Software Qual J (2013) 21:203–240
DOI 10.1007/s11219-011-9171-6

Optimizing decomposition of software architecture for local recovery

Hasan Sözer · Bedir Tekinerdoğan · Mehmet Akşit

Published online: 30 November 2011

Abstract The increasing size and complexity of software systems has led to an amplified number of potential failures and as such makes it harder to ensure software reliability. Since it is usually hard to prevent all the failures, fault tolerance techniques have become more important. An essential element of fault tolerance is the recovery from failures. Local recovery is an effective approach whereby only the erroneous parts of the system are recovered while the other parts remain available. For achieving local recovery, the architecture needs to be decomposed into separate units that can be recovered in isolation. Usually, there are many different alternative ways to decompose the system into recoverable units. It appears that each of these decomposition alternatives performs differently with respect to availability and performance metrics. We propose a systematic approach dedicated to optimizing the decomposition of software architecture for local recovery. The approach provides systematic guidelines to depict the design space of the possible decomposition alternatives, to reduce the design space with respect to domain and stakeholder constraints and to balance the feasible alternatives with respect to availability and performance. The approach is supported by an integrated set of tools and illustrated for the open-source MPlayer software.

This work has been carried out as part of the TRADER project (TRADER 2011) under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

H. Sözer (✉)
Department of Computer Science, Özyeğin University, İstanbul, Turkey
e-mail: hasan.sozer@ozyegin.edu.tr

B. Tekinerdoğan
Department of Computer Engineering, Bilkent University, Ankara, Turkey
e-mail: bedir@cs.bilkent.edu.tr

M. Akşit
Department of Computer Science, University of Twente, Enschede, The Netherlands
e-mail: aksit@cs.utwente.nl

Keywords Software architecture design · Fault tolerance · Local recovery · Availability · Performance

1 Introduction

The increasing size and complexity of software has led to an amplified number of potential failures that can occur in the system. To detect and recover from the potential failures, appropriate reliability techniques must be applied. Reliability techniques can be based on fault prevention, fault tolerance, fault removal or fault forecasting (Avizienis et al. 2004). Fault tolerance is a technique for ensuring the system's reliability even if faults remain and they get activated. Since it is usually impossible to prevent all the failures, fault tolerance techniques have now become more important.

The size and complexity of software systems have not only led to an increased number of potential failures but also affected the type of faults that can occur. Traditionally, software faults and the resulting errors have been assumed to be permanent. As a result, fault tolerance techniques have been mainly based on replication and design diversity e.g., recovery blocks, N-version programming (Laprie et al. 1995). We can now observe that software systems are exposed to increasing number of so-called *transient errors* (Huang and Kintala 1995). These errors are mostly triggered by peak conditions in workload and timing issues that could not have been anticipated before. To recover from these errors, usually *restart* protocols are applied that initializes the software to a consistent starting state (Huang and Kintala 1995). In this work, we only consider crash failures caused by transient errors, and we assume in our analysis that recovery based on restart protocols always succeeds.

Restarting can be applied at different granularity levels of the system. Very often, restarting is applied to the whole system whereby all the components are restarted together. Unfortunately, this makes the system completely unavailable until it is in normal operational mode again. For increasing availability, *local recovery* (Candea et al. 2004b) can be applied to only the components of the system that are affected by the corresponding error. In this way, only the erroneous components need to be restarted and the other components can remain available.

For achieving local recovery, the corresponding system modules need to be decomposed into a set of isolated *recoverable units* (RU) that can be independently restarted. An RU defines a unit of recovery in the system, which wraps a set of modules and isolates them from the rest of the system. There is no (necessarily) direct mapping between an RU and a function/service. The mapping depends on the grouping of modules, i.e., the RU decomposition. This decomposition can be done in different ways, and each decomposition alternative will lead to a different availability degree. Obviously, higher number of RUs can result in a higher system availability. However, increasing the number of RUs will also lead to a higher coupling among the isolated modules and as such introduce an additional performance overhead. On the other hand, keeping the modules together in one RU will increase the performance but will result in a lower availability since the failure of one module will affect the others as well. As a result, for selecting a decomposition alternative, we have to cope with a trade-off between *availability* and *performance*.

In this paper, we propose a dedicated approach for analyzing an existing system to decompose its software architecture for introducing local recovery. There exist many large and complex legacy systems that were designed without fault tolerance in mind. Usually, it is not feasible to develop these systems from scratch. Introducing fault tolerance to these systems and refactoring them for local recovery also require substantial development and maintenance effort. Therefore, it is important to analyze design alternatives beforehand.

It appears that the number of decomposition alternatives increases exponentially with respect to the number of modules in the system. This makes it infeasible to analyze design alternatives manually, involving the (1) derivation of possible design alternatives, (2) evaluation of various qualities per alternative and (3) calculation of the optimal alternative. These activities require the utilization of algorithmic optimization techniques and dedicated tool support. Algorithmic solutions may become computationally extensive. Nevertheless, they are still useful in practice for the following reasons: (1) state-of-the-art multiobjective optimization techniques are able to evaluate dozens of thousands of solution alternatives in the order of milliseconds (Meedeniya et al. 2011), (2) the complexity can be reduced by offering tool support that allows the designers to interactively exclude alternatives that are considered less relevant and (3) the architects may carry out what-if analysis by applying the techniques to selected parts of the architecture. The tool may provide confidence and a formal basis to justify why certain architectural design decision has been made.

In this work, our main contribution is a systematic methodology that combines several analysis techniques to (1) depict the design space of the possible decomposition alternatives, (2) reduce the design space with respect to domain and stakeholder constraints and (3) balance the feasible alternatives with respect to availability and performance. Throughout this process, the designer is supported by a set of analysis tools integrated in the Arch-Studio environment (Dashofy et al. 2002). We utilize optimization techniques for evaluating large design spaces, and we provide heuristics as a selection criterion that evaluates the availability of the (decomposed) system. The approach is illustrated for evaluating decomposition alternatives to introduce local recovery to the open-source media player, called MPlayer.

The remainder of this paper is organized as follows. In Sect. 2, we define the problem statement for selecting feasible decomposition alternatives. In Sect. 3, we present the top-level process of our approach. Sections 4 to 8 explain the steps of the top-level process in detail. In Sect. 9, we introduce the set of tools that support this process. In Sect. 10, we evaluate our approach based on real-time measurements from systems that are decomposed for local recovery. In Sect. 11, we discuss possible limitations, extensions and lessons learned. In Sect. 12, we summarize the related work. Finally, in Sect. 13, we provide the conclusions.

2 Problem statement

In the following subsection, we first present a case study and show an example decomposition for introducing local recovery for a given architecture. This case study will be used throughout the paper to illustrate our approach. Next, we will discuss the design space for the decomposition alternatives. Finally, we will present the criteria to be considered for selecting the optimal alternative in the design space.

2.1 Case study: MPlayer

We have applied local recovery to an open-source software, MPlayer (2010). MPlayer 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 complied on Linux platform (Ubuntu version 7.04). Figure 1 presents a simplified module view (Clements et al. 2002a) 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 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. *Mplayer* includes the *main* function of the application, which initializes and coordinates all the other modules. For this reason, there is a dependency depicted from *Mplayer* to each and every other module.

An error occurring in one part of the system can propagate and lead to errors in other parts. To prevent the propagation of errors, the system should be decomposed into a set of isolated RUs (Hunt et al. 2007; Candea et al. 2004b). For example, one possible decomposition 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 module view of the MPlayer software architecture shown in Fig. 1.

2.2 Design space

Figure 2 shows only one alternative decomposition for isolating the recoverable units and as such introducing local recovery. Obviously, the partitioning can be done in many different ways. To reason about the number of decomposition alternatives, we first need to

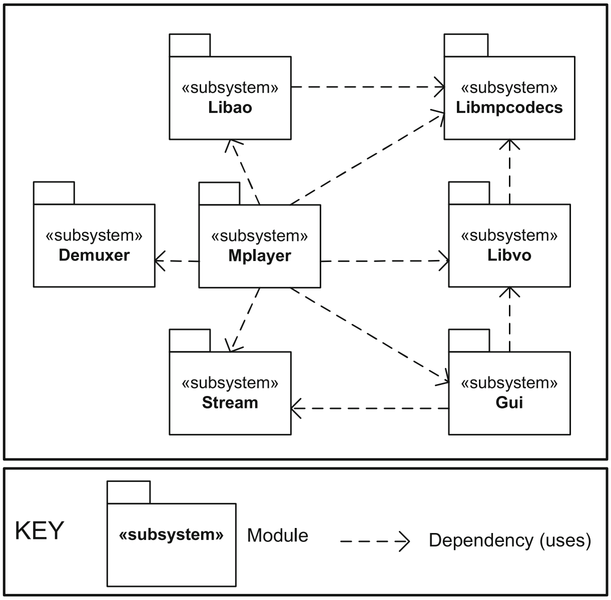


Fig. 1 A simplified module view of the MPlayer software architecture

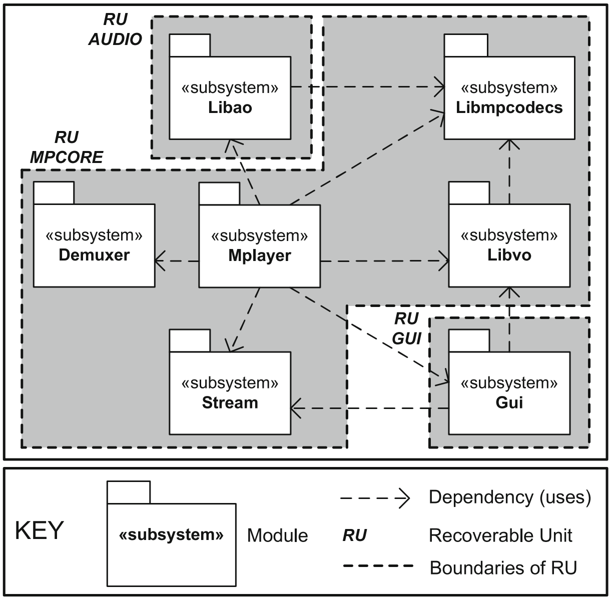


Fig. 2 The module view of the MPlayer software architecture with the boundaries of 3 Recoverable units (RUs)

model the design space. In fact, the partitioning of architecture into a set of RUs can be generalized to the well-known *set partitioning problem* (Harris et al. 2000). Hereby, a partition of a set S is a collection of disjoint subsets of S whose union is S. For example, there exists 5 alternatives to partition the set $\{a, b, c\}$. These alternative are: $\{\{a\}, \{b\}, \{c\}\}$, $\{\{a\}, \{b, c\}\}$, $\{\{b\}, \{a, c\}\}$, $\{\{c\}, \{a, b\}\}$, $\{\{a, b, c\}\}$. The number of ways to partition a set of n elements into k nonempty subsets is computed by the so-called *Stirling numbers of the second kind*, $S(n, k)$ (Harris et al. 2000). It is calculated with the following recursive formula.

$$S(n,k) = k \times S(n-1,k) + S(n-1,k-1), n \geq 1 \tag{1}$$

The total number of ways to partition a set of n elements into arbitrary number of nonempty sets is counted by the nth *Bell number* as follows.

$$B_n = \sum_{k=1}^{n} S(n,k) \tag{2}$$

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), $B_{15} = 1.382.958.545$. Therefore, searching for a feasible design alternative becomes quickly problematic as n (i.e., the number of modules) grows.

2.3 Criteria for selecting decomposition alternatives

Each decomposition alternative in the large design space may have both pros and cons. Therefore, the selection of a particular decomposition among the feasible alternatives leads to a trade-off. Recovery techniques impact several quality attributes of the system. In this work, we consider two main attributes: *performance* and *availability*. Depending on the application domain, emphasis can be put on other (possibly conflicting) quality attributes as well. For example, an erroneous system might be stopped for safety, which in turn compromises its availability (Grunske et al. 2007). In the following, we outline the basic criteria that we consider for choosing a decomposition alternative.

- *Availability*: In fact, the main goal of local recovery is to keep the system available as much as possible. The total availability of a system depends on the availability of its individual recoverable units. Equation 3 shows the formula of availability.

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF+MTTR}} \tag{3}$$

  In Eq. 3, 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 values of MTTF and MTTR properties of the system depend on the decomposition alternative, that is, how we separate and isolate the modules into a set of RUs.
- *Performance*: Logically, the optimal availability of the system is defined by the decomposition alternative in which all the modules in the system are separately isolated into RUs. However, increasing the number of RUs leads to a performance overhead due to the dependencies between the separated modules in different RUs. We distinguish between two important types of dependencies that cause a performance overhead: (1) *function dependency* and (2) *data dependency*.
  The function dependency is the number of function calls between modules across different RUs. For transparent recovery, these function calls must be redirected. The redirection of calls leads to an additional performance overhead. For this reason, for selecting a decomposition alternative, we should consider the number of function calls among modules across different RUs.
  The data dependencies are proportional to the size of the shared variables among modules across different RUs. In previous work on local recovery (Candea et al. 2004b; Herder et al. 2007), 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 (Candea et al. 2004b) and stateless device drivers in (Herder et al. 2007). However, when an existing system is decomposed into RUs, there might be shared state variables leading to data dependencies between RUs. Obviously, the size of data dependencies complicates the recovery and create performance overhead because the shared data need to be kept synchronized after recovery. This makes the amount of data dependency between RUs an important criteria for selecting RUs.

It appears that there exists an inherent trade-off between the availability and performance criteria. On the one hand, increasing availability will require to increase the number of RUs. On the other hand, increasing the number of RUs will introduce additional performance overhead because the amount of function dependencies and data dependencies will increase. Therefore, selecting decomposition alternatives implies basically leveraging these two criteria.

If there are only a few alternatives, trade-off analysis can be carried out manually. However, complex systems usually lead to a large amount of alternatives with subtle differences that have large impact. In such a context, it is hard to manually identify the architectural decomposition that offers the best trade-off. For this reason, we utilize optimization techniques as part of our methodology and the related analysis tool. We define heuristics for the selection of a decomposition alternative, where availability is considered as the main decomposition criterion (See Sect. 8.1). In the following, we provide an overview of our approach that comprises systematic analysis for performance estimation, availability evaluation and decomposition alternative selection.

3 The overall process

In this section, we define the approach that we use for selecting a decomposition alternative with respect to local recovery requirements. Our main contribution is the systematic methodology used for evaluating and selecting decomposition alternatives. This involves the combination of several analysis techniques for quality estimation and optimization. Since it is not practical to apply these techniques manually, we also provide integrated tool support together with the methodology. In the following, we describe the overall process and the main activities. We present an overview of our analysis tool and its main components in Sect. 9.

3.1 The overall process

Figure 3 depicts the main activities of the overall process in a UML activity diagram.

The activities are categorized into 4 groups as follows.

- *Architecture Definition*: The activities of this group are about the definition of the software architecture by specifying basic modules of the system. The given architecture will be utilized to define the feasible decomposition alternatives that show the isolated RUs consisting of the modules. To prepare the analysis for finding the decomposition alternative each module in the architecture will be annotated with reliability properties. These properties are used as an input for heuristics during the *Decomposition Selection* process.
- *Constraint Definition*: Using the initial architecture, we can in principle define the design space including the decomposition alternatives. As we have seen in Sect. 2.2, the design space can easily get very large and unmanageable. In addition, not all theoretically possible decompositions are practically possible because modules in the initial architecture cannot be freely allocated to RUs due to domain and stakeholder constraints. Therefore, before generating the complete design space first the constraints will be specified. The activities of the *Constraint Definition* group involve the specification of such domain and stakeholder constraints.
- *Measurement*: Even though some alternatives are possible after considering the domain constraints, they might in the end still not be feasible due to the performance overhead introduced by the particular allocation of modules to RUs. To estimate the real performance overhead of decomposition alternatives, we need to know the amount of function and data dependencies between the system modules for a running system. The activities of the *Measurement* group deal with performing related performance measurements from the existing running system. These measurements are utilized during the *Decomposition Selection* process.
- *Decomposition Selection*: Based on the input of all the other groups of activities, the activities of this group select an alternative decomposition. First, the feasible design space is defined based on the architecture description, specified constraints and measurements in the previous activities. Next, a set of optimization algorithms and related heuristics are used for selecting a decomposition alternative.

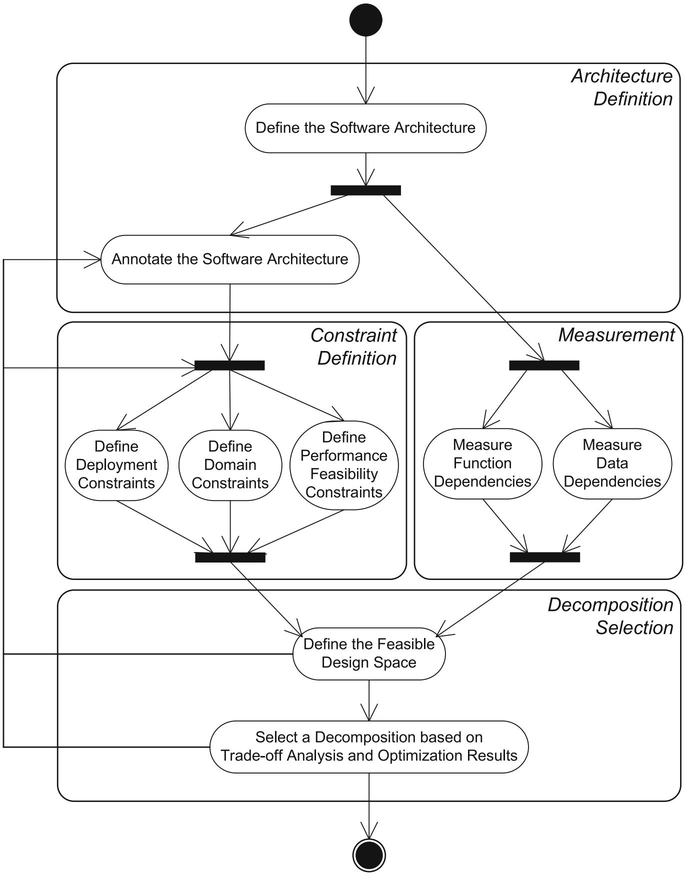

Architecture Definition
Define the Software Architecture
Annotate the Software Architecture
Constraint Definition
Measurement
Define Deployment Constraints
Define Domain Constraints
Define Performance Feasibility Constraints
Measure Function Dependencies
Measure Data Dependencies
Decomposition Selection
Define the Feasible Design Space
Select a Decomposition based on Trade-off Analysis and Optimization Results

Fig. 3 The overall process

In Fig. 3, there are flows depicted from the *Decomposition Selection* activities back to the *Constraint Definition* activities and the activity that involves the annotation of the software architecture. As such, design decisions can be revisited in several iterations; the designer can decide to refine the annotations and constraints to redefine the feasible design space. Hereby, the software architecture is given as an input, and we assume that it is correctly defined. The measurement activities are also solely based on this given architecture. For this reason, we have not considered iterations involving these activities.

In the following sections, we explain each activity of the proposed approach in detail, using the MPlayer case study as a running example.

4 Software architecture definition

The first step *Architecture Definition* in Fig. 3 is composed of two activities: *definition of the software architecture* and *annotation of the software architecture*. For describing the software architecture, we use the module view of the system, which includes basic implementation units and direct dependencies among them (Clements et al. 2002a). This

Fig. 4 xADL schema extension for specifying analytical parameters related to reliability

```
<xsd:complexType name="ReliabilityInterface">
<xsd:complexContent>
 <xsd:extension base="Interface">
  <xsd:sequence>
   <xsd:element name="MTTF"
                type="reliability:mttf"
                minOccurs="1" maxOccurs="1"/>
   <xsd:element name="MTTR"
                type="reliability:mttr"
                minOccurs="1" maxOccurs="1"/>
   <xsd:element name="Criticality"
                type="reliability:criticality"
                minOccurs="1" maxOccurs="1"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

activity is carried out by using the *Arch-Edit* tool of the *Arch-Studio* (Dashofy et al. 2002). *Arch-Studio* specifies the software architecture with an XML-based architecture description language (ADL)[1] called *xADL* (Dashofy et al. 2002). *Arch-Edit* is a front-end that provides a graphical user interface for creating and modifying an architecture description. Both the XML structure of the stored architectural description and the user interface of *Arch-Edit* (i.e., the set of properties that can be specified per module) conform to the xADL schema and as such can be used together with other XML-based tools.

In the second activity of *Architecture Definition*, a set of reliability properties per module is defined to prepare the subsequent analysis. These properties are *MTTF*, *MTTR* and *Criticality*. We have already explained *MTTF* and *MTTR* in Sect. 2.3. *Criticality* property defines how critical the failure of a module is with respect to the overall functioning of the system.

To be able to specify these properties, we have extended the existing xADL schema with a new type of interface, i.e., reliability interface, for modules. A part of the corresponding XML schema that introduces the new properties is shown in Fig. 4. The concept of analytical interface has been borrowed from (Grassi et al. 2005), where Grassi et al. define an extension to the xADL language to be able to specify parameters for performance analysis. They introduce a new type of interface that comprises two types of parameters: *constructive parameter* and *analytic parameter*. The analytic parameters are used for analysis purposes only. In our case, we have specified the reliability interface that consists of the three parameters *MTTF*, *MTTR* and *Criticality*. Using *Arch-Edit*, both the architecture and the properties per module can be defined. The modules together with their parameters are provided as an input to the heuristics that are used in the *Decomposition Selection* process.

Note that the values for properties MTTF, MTTR and Criticality need to be defined by the software architect. In principle, there are three strategies that can be used to determine these property values:

[1] In principle, UML or any ADL can be used for architecture definition. We have used an XML-based ADL for its extensibility capabilities and to be able to utilize the tool support (ArchStudio), which is also highly extensible for integrating our own tools. There exist also other extensible ADLs (di Ruscio et al. 2010) that could be utilized in our approach.

Table 1 Annotated *MTTR* values for the MPlayer case based on measurements

Modules	MTTR (ms)
Libao	480
Libmpcodecs	500
Demuxer	540
Mplayer	800
Libvo	400
Stream	400
Gui	600

- *Using Fixed values*: All values can be fixed to a certain value.
- *What-if analysis*: A range of values can be considered, where the values are varied and their effect is observed.
- *Measurement of actual values*: Values can be specified based on actual measurements from the existing running system. For example, the actual MTTF can be measured based on historical data or execution probabilities of elements that are obtained from scenario-based test runs (Yacoub et al. 2004).

The selection of these strategies is dependent on the available knowledge about the domain and the analyzed system. The measurement of actual values is usually the most accurate way to define the properties. However, this might not be possible due to lack of historical data. In that case, either fixed values or a what-if analysis or both can be used.

In our case study, we have fixed all the *Criticality* values to 1.[2] We have used different values for *MTTF* to observe and compare its effect on various design alternatives. We have specified *MTTR* values based on measurements from the existing running system. To compute the MTTR for the modules, we have measured the average time to restart and initialize each module of MPlayer. Table 1 shows the computed MTTR values.

5 Constraint definition

After defining the software architecture, we can start searching for the possible decomposition alternatives that define the structure of RUs. As stated before in Sect. 2.2, not all theoretically possible alternatives are practically possible. Moreover, system designers should be able to define their own RUs and/or enforce restrictions on the RU decomposition. For example, a set of critical functions or services might be isolated from the rest of the system for protection. Such decisions are made based on the domain knowledge of the designer, and constraints should be defined accordingly to steer the definition of RUs. During the *Constraint Definition* activity, the constraints are defined and as such the design space is reduced. We distinguish among the following three types of constraints:

- *Deployment Constraints*: An RU is a unit of recovery in the system and includes a set of modules. In general, the number of RUs that can be defined will also be dependent on the context of the system that can limit the number of RUs in the system. For example, if we employ multiple operating system processes to isolate RUs from each other, the number of processes that can be created can be limited due to operating

[2] In another study, we have worked on the derivation of *Criticality* values based on scenario-based analysis (Tekinerdogan et al. 2008).

system constraints. It might be the case that the resources are insufficient for creating a separate process for each RU, when there are many modules and all modules are separated from each other.

- *Domain Constraints*: Some modules can be required to be in the same RU, while other modules must be separated. In the latter case, for example, an untrusted third-party module can be required to be separated from the rest of the system. Usually, such constraints are specified with *mutex* and *require* relations (Kang et al. 1990).
- *Performance Feasibility Constraints*: As explained in Sect. 2.2, each decomposition alternative introduces a performance overhead due to function and data dependencies. Usually, there are thresholds for the acceptable amounts of these dependencies based on the available resources. The decomposition alternatives that exceed these thresholds are infeasible because of the performance overhead they introduce and as such must be eliminated.

6 Generation of design alternatives

The first activity in *Decomposition Selection* is to compute the size of the feasible design space based on the architecture description and the specified domain constraints. To this aim, the set of modules that are required to be together (based on the *requires* relations defined among the domain constraints) are grouped. In the rest of the activity, these groups of modules are treated as single modules. Then, based on the number of resulting modules and the deployment constraints (i.e., the number of RUs), *Stirling numbers of the second kind* (Sect. 2.2) is used for calculating the size of the reduced design space.

Next, *restricted growth (RG) strings* (Ruskey 2003) are used for generating the design alternatives. A RG string s of length n specifies the partitioning of n elements, where $s[i]$ defines the partition that ith element belongs to. For the MPlayer case, for instance, we can represent all possible decompositions with an RG string of length 7. The RG string 0000000 refers to the decomposition where all modules are placed in a single RU. Assume that the elements in the string correspond to the modules *Mplayer*, *Gui*, *Libao*, *Libmpcodecs*, *Demuxer*, *Stream* and *Libvo*, respectively. Then, the RG string 0120000 defines the decomposition that is shown in Fig. 2. Hereby, there are 3 RUs in total, enumerated as 0, 1 and 2. The modules *Mplayer*, *Libmpcodecs*, *Demuxer*, *Stream* and *Libvo* are placed in RU 0. The modules *Gui* and *Libao* are placed in RUs 1 and 2, respectively. A recursive lexicographic algorithm generates all valid RG strings for a given number of elements and partitions (Ruskey 1993).

During the generation process of decomposition alternatives, the alternatives that violate *mutex* relations (defined among the domain constraints) are eliminated. These are the alternatives, where two modules that must be separated are placed in the same RU. For the MPlayer case, there exist a total of 877 decomposition alternatives. The deployment and domain constraints that we have defined (See Fig. 13) reduced the number of alternatives down to 20.

7 Function and data dependency measurements from the existing system

The *Measurement* process follows the *Architecture Definition* (See Fig. 3) and the *Constraint Definition* processes. Using the domain constraints, the design space can already be generated. However, if performance feasibility constraints have been defined, then we can further reduce the design space. For this, we need to perform measurements from the existing running system. These measurements heavily depend on the usage scenarios and the inputs that are provided to the system. In our case study, we have performed the measurements for the video-playing scenario, which is a common usage scenario for a media player application. Elicitation of a representative set of usage scenarios is out of the scope of this paper. We refer to the related techniques in Sect. 11.

Based on the measurements, it can be decided whether design alternatives meet the *performance feasibility constraints* and as such are selected in the feasible set of alternatives. The *Measurement* process involves two activities for analyzing two different types of dependencies related to performance overhead: *function dependencies* and *data dependencies*. In the following subsections, we explain these activities in detail.

7.1 Function dependency analysis

Function calls among modules across different RUs impose an additional performance overhead due to the redirection of calls by RUs. In function dependency analysis, for each decomposition alternative, we analyze the frequency of function calls between the modules in different RUs. The overhead of a decomposition is calculated based on the ratio of the delayed calls to the total number of calls in the system.

Function dependency analysis is performed by first deriving the so-called module dependency graph (MDG) (Mitchell and Mancoridis 2006) of the program. MDG is a graph, where each node represents a C object file and edges represent the function dependencies between these files. Figure 5, for example, shows a partial snapshot of the generated MDG for MPlayer.[3]

After the MDG is created, it is stored in a database and queried for estimating the function dependency overhead. The estimation is performed based on the following equation.

$$fd = \frac{\sum_{RUx} \sum\limits_{\substack{RUy \wedge \\ (x \neq y)}} \mathrm{calls}(\mathrm{x} \rightarrow \mathrm{y}) \times t_{OV\,D}}{\sum_{f} \mathrm{calls}(\mathrm{f}) \times \mathrm{time}(\mathrm{f})} \times 100 \tag{4}$$

In Eq. 4, the denominator sums for all functions, the number of times a function is called multiplied by the time spent for that function. In the numerator, the number of times a function is called between different RUs is summed. The final result is multiplied by the overhead that is introduced per such function call ($t_{OV\ D}$). In our case study, where we isolate RUs with separate processes, the overhead is related to the interprocess communication.[4] For this reason, we use a measured worst-case overhead, 100 ms as $t_{OV\ D}$.

To calculate the function dependency overhead, we need to calculate the sum of function dependencies between all the modules that are part of different RUs. Consider, for example, the RU decomposition that is shown in Fig. 2. Hereby, the *Gui* and *Libao*

[3] The graphical representation of MDG is provided only for illustration purposes. This representation can be too complicated for manual interpretation, and as such, it is not exposed to the user. The module dependency data is stored in a database, and it is processed by a tool.

[4] In our implementation, we have used sockets for communication. The performance overhead is introduced mainly due to the marshalling of exchanged messages.

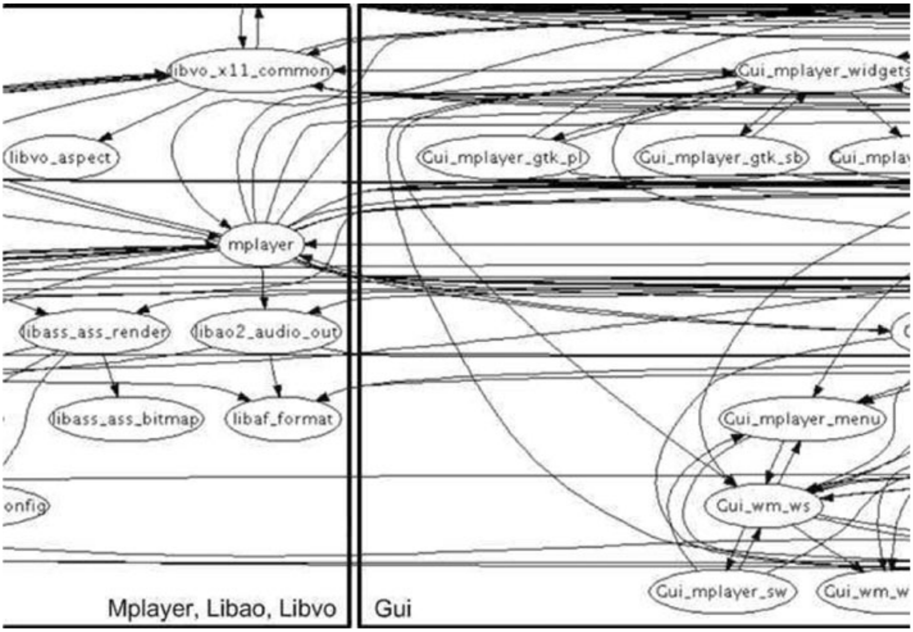


Fig. 5 A partial snapshot of the generated module dependency graph (MDG) of MPlayer together with the boundaries of the *Gui* module with the modules *Mplayer*, *Libao* and *Libvo*

Algorithm 1 Calculate the amount of function dependencies between the selected recoverable units (RUs)

```
$sum \leftarrow 0$
for all RU $x$ do
    for all Module $m \in x$ do
        for all RU $y \wedge x \neq y$ do
            for all Module $k \in y$ do
                $sum \leftarrow sum + noOfCalls(m, k)$
            end for
        end for
    end for
end for
```

modules are encapsulated in separate RUs and separated from all the other modules of the system. The other modules in the system are allocated to a third RU, *RU MPCORE*. For calculating the function dependency overhead of this example decomposition, we need to calculate the sum of function dependencies between the *Gui* module and all the other modules plus the function dependencies between the *Libao* module and all the other modules. In the following, Algorithm 1 shows the pseudo-code for counting the number of function calls between different RUs.

For the example decomposition shown in Fig. 2, the function dependency overhead was calculated as 5.4%. This makes the decomposition a feasible alternative with respect to the function dependency overhead constraints, where the threshold was specified as 15% for the case study. The calculation of the overhead by our tools (Sect. 9.3) took less than 100 ms for this example decomposition. In general, the time it takes for the calculation depends on the number of modules and the particular decomposition. The worst-case asymptotic complexity of Algorithm 1 is $O(n^2)$ with respect to the number of modules.

7.2 Data dependency analysis

Data dependency analysis is performed by instrumenting the program and identifying memory addresses that are shared by multiple modules at run time (the tool support for performing this identification is explained in Sect. 9.4). This information leads to the total number and size of data dependencies between the modules of the system. Table 2 shows the results for the MPlayer case.

In Table 2, we see per pair of modules, the number of common memory locations accessed (i.e., count) and the total size of the shared memory. This table is stored in a database and queried for each RU decomposition alternative to estimate the data dependency size. The size of the data dependency is calculated simply by summing up the shared data size between the modules of the selected RUs. This calculation is depicted in the following equation.

$$dd = \sum_{RUx} \sum_{\substack{RUy \wedge \\ (x \neq y)}} \sum_{m \in x} \sum_{k \in y} \text{memsize(m,k)} \tag{5}$$

For the example decomposition shown in Fig. 2, the data dependency size is approximately 5 KB. It took less than 100 ms to calculate this by our tools (Sect. 9.4). The decomposition also turns out to be a feasible alternative with respect to the data dependency size constraints, where the threshold was specified as 10 KB for the case study.

Table 2 Measured data dependencies between the modules of the MPlayer

Module 1	Module 2	Count	Size (bytes)
Gui	Libao	64	256
Gui	Libmpcodecs	360	1329
Gui	Libvo	591	2217
Gui	Demuxer	47	188
Gui	MPlayer	36	144
Gui	Stream	242	914
Libao	Libmpcodecs	78	328
Libao	Libvo	63	268
Libao	Demuxer	3	12
Libao	Mplayer	43	172
Libao	Stream	54	232
Libmpcodecs	Libvo	332	1344
Libmpcodecs	Demuxer	29	116
Libmpcodecs	Mplayer	101	408
Libmpcodecs	Stream	201	812
Libvo	Demuxer	53	212
Libvo	Mplayer	76	304
Libvo	Stream	246	995
Demuxer	Mplayer	0	0
Demuxer	Stream	23	92
Mplayer	Stream	28	116

7.3 Depicting function and data dependency analysis results

Using the deployment and domain constraints, we have seen that for the MPlayer case, the total number of 877 decomposition alternatives was reduced to 20. For each of these alternatives we can now follow, the approach of the previous two subsections to calculate the function dependency and data dependency values. Figure 6 shows the plot for the 20 decomposition alternatives that remain after applying the deployment and domain constraints. Hereby, the decomposition alternatives are listed along the *x-axis*. The *y-axis* on the left-hand side is used for showing the function dependency overheads of these alternatives. The *y-axis* on the right-hand side is used for showing the data dependency sizes of the decomposition alternatives. Using these results, the software architect can already have a first view on the feasible alternatives. The final selection of the alternatives will be explained in the next section.

8 Trade-off analysis and optimization

After the software architecture is described, design constraints are defined and the necessary measurements are performed on the system, the final set of decomposition alternatives can be selected as defined by the last group of activities (See Fig. 3). Using the domain constraints, we have seen that for the MPlayer case 20 alternatives were possible. This set of alternatives is further evaluated with respect to the performance feasibility constraints based on the defined thresholds and the measurements performed on the running system. For the MPlayer case, we have set the function dependency overhead threshold to 15% and the data dependency size threshold to 10.0 KB. It appears that after the application of performance feasibility constraints, only 6 alternatives are left in the feasible design space as listed in Fig. 7. Hereby, RUs are defined in the straight brackets '[' and ']'. For example, the alternative # 1 represents the alternative as defined in Fig. 2. Figure 8 shows the function dependency overhead and data dependency size for the 6

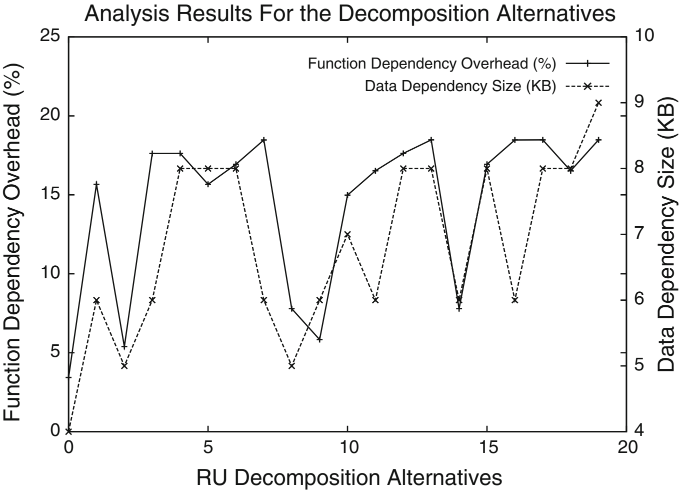


Fig. 6 Function dependency overhead and data dependency sizes for decomposition alternatives after domain constraints are applied

Fig. 7 The feasible decomposition alternatives with respect to the specified constraints

```
0: { [Mplayer, Libao, Libmpcodecs, Demuxer, Stream, Libvo] [Gui] }

1: { [Mplayer, Libmpcodecs, Demuxer, Stream, Libvo] [Gui] [Libao] }

2: { [Mplayer] [Gui] [Libao, Libmpcodecs, Demuxer, Stream, Libvo] }

3: { [Mplayer, Libao] [Gui] [Libmpcodecs, Demuxer, Stream, Libvo] }

4: { [Mplayer, Libao, Libmpcodecs, Demuxer, Stream] [Gui] [Libvo] }

5: { [Mplayer] [Gui] [Libao] [Libmpcodecs, Demuxer, Stream, Libvo] }
```

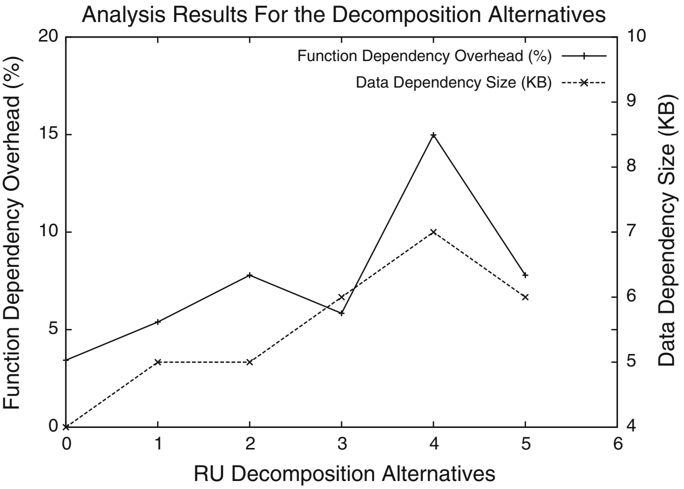


Fig. 8 Function dependency overhead and data dependency sizes for 6 decomposition alternatives

feasible decomposition alternatives. Hereby, we can see that the alternative # 4 has the highest value for the function dependency overhead. This is because this alternative corresponds to the decomposition, where the two highly coupled modules *Libvo* and *Libmpcodecs* are separated from each other. We can see that this alternative has also a distinctively high data dependency size. This is because this decomposition alternative separates the modules *Gui*, *Libvo* and *Libmpcodecs* from each other. As can be seen in Table 2, the size of data that is shared between these modules is the highest among all.

As described in Sect. 2.3, the selection of a particular decomposition alternative leads to a trade-off, where we consider two main attributes: *performance* and *availability*. In the following subsections, we focus on this trade-off. On one hand, we want to keep availability as high as possible. On the other hand, we want to keep function and data dependencies as low as possible to keep the performance overhead low.

8.1 Availability estimation

Since the main goal of local recovery is to maximize the system availability, we need to evaluate and compare the feasible decomposition alternatives based on availability as well. If the reduced design space is small enough, we can estimate the availability for each decomposition alternative by means of analytic models (Boudali et al. 2009) and select an alternative accordingly. If the design space is too large for this, we can use optimization techniques as explained in the following subsection.

We have adopted the following objective function to estimate the gain in availability that can be achieved by an RU decomposition.

$$MTTR_{RU_x} = \sum_{m \in RU_x} MTTR_m \tag{6}$$

$$1/MTTF_{RU_x} = \sum_{m \in RU_x} 1/MTTF_m \tag{7}$$

$$\text{Criticality}_{RU_x} = \sum_{m \in RU_x} \text{Criticality}_m \tag{8}$$

$$\text{objective function} = \text{min.} \sum_{RURU_x} \text{Criticality}_{RU_x} \times \frac{MTTR_{RU_x}}{MTTF_{RU_x}} \tag{9}$$

In Eqs. 6 and 7, we calculate for each RU the *MTTR* and *MTTF*, respectively. The calculation of *MTTR* for an RU is based on the assumption that all the modules comprised by an RU are recovered sequentially. That is why, the *MTTR* for an RU is simply the addition of *MTTR* values of the modules that are comprised by the corresponding RU. The calculation of *MTTF* for an RU is based on the assumption that the failure probability follows an exponential distribution with rate $\lambda = 1/MTTF$. If X_1, $X_2, \ldots$ and X_n are independent exponentially distributed random variables with rates $\lambda_1, \lambda_2, \ldots$ and λ_n, respectively, then $\min(X_1, X_2, \ldots, X_n)$ is also exponentially distributed with rate $\lambda_1 + \lambda_2 + \ldots + \lambda_n$ (Ross 2007). As a result, the failure rate of an RU ($1/MTTF_{RU\_x}$) is equal to the sum of failure rates of the modules that are comprised by the corresponding RU. In Eq. 8, we calculate the criticality of an RU by simply summing up the criticality values of the modules that are comprised by the RU. Equation 9 shows the objective function that is utilized by the optimization algorithms. As explained in Sect. 3.3, to maximize the availability, MTTR of the system must be kept as low as possible and MTTF of the system must be as high as possible. As a heuristic based on this fact, the objective function is to minimize the *MTTR*/*MTTF* ratio in total for all RUs, which are weighted based on the criticality values of RUs.

The objective function we define in Eq. 9 is actually a heuristic for the selection of a decomposition alternative. In this paper, we have defined a heuristic, where availability is considered as the main decomposition criterion. Note that this can be changed according to the designers' concerns and trade-off decisions. The defined heuristic is used as an input for the applied optimization technique(s) as described in the following subsection.

8.2 Optimization approaches

The selection of a decomposition alternative considering multiple quality attributes (in this case, availability and performance) requires us to solve a multicriteria optimization problem. We can benefit from several approaches (Grunske et al. 2007) to make the necessary trade-off decisions and optimize the selection of a decomposition alternative based on the analysis results as shown in Fig. 6. First, the designer must select an optimization strategy and technique based on the quality requirements and the size of the feasible design space.

One approach as an optimization strategy would be to reduce the problem to a single-objective function. This can be achieved by optimizing with respect to one quality factor while keeping the other quality factors as constraints. Another common multiobjective optimization approach forms a single-objective function from linearly weighted criteria

(Athon and Papalambros 1996). The third option is to use multiple objectives at the same time and use pareto-optimization if quality criteria are considered to be equally important.

The optimization technique to be utilized is mainly determined by the size of the feasible design space. If there are not too many modules and/or constraints are able to prune the design space extensively, the designer can directly use exhaustive search. Otherwise, if the design space is large, local search (e.g., hill-climbing algorithm) and approximation techniques must be utilized. In the latter case, it is possible to end up in a suboptimal but still an acceptable solution with respect to requirements. If the selected decomposition is not satisfactory, the granularity of the module decomposition or provided constraints can be revisited to decrease the size of the design space for tractability.

In the following, we provide examples for the application of two different approaches. (1) single-objective function (i.e., maximize availability such that function and data dependencies are below a certain threshold) optimized with exhaustive search or hill-climbing algorithm and (2) multiobjective pareto-optimization (i.e., maximize availability while minimizing function dependencies and data dependencies).

Single-objective optimization. The objective function introduced in Eq. 9 is adopted for single-objective optimization. We have considered two optimization techniques: *exhaustive search* and *hill-climbing algorithm*. In the case of exhaustive search, all the alternatives are evaluated and compared with each other based on the objective function. If the design space is too large for exhaustive search, hill-climbing algorithm can be utilized to search the design space faster but ending up with possibly a suboptimal result with respect to the objective function.

Hill-climbing algorithm starts with a random (potentially bad) solution to the problem. It sequentially makes small changes to the solution, each time improving it a little bit. At some point, the algorithm arrives at a point where it cannot see any improvement anymore, at which point the algorithm terminates. In our approach, solution alternatives (possible partitions) and iterations over these are defined based on the hill-climbing algorithm proposed by Mitchell and Mancoridis (2006). Hereby, our algorithm starts from the worst decomposition with respect to availability, where all modules of the system are placed in a single RU. Then, it systematically generates a set of *neighbor decompositions* by moving modules between RUs.[5] A decomposition *NP* is defined as a neighbor decomposition of *P* if *NP* is exactly the same as *P* except that a single module of an RU in *P* is in a different RU in *NP*. During the generation process, a new RU can be created by moving a module to a new RU. It is also possible to remove an RU when its only module is moved into another RU. The algorithm generates neighbor decompositions of a decomposition by systematically manipulating the corresponding RG string. For example, consider the set of RG strings of length 7, where the elements in the string correspond to the modules *Mplayer*, *Gui*, *Libao*, *Libmpcodecs*, *Demuxer*, *Stream* and *Libvo*, respectively. Then, the decomposition { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] } is represented by the RG string 1240350. By incrementing or decrementing one element in this string, we end up with the RG strings 1241350, 1242350, 1243350, 1244350, 1245350, 1246350, 1240340, 1240351, 1240352, 1240353, 1240354, 1240355 and 1240356, which correspond to the decompositions shown in Fig. 9, respectively.

For the MPlayer case, the optimal decomposition (ignoring the deployment and domain constraints) based on the heuristic-based objective function (Eq. 9) is { [Mplayer] [Gui] [Libao] [Libmpcodecs, Libvo] [Demuxer] [Stream] }. It took 89 seconds to find this decomposition with exhaustive search. The hill-climbing algorithm terminated on the same

[5] Each RU is a *partition* in the parlance of (Mitchell and Mancoridis 2006).

```
 1: { [Libvo] [Mplayer, Libmpcodecs] [Gui] [Demuxer] [Libao] [Stream] }
 2: { [Libvo] [Mplayer] [Gui, Libmpcodecs] [Demuxer] [Libao] [Stream] }
 3: { [Libvo] [Mplayer] [Gui] [Libmpcodecs, Demuxer] [Libao] [Stream] }
 4: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao, Libmpcodecs] [Stream] }
 5: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Libmpcodecs, Stream] }
 6: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] [Libmpcodecs] }
 7: { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao, Stream] }
 8: { [Libmpcodecs] [Mplayer, Libvo] [Gui] [Demuxer] [Libao] [Stream] }
 9: { [Libmpcodecs] [Mplayer] [Gui, Libvo] [Demuxer] [Libao] [Stream] }
10: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer, Libvo] [Libao] [Stream] }
11: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao, Libvo] [Stream] }
12: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao] [Stream, Libvo] }
13: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] [Libvo] }
```

Fig. 9 The neighbor decompositions of the decomposition { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] }

machine in 8 seconds with the same result. A total of 76 design alternatives had to be evaluated and compared by the hill-climbing algorithm, instead of 877 alternatives in the case of exhaustive search.

Multi-objective optimization. In some cases, multiple quality criteria can be treated to be equally important. As a result, there might not be a single optimal solution, but a set of Pareto-optimal (Pareto 1896) solutions instead. In this case, a design alternative can be selected in two steps: first by identifying and evaluating the Pareto-optimal alternatives and then selecting a particular alternative among these alternatives. There have been different techniques used for such a multiobjective optimization approach. For example, evolutionary strategies have been applied to solve multiobjective optimization problems in the context of architecture design (Aleti et al. 2009). In Fig. 10, we can see the feasible decomposition alternatives plotted with the corresponding values for 1/availability (the reciprocal of the objective function presented in Eq. 9), function dependency and data dependency, all of which must be minimized. The local recovery design alternative with 3 RUs i.e., MPCORE, AUDIO and GUI (Fig. 2) happens to be one of the Pareto-optimal decompositions. The corresponding 1/availability, function dependency and data dependency values are 3.53, 5.39 and 5.86, respectively. One of the other example Pareto-optimal decompositions shown in Fig. 10 has the minimal 1/availability (3.11, 18.48, 9.03). The other decomposition has the minimal function and data dependency values (8.07, 3.43, 4.86).

9 Tool support

Our approach requires the utilization of various analysis techniques including control/data flow analysis and optimization. It is not feasible to manually apply and integrate these techniques for large-scale systems with many decomposition alternatives. Therefore, we believe that it is important to provide tool support for such a process; however, it is not possible (and usually not desirable) to fully automate all the activities. The designer should be able to steer the process by providing domain knowledge, certain design choices and

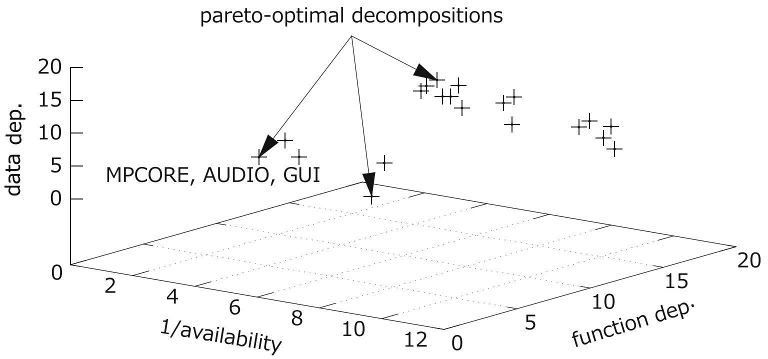


Fig. 10 Pareto-optimal decompositions with respect to availability and interdependencies (values are normalized)

Table 3 The level of automation for the main activities of the process

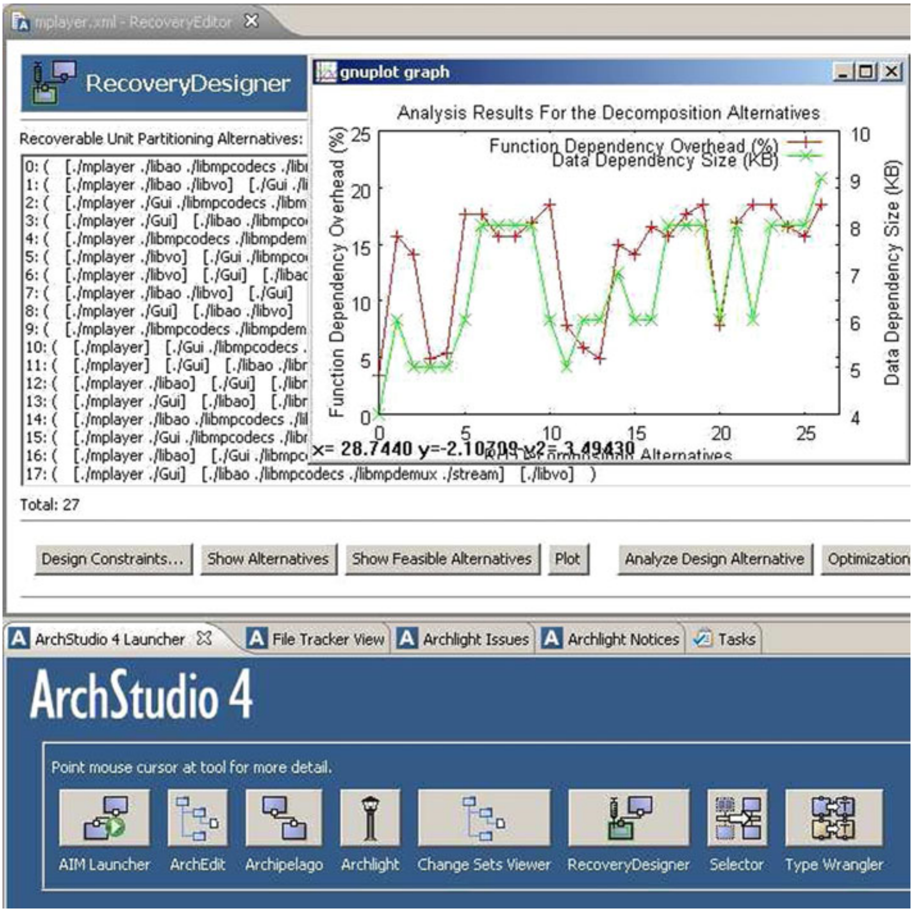
Activities	Level of automation
Architecture definition	Manual
Constraint definition	Semi-automatic
Measurement	Fully automatic
Decomposition selection	Semi-automatic

trade-off decisions. Therefore, we have developed an analysis tool to be used by the designer interactively. This tool, called *Recovery Designer*,[6] consists of several subtools each of which automate different parts of the process. Table 3 summarizes the level of automation provided for each group of activities depicted in Fig. 3.

The specification of the software architecture and its annotation regarding reliability properties are manual activities (though graphical editors are used for this purpose). The annotated architecture description is provided as an input to *Recovery Designer*. Similarly, the specification of constraints is also a manual activity; however, *Recovery Designer* checks the specified constraints for inconsistencies, and it provides immediate feedback about the achieved design space reduction. All the measurement activities are automatically performed, and charts are generated to present the measurement results. *Recovery Designer* implements a set of optimization algorithms. Hence, the selection of decomposition is also automated; however, the designer can influence the outcome by changing the constraints and the objective function that are provided as inputs to the optimization algorithms. As a default objective function, we have defined a heuristic for decomposition selection, where availability is considered as the main decomposition criterion (Eq. 9). In brief, the designer defines/updates the annotated architecture description, a set of constraints and (optionally) the objective function, which are provided as an input to the analysis tool.

Recovery Designer is fully integrated in *Arch-Studio* (Dashofy et al. 2002), which is an open-source software and systems architecture development environment based on the Eclipse open development platform. It originally includes a set of tools for specifying and analyzing software architectures. We have used the meta-modeling facilities of ArchStudio

[6] The tool is available online at http://srl.ozyegin.edu.tr/tools/ard/.



```
0: (   [./mplayer ./libao ./libmpcodecs ./libı
1: (   [./mplayer ./libao ./libvo]   [./Gui ./li
2: (   [./mplayer ./Gui ./libmpcodecs ./libm
3: (   [./mplayer ./Gui]   [./libao ./libmpco
4: (   [./mplayer ./libmpcodecs ./libmpdem
5: (   [./mplayer ./libvo]   [./Gui ./libmpco
6: (   [./mplayer ./libvo]   [./Gui]   [./libac
7: (   [./mplayer ./libao ./libvo]   [./Gui]
8: (   [./mplayer ./Gui]   [./libao ./libvo]
9: (   [./mplayer ./libmpcodecs ./libmpdem
10: (   [./mplayer]   [./Gui ./libmpcodecs .
11: (   [./mplayer]   [./Gui]   [./libao ./libr
12: (   [./mplayer ./libao]   [./Gui]   [./libr
13: (   [./mplayer ./Gui]   [./libao]   [./libr
14: (   [./mplayer ./libao ./libmpcodecs ./lil
15: (   [./mplayer ./Gui ./libmpcodecs ./libr
16: (   [./mplayer ./libao]   [./Gui ./libmpc
17: (   [./mplayer ./Gui]   [./libao ./libmpcodecs ./libmpdemux ./stream]   [./libvo]   )
Total: 27
```

mplayer.xml - RecoveryEditor
RecoveryDesigner
Recoverable Unit Partitioning Alternatives:
gnuplot graph
Analysis Results For the Decomposition Alternatives
Function Dependency Overhead (%)
Data Dependency Size (KB)
0
5
10
15
20
25
4
5
6
7
8
9
10
RUs Partitioning Alternatives
x= 28.7440 y=-2.10709 y2=-3.49430
Design Constraints...
Show Alternatives
Show Feasible Alternatives
Plot
Analyze Design Alternative
Optimization
ArchStudio 4 Launcher
File Tracker View
Archlight Issues
Archlight Notices
Tasks
ArchStudio 4
Point mouse cursor at tool for more detail.
AIM Launcher
ArchEdit
Archipelago
Archlight
Change Sets Viewer
RecoveryDesigner
Selector
Type Wrangler

Fig. 11 A snapshot of the arch-studio recovery designer

to extend the tool and integrate *Recovery Designer* with the provided default set of tools. A snapshot of the extended *Arch-Studio* can be seen in Fig. 11 in which *Recovery Designer* is activated. The architecture of *Recovery Designer* is shown in Fig. 12. The tool boundary is defined by the large rectangle with dotted lines. The tool itself consists of 5 subtools; *Constraint Evaluator*, *Design Alternative Generator*, *Function Dependency Analyzer*, *Data Dependency Analyzer* and *Optimizer*. Further, it uses 4 external tools *Arch-Edit*, *GNU-gprof*, *Valgrind* and *gnuplot*.

In the following subsections, we explain different components of the analysis tool in detail.

9.1 Constraint evaluator

Constraint Evaluator gets as input the architecture description that is created with the *Arch-Edit* tool. We can see a snapshot of the *Constraint Evaluator* in Fig. 13. The user interface consists of three parts: *Deployment Constraints*, *Domain Constraints* and *Performance Feasibility Constraints*, each corresponding to a type of constraint to be

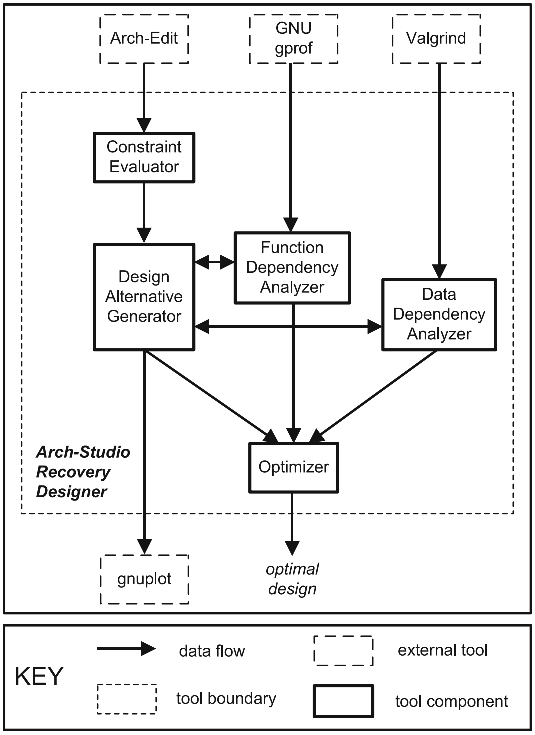
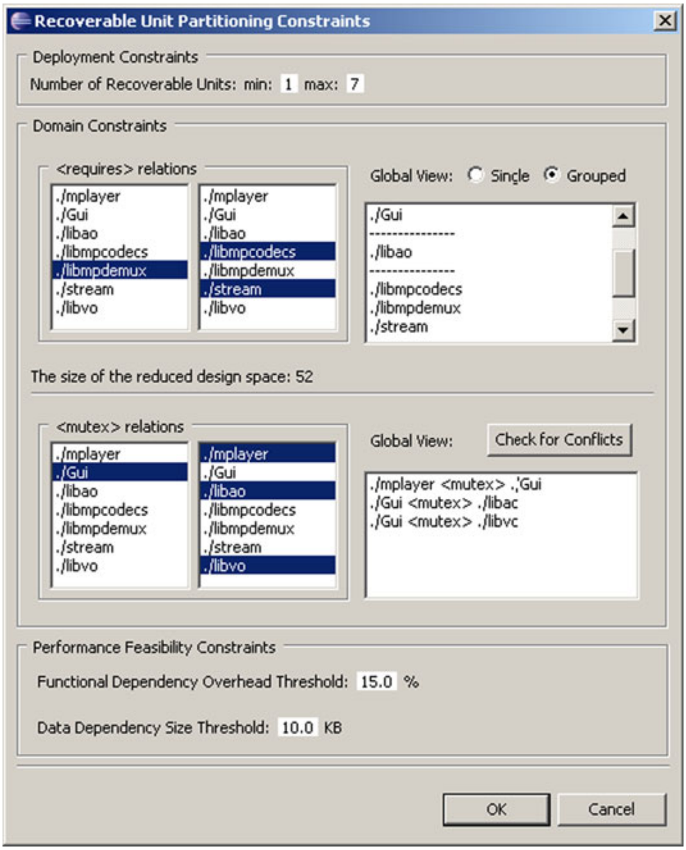


Fig. 12 Arch-studio recovery designer analysis tool architecture

specified. In the *Deployment Constraints* part, we specify the limits for the number of RUs. In Fig. 13, the *minimum* and *maximum* number of RUs are specified as 1 and 7 (i.e., the total number of modules), respectively, which means that there is no limitation to the number of RUs. In the *Domain Constraints* part, we specify *requires*/*mutex* relations. For each of these relations, *Constraint Evaluator* provides two lists that include the name of the modules of the system. When a module is selected from the first list, the second list is updated, where the modules that are related are selected (initially, there are no selected elements). The second list can be modified by multiple (de)selection to change the relationships. For example, in Fig. 13, it has been specified that *Demuxer* must be in the same RU as *Stream* and *Libmpcodecs*, whereas *Gui* must be in a separate RU than *Mplayer*, *Libao* and *Libvo*. *Constraint Evaluator* can also automatically check whether there are any conflicts between the specified *requires* and *mutex* relations (i.e., two modules must be kept together and separated at the same time).

In the *Performance Feasibility Constraints* part, we specify thresholds for the amount of function and data dependencies between the separated modules. The specified constraints are used to eliminate alternatives that exceed the given threshold values. Evaluation of the constraints requires measurements to be performed from the running system. However, if there is no existing system available and we cannot perform the necessary measurements, we can skip the analysis of the *performance feasibility constraints*. *Arch-Studio Recovery Designer* provides an option to enable/disable the performance feasibility analysis. In case this analysis is disabled, the design space will be evaluated based on only the *deployment*



Number of Recoverable Units: min: 1 max: 7

./mplayer ./Gui ./libao ./libmpcodecs ./libmpdemux ./stream ./libvo

./mplayer ./Gui ./libao ./libmpcodecs ./libmpdemux ./stream ./libvo

Global View: ○ Single ● Grouped

./Gui

./libao

./libmpcodecs
./libmpdemux
./stream

The size of the reduced design space: 52

./mplayer ./Gui ./libao ./libmpcodecs ./libmpdemux ./stream ./libvo

./mplayer ./Gui ./libao ./libmpcodecs ./libmpdemux ./stream ./libvo

Global View:

./mplayer <mutex> ./Gui
./Gui <mutex> ./libao
./Gui <mutex> ./libvo

Functional Dependency Overhead Threshold: 15.0 %

Data Dependency Size Threshold: 10.0 KB

Fig. 13 Specification of design constraints

constraints and *domain constraints* so that it is still possible to generate the decomposition alternatives and depict the reduced design space.

9.2 Design alternative generator

Design Alternative Generator computes the size of the feasible design space based on the architecture description and the specified domain constraints (as explained in Sect. 6) Note that this information is provided to the user already during the *Constraint Definition* process (See the GUI of the *Constraint Evaluator* tool in Fig. 13). Then, it generates the set of decomposition alternatives using the *restricted growth (RG) strings* (Ruskey 2003) and eliminates alternatives that violate any constraints. The generated design alternatives are provided to the *Function Dependency Analyzer*, *Data Dependency Analyzer* and *Optimizer* tools.

Design Alternative Generator also generates charts corresponding to the generated design spaces, and it uses *gnuplot* to depict them (e.g., Fig. 6). The generated charts show the function dependency overhead and the data dependency size for each decomposition alternative as calculated by the *Function Dependency Analyzer* and *Data Dependency Analyzer* tools, respectively.

9.3 Function dependency analyzer

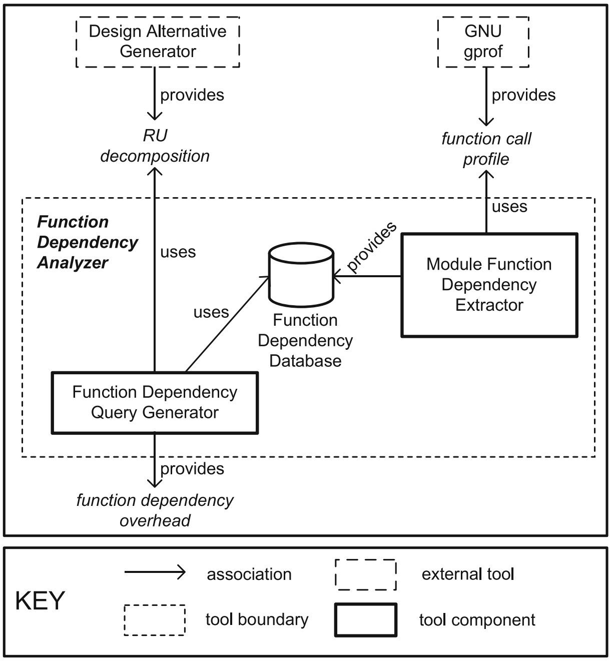
Function Dependency Analyzer tool performs function dependency analysis based on the inputs from the *Design Alternative Generator* and *GNU gprof* tools (See Fig. 12). As shown in Fig. 14, the *Function Dependency Analyzer* tool itself is composed of three main components: (1) *Module Function Dependency Extractor* (2) *Function Dependency Database* and (3) *Function Dependency Query Generator*.

Module Function Dependency Extractor uses the *GNU gprof* tool to obtain the function call graph of the system. *GNU gprof* also collects statistics about the frequency of performed function calls and the execution time of functions. Once the *function call profile* is available, *Module Function Dependency Extractor* uses an additional GNU tool called *GNU nm* (provides the symbol table of a C object file) to relate the function names to the C object files. As a result, *Module Function Dependency Extractor* creates the corresponding MDG.

To be able to query the function dependencies between the system modules, we need to relate the set of nodes in the MDG to the system modules. *Module Function Dependency Extractor* uses the package structure of the source code to identify the module that a C object file belongs to. This is also reflected to the prefixes of the nodes of the MDG. For example, each file that belongs to the Gui module has "*Gui*" as the prefix. *Module Function Dependency Extractor* exploits the full path of the C object file, which reveals its prefix (e.g., "./Gui*") and the corresponding module. For the MPlayer case, the set of packages that corresponds to the provided module view (Fig. 1) was processed.

After the MDG is created, it is stored in the *Function Dependency Database*, which is a relational database. Once the MDG is created and stored, *Function Dependency Analyzer* becomes ready to calculate the function dependency overhead for a particular selection of RUs defined by the *Design Alternative Generator*. For each alternative, *Function*

Fig. 14 Function dependency analysis


Design Alternative Generator
GNU gprof
provides
provides
RU decomposition
function call profile
uses
Function Dependency Analyzer
uses
provides
Module Function Dependency Extractor
uses
Function Dependency Database
Function Dependency Query Generator
provides
function dependency overhead
KEY
association
external tool
tool boundary
tool component

Dependency Query Generator accesses the *Function Dependency Database* and automatically creates and executes queries to estimate the function dependency overhead (Eq. 4). An example query is shown in Fig. 15, where the number of calls from the module *Gui* to the module *Libao* is queried.

9.4 Data dependency analyzer

Data Dependency Analyzer is responsible for providing information regarding the data dependencies among the software modules. It is composed of three main components: (1) *Module Data Dependency Extractor* (2) *Data Dependency Database* and (3) *Data Dependency Query Generator* (See Fig. 16).

Module Data Dependency Extractor uses the *Valgrind* tool to obtain the data access profile of the system. *Valgrind* (Nethercote and Seward 2007) is a dynamic binary instrumentation framework, which enables the development of dynamic binary analysis tools. Such tools perform analysis at run time at the level of machine code. *Valgrind* provides a core system that can instrument and run the code, plus an environment for writing tools that plug into the core system (Nethercote and Seward 2007). A *Valgrind tool*

```
SELECT Sum(module_dependency.calls)
AS total FROM module_dependency
WHERE (module_dependency.src Like './Gui%'
And module_dependency.dest Like './libao%')
```

Fig. 15 The generated SQL query for calculating the number of calls from the module *Gui* to the module *Libao*

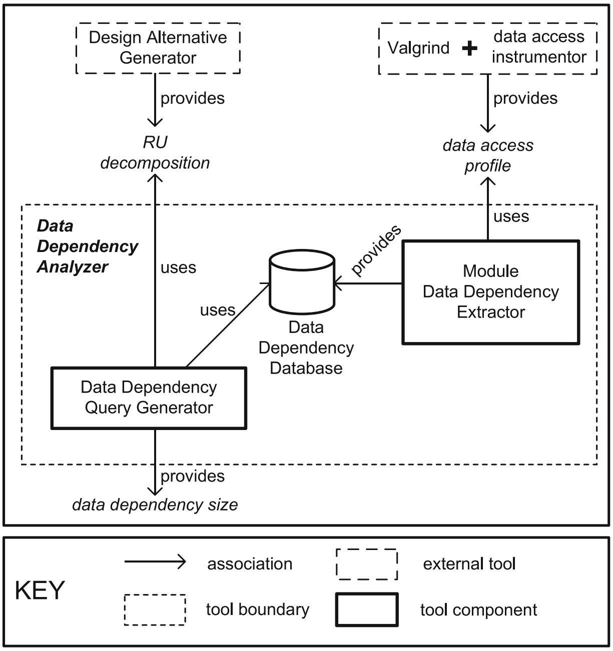


Fig. 16 Data dependency analysis

is basically composed of this core system plus the plug-in tool that is incorporated to the core system. We use *Valgrind* to obtain the data access profile of the system. To do this, we have written a plug-in tool for *Valgrind*, namely the *Data Access Instrumentor* (DAI), which records the addresses and sizes of the memory locations that are accessed by the system. A sample of this data access profile output can be seen in Fig. 17.

In Fig. 17, we can see the addresses and sizes of memory locations that have been accessed. In line 7 for instance, we can see that there was a memory access at address *bea*1*d*4*a*8 of size 4 from the file "audio_out.c". DAI outputs the full paths of the files, where a memory access was performed. The full path information is used for identifying the corresponding module. The related parts of the file paths are underlined in Fig. 17. For example, in line 9, we can see that the file "aclib_template.c" belongs to the *Libvo* module. From this data access profile, we can also observe that the same memory locations can be accessed by different modules. Based on this, we can identify the data dependencies. For instance, Fig. 17 shows a memory address *bea*97498 that is accessed by both the *Gui* and the *Mplayer* modules as highlighted in lines 2 and 6, respectively. The output of DAI is used by the *Module Data Dependency Extractor* to search for all such memory addresses that are shared by multiple modules. The output of the *Module Data Dependency Extractor* is the total number and size of data dependencies between the modules of the system. This information is stored in the *Data Dependency Database*.

Once the data dependencies are stored, *Data Dependency Analyzer* becomes ready to calculate the data dependency size for a particular selection of RUs defined by the *Design Alternative Generator*. *Data Dependency Query Generator* accesses the *Data Dependency Database*, creates and executes queries for each RU decomposition alternative to estimate the data dependency size. The querying process is very similar to the querying of function dependencies as explained in Sect. 9.3. The only difference is that the information being queried is data size instead of the number of function calls.

9.5 Optimizer

The *Optimizer* tool of *Arch-Studio Recovery Designer* implements two single-objective optimization techniques: *exhaustive search* and *hill-climbing algorithm*. In the case of exhaustive search, all the decomposition alternatives are compared with each other based on the objective function (Eq. 9). The hill-climbing algorithm is implemented as described in Sect. 8.2. *Optimizer* receives the set of generated design alternatives from *Design Alternative Generator* and outputs a decomposition alternative that is selected as the result of optimization.

```
 1: ... /MPlayer-1.0rc1/Gui/interface.c: bea1d298,4;
 2: bea97498, 4; ... /MPlayer-1.0rc1/Gui/cfg.c:
 3: bea1d2a0, 4; 0862b714, 4; bea1d29c, 4; bea1d2a8,
 4: 4; bea1d2a4, 4; bea1d294, 4; bea1d288, 4; ...
 5: /MPlayer-1.0rc1/mplayer.c: bea1d4e8, 4; bea1d4e4,
 6: 4; bea1d4e0, 4; bea1d4dc, 4; bea97498, 4; ...
 7: /MPlayer-1.0rc1/libao2/audio_out.c: bea1d4a8, 4;
 8: bea1d4a4, 4; bea1d4a0, 4; bea1d49c, 4; bea1d498,
 9: 4; ... /MPlayer-1.0rc1/libvo/aclib_template.c:
10: 086b4860, 16; 086b4870, 16; ...
```

Fig. 17 A sample output of Valgrind + Data Access Instrumentor

10 Evaluation

The utilization of analysis and optimization techniques helps us to analyze, compare and select a decomposition among many alternatives. The main goal of decomposing the software architecture is to introduce local recovery and as such increase the system availability. To evaluate the increase in availability and the viability of the heuristics used for optimization, we have performed real-time measurements from systems that are decomposed for local recovery. We have also collected actual performance data to evaluate our metrics regarding the performance overhead. In this section, we briefly explain the implementation issues, how the measurements are performed and the results of our assessments.

10.1 Implementation of local recovery

Introducing local recovery to a software system, while preserving the existing decomposition, is not trivial and requires a substantial development and maintenance effort. To reduce this effort, we have implemented a framework, FLORA (Sozer et al. 2009), for supporting the decomposition and implementation of software architecture for local recovery. FLORA comprises interprocess communication (IPC) utilities, serialization/de-serialization primitives, error detection and diagnosis mechanisms, an RU wrapper template, a *recovery manager* that coordinates recovery actions and a *connector* that mediates and controls the communication among RUs. Each RU (a set of modules as defined by an RU wrapper template) is assigned to a separate process. All the function calls among the modules that are part of different RUs are captured by FLORA. These calls are marshaled and forwarded to the corresponding RUs through IPC using domain sockets. The framework has been implemented in the C language on a Linux platform.

By means of FLORA, we have introduced local recovery to MPlayer for 3 different decomposition alternatives: (1) global recovery, where all the modules are placed 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] }) as shown in Fig. 18. Note that the 2nd and the 3rd implementations correspond to the decomposition alternatives # 0 and # 1 in Fig. 7, respectively. We have selected these decomposition alternatives because they have the lowest function dependency overhead and data dependency size.

The 3rd decomposition alternative was also shown in Fig. 2. Figure 18 shows the recovery view (Sozer and Tekinerdogan 2008) of the architecture of MPlayer after it is decomposed with FLORA according to this decomposition alternative. Hereby, we can see three recoverable units, *RU MPCORE*, *RU GUI* and *RU AUDIO*. *RU AUDIO* provides the functionality of *Libao*. *RU GUI* encapsulates the *Gui* functionality. *RU MPCORE* comprises the rest of the system. The components *Connector* and *Recovery Manager* are introduced by FLORA. Each RU can detect deadlock errors.[7] *Recovery Manager* can detect fatal errors.[8] All error notifications are sent to *Connector*, which comprises the

[7] An RU detects if an expected response to a message is not received within a period of time.

[8] The *Recovery Manager* is the parent process of all RUs and receives and handles a signal when a child process is dead.

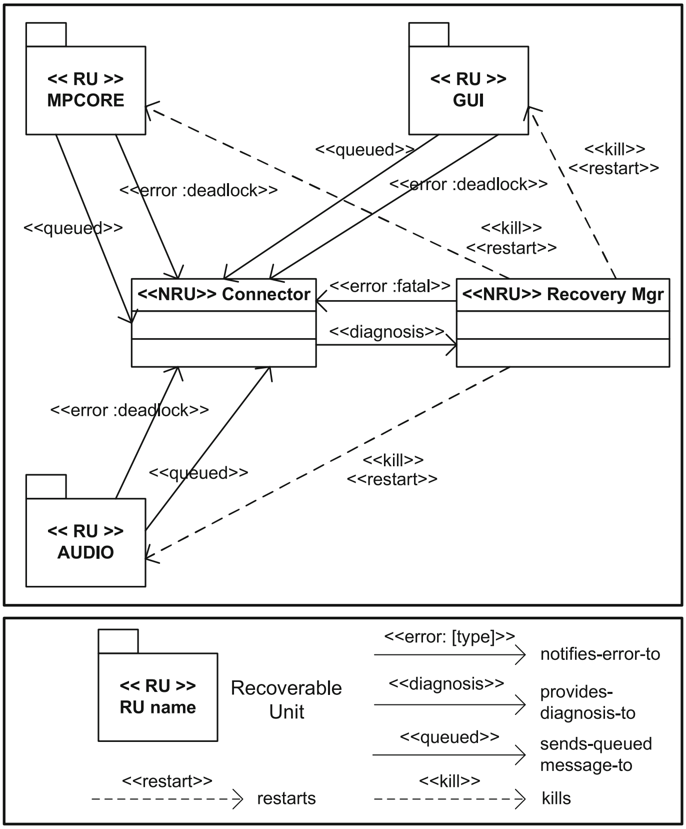


Fig. 18 The implementation of local recovery for the decomposition { [Mplayer, Libmpcodecs, Libvo, Demuxer, Stream] [Gui] [Libao] }

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.

10.2 Evaluation of availability

To be able to measure the availability achieved with the implementations of 3 decomposition alternatives, we have modified each module so that they fail with a specified failure rate (assuming an exponential 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 2.

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 to 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).

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 times, when the core system module *Mplayer* has been down. We have calculated the availability of the system based on the total time that the system has been running (5 h). For the error injection, first we have used the same MTTF values (1,800 s) for all the modules. Then, to amplify the effect of decomposition on availability, we have assigned 60 and 30 s to the MTTF values of *Libao* and *Gui* modules, respectively. Table 4 lists the results for the three decomposition alternatives and for the two sets of MTTF values used.

In Table 4, the 1st column lists the decomposition alternatives. The first part of the table (2nd and 3rd columns) presents the results for test runs, where *MTTF* values of all the modules are specified as 1,800 s. The second part (4th and 5th columns) presents the results for test runs, where the *MTTF* values for the modules *Libao* and *Gui* are specified as 60 and 30 s, respectively. The 2nd and 4th columns show the measured availability of the RU that comprises the *Mplayer* module. The 3rd and 5th columns show the *MTTR*/*MTTF* ratio of this RU, which is used as a part of the cost function in Eq. 9. The cost function, which based on *MTTR*/*MTTF* ratios, reflects the lack of availability of a design alternative, and as such, it is tried to be minimized. Based on the measured availabilities, we have seen that the alternatives were ordered correctly with respect to the heuristic cost function used for optimization. As such, our approach can be used for accurately analyzing, comparing and selecting decomposition alternatives for local recovery.

Algorithm 2 Periodically activated thread for error injection

```
1: $time\_init \leftarrow currentTime()$
2: while $TRUE$ do
3:     $time\_elapsed \leftarrow currentTime() - time\_init$
4:     $p \leftarrow 1 - 1/e^{time\_elapsed/MTTF}$
5:     $r \leftarrow random()$
6:     if $p \geq r$ then
7:         $injectError()$
8:         $break$
9:     end if
10: end while
```

Table 4 Comparison of the measured availability with the heuristic function values

Decomposition alternative	all $MTTF = 1800$ s		$MTTF_{Libao} = 60$ s, $MTTF_{Gui} = 30$ s, all other $MTTF = 1800$ s	
	RU_{MPCORE} Availability	*MTTR*/*MTTF*	RU_{MPCORE} Availability	*MTTR*/*MTTF*
all in 1 RU	97.57	14.47	83.59	196.31
Gui, *the rest*	97.58	10.80	93.25	62.99
Gui, *Libao*, *the rest*	97.75	7.33	97.75	7.33

10.3 Evaluation of performance overhead

One of the main characteristics of recovery-oriented architectures is the isolation of (faulty) components (Patterson et al. 2002). Isolation is necessary, and the overhead introduced by our approach, like in many recoverable systems, is mainly due to isolation of modules in separate processes. For instance, FLORA marshals and transfers all the function calls among the isolated modules through IPC. For this reason, we queried function dependencies among the isolated modules and used this to define a metric (Eq. 4) for estimating the performance overhead.

To evaluate the accuracy of our estimations, we have collected measurements from the implementations of 3 decomposition alternatives. For each of these systems, we have calculated the average time elapsed for frame processing during a video-playing scenario. We have compared the obtained measurements with our estimations. Table 5 presents the results.

As it can be seen in the table, the estimations based on function dependencies closely reflect the measurements of actual performance overhead obtained from the implementations. As such, our metric that is defined in terms of function dependencies can be used for comparing and selecting decomposition alternatives.

11 Discussion

We have introduced a systematic approach to analyze the decomposition of the software architecture of an existing system to introduce local recovery. The overall process starts with the architecture definition. Hereby, we have used the module view of the architecture for analysis. We could also use the component and connector views or the allocation views (Clements et al. 2002a) to take run-time or deployment elements into account. In our approach, the basic abstraction mechanism is the *recoverable unit* (RU), which can include a set of architectural modules. When we would apply the approach to, for example, the component and connector view, the recoverable unit would not include modules but components and/or connectors instead. Similarly, recoverable units might include nodes when we adapt the allocation view of the architecture. In principle, the concept of RU is agnostic to the type of the architectural element and as such can be applied to multiple views.

For defining the decomposition alternatives, an important step is the specification of constraints. The more and the better we can specify the corresponding constraints, the more we can reduce the design space of decomposition alternatives. In this work, we have specified deployment constraints (number of possible RUs), domain constraints and feasibility constraints (performance overhead thresholds). The domain constraints are specified with binary *requires* and *mutex* relations. This has shown to be practical for designers who are not experts in defining complicated formal constraints. Nevertheless, we are going

Table 5 Comparison of measured performance overhead with function dependencies

Decomposition alternative	Function dependency (%)	Performance overhead (%)
all in 1 RU	0.0	0.03
Gui, *the rest*	3.44	3.59
Gui, *Libao*, *the rest*	5.39	5.90

to perform further research to improve the expressiveness power of the constraint specification. For this, we are considering to explore possibilities based on first-order logic. We are aware of the fact that in certain cases, the evaluation of the logical expressions and checking for conflicts can become NP-complete (e.g., the satisfiability problem). However, we will explore the workarounds and simplifications that might keep the approach feasible and practical.

To evaluate the performance feasibility constraints, one of the main activities in the overall process involves the analysis of function and data dependencies in the existing system. For this, we have applied *dynamic analysis*, in which we run the existing system and collect its function call and data access profile. We could successfully derive the amount of function dependency and the size of shared data among the modules and use this in the evaluation of the decomposition alternatives. In this case, dynamic analysis appeared to be essential, and we could not derive the required information using *static analysis* techniques. The reason for this is twofold. First of all, static analysis techniques do not scale up for data analysis, especially when it comes to performing expensive operations like pointer analysis. We have experienced this when we first tried to use existing static analysis tools (Necula et al. 2002; Teitelbaum 2000) for the MPlayer case. Secondly, the frequency and execution times of function calls can inherently not be measured using static analysis techniques.

During the dynamic analysis, the frequency of calls, execution times of functions and the data access profile can vary depending on the usage scenario and the inputs that are provided to the system. In our case study, we have performed the 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 (Fenlason and Stallman 2000). The analysis process will remain the same although the input profile data can be different. However, experiments (de Visser 2008) and case studies with real users need to be performed to obtain a representative set of usage scenarios for a system.

Due to software evolution, the defined decomposition for recoverability might need to be redefined. This might also require the dynamic configurability of the system. However, our focus in this work is on systems, which do not evolve rapidly concerning the software architecture (e.g., TV systems). Furthermore, in case we consider the decomposition for recovery, we can state that this is mainly influenced by reliability, availability and related properties of software modules like MTTF and MTTR. These properties are usually stable, and they are unlikely to change quickly with evolution. In TV systems, for instance, the streaming platform has always been relatively very reliable, whereas the faults in the application layer (e.g., teletext, epg) have caused the majority of failures (Tekinerdogan et al. 2008).

In our approach, we have considered two quality attributes: availability and performance. However, the approach itself is generic and extensible for consideration of other perspectives and quality attributes. For example, constraints can be defined to eliminate decomposition alternatives that are deemed to be precarious from the maintenance or evolution perspective. For integrating such a different/additional quality attribute (e.g., maintainability), *i)* means should be provided to make quantitative estimations regarding this quality and *ii)* these estimations should be used in the optimization approach, either as a constraint or as an additional dimension in a pareto-optimization.

12 Related work

Candea et al. introduced the *microreboot* (Candea et al. 2004b) 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* (Candea et al. 2004b), where components are isolated from each other and their state information is kept in stable repositories. Unfortunately, designs of many existing systems do not have these properties. Such systems have to be decomposed to support isolation, and until now, the decisions on how to decompose an existing system have been based on qualitative analysis (Candea et al. 2004a).

In (Herder et al. 2007), 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 operating system. The design of the operating system must support isolation between the core operating system and its extensions to enable such a recovery with limited re-engineering effort (Herder et al. 2007). In this work, we focus on software systems, in which isolation is not supported by the existing design.

Microrebooting has also been investigated in the context of service-oriented architectures. In (White et al. 2009), application containers are exploited for independent rebooting and each time the corresponding subsystem is restarted with a new service composition for recovery. The alternative set of composable services are specified with a feature diagram. Hereby, subsystems are already defined as possible compositions of loosely coupled services. Therefore, decomposing/partitioning the system for microrebooting is not necessary.

In cluster computing paradigm, a problem is split into smaller tasks, which are processed by a collection of interconnected, stand-alone computing resources. This paradigm has been mainly used for achieving better performance and throughput. Recently, the distribution of computation has also been studied for recoverability and high-availability (Nguyen et al. 2001; Santos et al. 2008). Fault isolation, dynamic redundancy and run-time configuration are the basic principles employed by these studies. There have been efforts to provide middleware support for fault tolerance by employing these principles. Fault Tolerant CORBA (Object Management Group 2001) is a result of these efforts, enabling the distribution of application software processing. More recently, the Service Availability Forum established standards (Jokiaho et al. 2003) for providing high availability middleware. Standard interfaces are defined to utilize reusable tools for automatic failure detection and automatic reconfiguration. To be able to utilize such tools and middleware support in general, the designer should first decide on the architecture decomposition of the existing system for recoverability and determine the units of recovery (e.g., clusters) before deploying the system on such enabling platforms. In this work, our basic assumption is that we have an existing system that was designed without recovery in mind. Hence, there is no built-in configuration mechanism/possibility, and as such, the goal is to refactor this system for error containment. There exist such legacy systems with millions lines of code (implemented as a one single unit), and it is not feasible to develop these systems from scratch. If a system is being designed with recovery in mind from the beginning, the approach could be different.

To model and analyze architectural decomposition alternatives, several architecture approaches have been proposed in the literature. For modeling software architectures, there is now an increasing agreement that software architecture should be represented using multiple, different views. Hereby, an architectural view is defined as a representation of a set of system elements and relations associated with them to support a particular concern (Clements et al. 2002a). Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture from the perspective of different concerns of various stakeholders. In addition to architectural views, architectural tactics (Bachman et al. 2003) and architectural patterns (Buschmann et al. 1996) are utilized to ensure that the architecture meets the required quality concerns. In the last decade, the software engineering community has witnessed the proposal of an increasing number of architectural patterns that are also applied in practice. Certainly, architectural modeling approaches support the design of architecture for quality. We have applied and enhanced existing view-based approaches to model each individual decomposition alternative (Sozer and Tekinerdogan 2008). Unfortunately, the modeling approaches by themselves lack explicit support for selecting feasible alternatives in the large design space.

For analyzing software architectures, broad number of architecture analysis approaches has been introduced in the last two decades (Clements et al. 2002b; Dobrica and Niemela 2002; Gokhale 2007). These architecture analysis approaches usually either perform static analysis of formal architectural models (Medvidovic and Taylor 2000) or utilize scenario-based approaches as described in (Dobrica and Niemela 2002). The goal of these approaches is to assess whether or not a given software architecture design satisfies the desired quality requirements. In alignment with this, more recently, several architecture analysis approaches have focused on identifying the trade-offs of different quality concerns for a given architecture. For example, architectural analysis methods like ATAM (Clements et al. 2002b) consider the impact of design alternatives on different quality attributes and discover conflicting attributes leading to trade-offs. The architecture analysis approaches in the literature help to identify the risks, sensitivity points and trade-offs in the architecture. These are general-purpose approaches, which are not dedicated to a particular decomposition issue or a specific set of qualities. In this paper, we have proposed a dedicated analysis approach that is required for optimizing the decomposition of architecture for local recovery in particular. Moreover, trade-off analysis techniques such as ATAM (Clements et al. 2002b) are based on informal reasoning, and they are carried out manually. It is difficult to carry out trade-off analysis in such a way, especially for complex systems, because subtle differences that have large impact cannot be easily considered, and due to large amount of alternatives, one cannot easily identify the architectural decomposition that offers the best trade-off. For this reason, we believe that it is important to provide integrated tool support for quality estimation (e.g., via dynamic control/data flow analysis) and optimization.

Clustering/partitioning techniques have been applied in many different disciplines like civil engineering (Alexander 1964). The basic design principle of clustering is isolation of components from each other and as such achieving high cohesion and low coupling (Heyliger 1994). In software engineering, clustering has been mainly utilized for reverse engineering (Mitchell and Mancoridis 2006; Davey and Burd 2000; Anquetil et al. 1999; Wiggerts 1997) by creating views of the structure of complex, large-scale software systems. Here, the assumption is that there exist high coupling among highly related modules. Clustering techniques have also been used for forward engineering. For example, in (Lung et al. 2007), requirements are clustered to create a conceptual architecture in early design phases. The interdependencies between requirements are identified based on their attributes as described in the requirements document. The use of clustering in this work is mainly aligned with the software clustering techniques (Wiggerts 1997) utilized for reverse engineering. In this domain, the term *entity* describe elements being grouped together. *Features* denote the attributes of these entities. A *clustering algorithm* is applied to group entities together based on a *similarity measurement*. A (possible) clustering generated as the outcome is called a *partition*. Files, functions or procedures of a program can be considered as entities for software clustering. References/calls to variables/functions of other entities or its naming convention are usually treated as the features of an entity (Davey and Burd 2000; Anquetil et al. 1999). Similarity measurement is mainly defined based on the degree of coupling between the compared entities (Anquetil et al. 1999; Mitchell and Mancoridis 2006). Several optimization techniques are utilized for the clustering algorithm, ranging from hill-climbing algorithms to genetic algorithms (Mitchell and Mancoridis 2006). The resulting partitions infer the modules, packages, subsystems or components of the software architecture that is tried to be retrieved by reverse engineering. In the parlance of this terminology, entities are *software modules* in our approach. Software modules as documented in the architecture description, and they correspond to files and folders in the file system. Features of an entity include (1) the deployment constraints (requires/mutex relations with respect to other entities), (2) the (function and data) *dependencies* to other entities and (3) the *reliability properties* (MTTF, MTTR, criticality). Instead of a similarity measurement, we define a *heuristic* as an estimation of availability for a particular clustering alternative (Eq. 9). This heuristic is used as an objective function for our clustering algorithm, while the deployment constraints and the dependencies among entities are used for eliminating infeasible clustering alternatives. We consider various optimization techniques to be applicable as the clustering algorithm 8.2; however, the main algorithm we use in this work is an application of the *hill-climbing approach*. This algorithm is very similar to and inspired from (Mitchell and Mancoridis 2006). Although the clustering algorithm is similar, note that the goals and the end results are different in our approach. In general, the only quality attribute considered by software clustering approaches is modularity (Mitchell and Mancoridis 2006). In our approach, the main focus is on availability, and each partition defines a *recoverable unit*.

13 Conclusion

Local recovery is an effective approach for recovering from errors, in which the erroneous parts of a system are recovered while the other parts of the system are operational. One of the requirements for introducing local recovery to a system is isolation. To prevent the propagation of errors, a software architecture must be decomposed into a set of isolated recoverable units. There exist many alternatives to decompose a software architecture for local recovery. Each alternative has an impact on availability and performance of the system.

In this paper, we have proposed a systematic approach for analyzing an existing system to decompose its software architecture to introduce local recovery. The approach provides systematic guidelines to depict the alternative space of the possible decomposition alternatives, reduce the alternative space with respect to domain and stakeholder constraints and balance the feasible alternatives with respect to availability and performance. We have provided the complete integrated tool set, which supports the whole process of decomposition optimization. Each tool automates a particular step of the process and it can be utilized for other types of analysis and evaluation as well.

We have illustrated our approach by introducing local recovery to the open-source media player, called MPlayer. We have seen that the impact of decomposition alternatives can be easily observed, which are based on actual measurements regarding the isolated modules and their interaction. We have implemented local recovery for three decomposition alternatives to measure the actual availability achieved. Based on the measured availabilities, we have seen that the alternatives were ordered correctly with respect to the heuristic cost function used for optimization.

Acknowledgments We acknowledge the feedback from the discussions with our TRADER project (TRADER, 2011) partners from NXP Research, NXP Semiconductors, TASS, Philips Consumer Electronics, Design Technology Institute, Embedded Systems Institute, IMEC, Leiden University and Delft University of Technology. We thank the anonymous reviewers for their feedback to improve this paper.

References

Aleti, A., Björnander, S., Grunske, L., & Meedeniya, I. (2009). Archeopterix: An extendable tool for architecture optimization of aadl models. In *Proceedings of the ICSE 2009 workshop on model-based methodologies for pervasive and embedded software (MOMPES)*, Vancouver, Canada, pp. 61–71.

Alexander, C. (1964). *Notes on the synthesis of form*. Harvard Cambridge, MA: University Press.

Anquetil, N., Fourrier, C., & Lethbridge, T. (1999). Experiments with clustering as a software remodularization method. In *Proceedings of the 6th working conference on reverse engineering (WCRE)*, IEEE Computer Society, pp. 235–245.

Athon, T., & Papalambros, P. (1996). A note on weighted criteria methods for compromise solutions in multi-objective optimization. *Engineering Optimization, 27*(2), 155–176.

Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing, 1*(1), 11–33.

Bachman, F., Bass, L., & Klein, M. (2003). *Deriving architectural tactics: A step toward methodical architectural design*. Tech. Rep. CMU/SEI-2003-TR-004, SEI, Pittsburgh, PA, USA.

Boudali, H., Sozer, H., & Stoelinga, M. (2009). Architectural availability analysis of software decomposition for local recovery. In *Proceedings of the third IEEE international conference on secure software integration and reliability improvement*, Shanghai, China, pp. 14–22.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture, a system of patterns*. Wiley.

Candea, G., Cutler, J., & Fox, A. (2004). Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation, 56*(1-4), 213–248.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., & Fox, A. (2004b). Microreboot: A technique for cheap recovery. In *Proceedings of the 6th symposium on operating systems design and implementation (OSDI)*, San Francisco, CA, USA, pp. 31–44.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., & Stafford, J. (2002a). *Documenting software architectures: Views and beyond*. Boston, MA: Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2002b). *Evaluating software architectures: Methods and case studies*. Boston: Addison-Wesley.

Patterson, D. et al. (2002). *Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies*. Technical Report UCB/CSD-02-1175, University of California, Berkeley.

Dashofy, E., van der Hoek, A., & Taylor, R. (2002). An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 22rd international conference on software engineering (ICSE)*, ACM, Orlando, FL, USA, pp. 266–276.

Davey, J., & Burd, E. (2000). Evaluating the suitability of data clustering for software remodularization. In *Proceedings of the 7th working conference on reverse engineering (WCRE)*. IEEE Computer Society, pp. 268–278.

Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering, 28*(7), 638–654.

Fenlason, J., & Stallman, R. (2000). GNU gprof: The GNU profiler. Free Software Foundation, http://www.gnu.org.

Gokhale, S. (2007). Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing, 4*(1), 32–40.

Grassi, V., Mirandola, R., & Sabetta, A. (2005). An XML-based language to support performance and reliability modeling and analysis in software architectures. In R. Reussner, J. Mayer, J. Stafford,

S. Overhage, S. Becker, & P. Schroeder (Eds.), QoSA/SOQUA, Springer, Lecture Notes in Computer Science, Vol. 3712, pp. 71–87.

Grunske, L., Lindsay, P., Bondarev, E., Papadopoulos, Y., & Parker, D. (2007). An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In R. de Lemos, C. Gacek, & A. B. Romanovsky (Eds.), *Architecting dependable systems IV* (pp. 188–209). Berlin: Springer.

Harris, J., Hirst, J., & Mossinghoff, M. (2000). *Combinatorics and graph theory*. New York: Springer.

Herder, J., Bos, H., Gras, B., Homburg, P., & Tanenbaum, A. (2007). Failure resilience for device drivers. In *Proceedings of the 37th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. Edinburgh, UK, pp. 41–50.

Heyliger, G. (1994). Coupling. In J. Marciniak (Ed.), *Encyclopedia of software engineering* (pp. 220–228). Wiley.

Huang, Y., & Kintala, C. (1995). Software fault tolerance in the application layer. In M. R. Lyu (Ed.), *Software fault tolerance, chapter 10* (pp. 231–248). New York: Wiley

Hunt, G., Aiken, M., Fhndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., & Wobber, T. (2007). Sealing OS processes to improve dependability and safety. *SIGOPS Operating Systems Review, 41*(3), 341–354.

Jokiaho, T., Herrmann, F., Penkler, D., & Moser, L. (2003). The service availability forum application interface specification. *RTC Magazine, 12*(6), 52–58.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, SEI.

Laprie, J. C., Arlat, J., Beounes, C., & Kanoun, K. (1995). Architectural issues in software fault tolerance. In M. R. Lyu (Ed.), *Software fault tolerance, chapter 3* (pp. 47–80). Cichester: Wiley.

Lung, C. H., Xu, X., & Zaman, M. (2007). Software architecture decomposition using attributes. *International Journal of Software Engineering and Knowledge Engineering, 17*, 599–613.

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 26*(1), 70–93.

Meedeniya, I., Buhnova, B., Aleti, A., & Grunske L. (2011). Reliability-driven deployment optimization for embedded systems. *Journal of Systems and Software, 84*(5), 835–846.

Mitchell, B. S., & Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering, 32*(3), 193–208.

MPlayer (2010). MPlayer official website. http://www.mplayerhq.hu/. Accessed 31 Mar 2011.

Necula, G., McPeak, S., Rahul, S., & Weimer, W. (2002). CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the conference on compiler construction*, pp. 213–228.

Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices, 42*(6), 89–100.

Nguyen, G., Hluchý, L., Tran, V., & Kotocova, M. (2001). DDG task recovery for cluster computing. In *Proceedings of the 4th international conference on parallel processing and applied mathematics*, Springer, Naleczow, Poland, Lecture Notes in Computer Science, Vol. 2328, pp. 369–378.

Object Management Group (2001) Fault tolerant CORBA. Tech. Rep. OMG Document formal/2001-09-29, Object Management Group.

Pareto, V. (1896). *Cours D' economie politique*. Lausanne, Switzerland: F. Rouge

Ross, S. (2007). *Introduction to probability models*. San Diego: Elsevier Inc.

di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., & Pierantonio, A. (2010). Developing next generation ADLs through MDE techniques. In *Proceedings of the 32nd international conference on software engineering (ICSE)*, Cape Town, South Africa, pp. 85–94.

Ruskey, F. (1993). Simple combinatorial gray codes constructed by reversing sublists. In *Proceedings of the 4th international symposium on algorithms and computation (ISAAC)*, Springer, Lecture Notes in Computer Science, Vol. 762, pp. 201–208.

Ruskey, F. (2003). Combinatorial generation. University of Victoria, Victoria, BC, Canada, manuscript CSC-425/520

Santos, G., Duarte, A., Rexachs, D., & Luque, E. (2008). Increasing the performability of computer clusters using RADIC II. In *Proceedings of the third international conference on availability, reliability and security*, IEEE Computer Society, pp. 653–658.

Sozer, H., & Tekinerdogan, B. (2008). Introducing recovery style for modeling and analyzing system recovery. In *Proceedings of the 7th working IEEE/IFIP conference on software architecture (WICSA)*. Vancouver, BC, Canada, pp. 167–176.

Sozer, H., Tekinerdogan, B., & Aksit, M. (2009). FLORA: A framework for decomposing software architecture to introduce local recovery. *Software: Practice and Experience, 39*(10), 869–889.

Teitelbaum, T. (2000). Codesurfer. *SIGSOFT Software Engineering Notes, 25*(1), 99.

Tekinerdogan, B., Sozer, H., & Aksit, M. (2008). Software architecture reliability analysis using failure scenarios. *Journal of Systems and Software, 81*(4), 558–575.

TRADER (2011). Trader project, ESI. http://www.esi.nl/projects/trader. Accessed 31-March-2011.

de Visser, I. (2008). Analyzing user perceived failure severity in consumer electronics products. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands.

White, J., Doughtery, B., Strowd, H., & Schmidt, D. (2009). Creating self-healing service compositions with feature models and microrebooting. *International Journal of Business Process Integration and Management, 4*, 35–46.

Wiggerts, T. (1997). Using clustering algorithms in legacy systems remodularization. In Proceedings of the 4th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, pp. 33–43.

Yacoub, S., Cukic, B., & Ammar, H. (2004). A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability, 53*(14), 465–480.

Author Biographies

Hasan Sözer received his B.Sc. and M.Sc. degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a post-doctoral researcher at the University of Twente. He is currently an assistant professor at Özyegin University.

Bedir Tekinerdoğan received his MSc degree in Computer Science in 1994, and a PhD degree in Computer Science in 2000, both from the University of Twente, The Netherlands. From September 2003 until September 2008 he served as an assistant professor at University of Twente. Currently he is an assistant professor at Bilkent University in Turkey.

Mehmet Akşit holds an M.Sc. degree from Eindhoven University of Technology and a Ph.D. degree from the University of Twente. Currently, he is working as a full professor at the Department of Computer Science, University of Twente and affiliated with the institute Centre for Telematics and Information Technology. He worked for Océ Nederland from 1981–1982 and 1983–1987. As a visiting scientist, in 1989 he was at the IBM T. J. Watson Research Laboratory, New York, in 1993 at the University of Tokyo, and in 1994 at the New Jersey Institute of Technology. He is currently the head of the Software Engineering chair at the University of Twente.