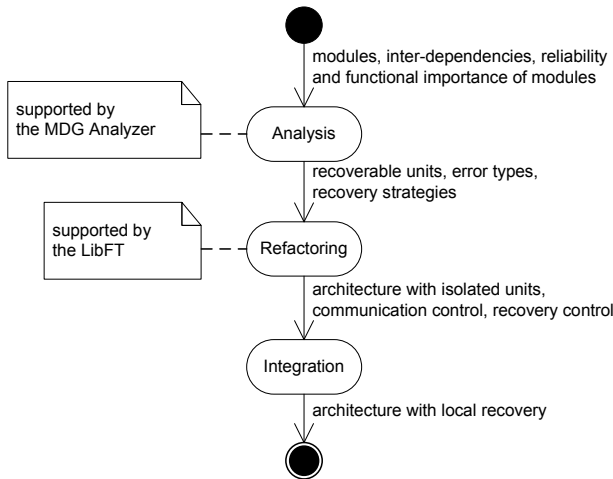


Figure 11. The basic steps of the method for realizing the local recovery style

includes; modules of the system, their interdependencies, relative reliability of the modules, an error model and functional importance of the modules. Modules that are important in terms of the functionality they provide should be isolated from the rest of the system. On the other hand, modules that are not reliable should be isolated as well. For instance, 3rd party software can be isolated to prevent its undesired effects on the core system functions. Types of errors that are considered for recovery are also important and guide the decomposition for recovery. Performance overhead of the decomposition should also be analyzed, for which tools can be utilized like the MDG Analyzer as introduced in the previous section.

At the end of the Analysis step, boundaries of recoverable units, error types and related recovery strategies should be known, which are provided as an input for the *Refactoring* step. In this step, the architecture is decomposed according to the boundaries chosen for recoverable units. Communication control and recovery coordination mechanisms are included based on the error types and recovery strategies to be applied. At this step, related architectural elements and wrappers can be re-used by configuring them to the application-specific needs, error types and recovery actions. For this purpose, a library like *Libft* can be utilized as introduced in the previous section.

The final step is *Integration*, where RUs, communication manager(s) and recovery manager(s) are integrated. This requires the consideration of basic connections, the synchronization between RUs and combination of error detection and diagnosis mechanisms with the recovery mechanism. As a result, the system has the local recovery abilities for chosen error types and recoverable unit boundaries.

7. Related Work

Architectural tactics [2] aim at identifying architectural decisions related to a quality attribute requirement and composing these into an architecture design. The recovery style is used for representing and analyzing architectural tactics for recovery. It provides a view of the system, by means of which the recovery design of a system can be captured.

Perspectives [20] guide an architect to modify a set of existing views to document and analyze quality properties. Our work addresses the same problem for recovery by introducing the recovery style, which is instantiated as a view. One of the motivations for using perspectives instead of creating a quality-based view is to avoid the duplicated information. This was less an issue in our project context, where we needed an explicit view for recovery to depict the related decomposition, dedicated architectural elements with complex interactions among them (e.g. Figure 8).

Idealized fault tolerant element [3] makes exception handling capabilities explicit at the architectural level. This approach is a way to reason about and analyze recovery properties of a system based on its architecture design.

Other than the solution proposed in this paper, there are several works that apply local recovery in different contexts. They can be represented with the local recovery style as well. For example, in [10] device drivers are executed on separate processes at user space and microreboot [6] is applied to increase the failure resilience of operating systems. They provide a view of the architecture of the operating system, where the architectural elements related to failure resilience and corresponding relations are shown. According to this view *Process Manager*, which restarts the processes, is a non-recoverable unit that coordinates the recovery actions. *Reincarnation Server* monitors the system to facilitate error detection and diagnosis and it guides the recovery procedure. *Data Store*, which is a name server for interconnected components, mediates and controls communication. The view also utilizes relations for recovery actions and error notification.

A generic run time adaptation framework is presented in [7] together with its specialization and realization for performance adaptation. According to the concepts in this framework, our *monitoring mechanisms* are error detectors. Diagnosis facilities can be considered as *gauges*, which interpret monitored low-level events, and recovery actions can be considered as *architectural repairs*. As a difference from the approach presented in [7], we propose a style specific for fault tolerance and reliability.

Several software architecture analysis approaches have been introduced for addressing quality properties. They usually perform either static analysis of formal architectural models [15] or a set of scenario-based architecture analysis methods as described in [4] are applied. The goal of these

approaches is to assess whether or not a given architecture design satisfies desired quality requirements. The main aim of the recovery style that we have introduced, on the other hand, is to communicate and support the design of recovery to enhance reliability. Analysis is a complementary work to make trade-off decisions, tune and select recovery design alternatives.

8. Conclusion and Future Work

We have observed that designing a system with recovery imposes requirements at the architectural level and it results in several new elements, complex interactions and even a particular decomposition for error containment. Existing viewpoints mostly capture functional aspects of a system and they are limited for explicitly representing elements, interactions and design choices related to recovery.

In this paper, we introduced the recovery style to document and analyze recovery properties of a software architecture. The recovery style is a specialization of the module viewtype. We also introduced a local recovery style and illustrated its application on the open source software, MPlayer. Further, a generic method is presented for introducing local recovery to an existing system. Recovery views of MPlayer based on the recovery style have been used within our project to communicate the design of the *Libft* and its application to MPlayer. These views have formed the basis for analysis and support for the detailed design of the recovery mechanisms. Using the views and analysis we could derive the design that is most feasible with respect to recovery.

As a future work, we will enhance our analysis metrics and develop additional analysis tools to analyze several properties related to recovery based on a recovery view.

Acknowledgments

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.

References

- [1] A. Avizienis et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.
- [2] F. Bachmann, L. Bass, and M. Klein. Architectural tactics: A step toward methodical architectural design. Technical Report CMU/SEI-2003-TR-004, Pittsburgh, PA, 2003.

- [3] R. de Lemos, P. Guerra, and C. Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, Mar/Apr 2004.
- [4] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Trans. Software Eng.*, 28(7):638–654, 2002.
- [5] G. C. Hunt et al. Sealing OS processes to improve dependability and safety. *SIGOPS Oper. Syst. Rev.*, 41(3):341–354, 2007.
- [6] G. Candea et al. Microreboot: A technique for cheap recovery. In *OSDI*, San Francisco, CA, 2004.
- [7] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In A. R. R. de Lemos, C. Gacek, editor, *Architecting Dependable Systems*. Springer-Verlag, 2003.
- [8] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, NJ, USA.
- [9] 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.
- [10] J.N. Herder et al. Failure resilience for device drivers. In *DSN'07*, pages 41–50, Washington, DC, USA, 2007.
- [11] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.
- [12] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, Boston, MA, USA, 2000.
- [13] M. Elnozahy et al. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, 1996.
- [14] M. W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 34(4):107–109, 2001.
- [15] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.
- [16] MPlayer official website, 2007. <http://www.mplayerhq.hu/>.
- [17] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, September 2002.
- [18] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, 1999.
- [19] Trader project, ESI, 2007. <http://www.esi.nl>.
- [20] E. Woods and N. Rozanski. Using architectural perspectives. In *WICSA '05*, pages 25–35, Los Alamitos, CA, USA, 2005.