Architectural Views through Collapsing Strategies

Christoph Stoermer
Robert Bosch Corporation
2 North Shore Center
Pittsburgh, PA 15213 USA
+1 412 323 6009
Christoph.Stoermer@rtc.bosch.com

Liam O'Brien
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213 USA
+1 412 268 7727
lob@sei.cmu.edu

Chris Verhoef
Free University of Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
+31 20 4447760
x@cs.vu.nl

ABSTRACT
Architectural views help to better understand and analyze software from particular stakeholder perspectives. Views are abstractions that are generated in an architecture reconstruction effort with collapsing strategies. Collapsing is the mechanism to aggregate detailed source information into architectural elements that constitute the architectural views. The elements are presented in a particular viewtype and style. Traditional software architecture reconstruction tools assume that source elements are collapsed into mostly one container. However, the Satellite Tracking System case study, outlined in this paper, required the introduction of multi-collapses. Multi-collapses allow the aggregation of one element into multiple containers. Multi-collapses are either the result of applying incorrect collapsing strategies or an excellent starting point for software analysis to gain better understanding of existing software. We describe implementation and visualization aspects of multi-collapses within an architecture reconstruction environment.

Keywords
Aggregation, Architecture, Architecture Reconstruction, Collapsing, Components and Connectors, Modules, Multi-Collapsing, View, Viewtypes, Viewstyles.

1 INTRODUCTION

Software architecture is an important vehicle for stakeholder communication in organizations [1]. However, software architectures are often hard to understand. Some reasons are that documented architectures do not conform to the implemented architectures, stakeholders do not find their particular views in the documentation, architecture experts are not available for interviews, or poorly documented components have to be reused. These examples are driving factors for organizations to uncover software architectures from existing sources in order to increase architectural understanding among stakeholders.

Architecture reconstruction is the process by which architectural views of an implemented system are obtained from existing artifacts. One key mechanism for architectural reconstruction is collapsing. Collapsing is a mechanism to aggregate detailed source information into higher levels of abstraction, of which software architectures are a prime example. For example, many systems use name conventions to express what in fact are architectural aspects. A good collapsing strategy is to combine source elements, such as functions, that satisfy a particular naming convention to recover the intended architectural aspects. More general, collapsing is achieved by clustering of related parts [9], lattice partitioning [17], and aggregation into containment-hierarchies [7]. Today, a rich source for collapsing strategies is available in the reverse engineering community. For example, the composition of subsystem structures by using (K,2)-Partite Graphs [12], and the generation of system descriptions from source code alone [16] describe a variety of collapsing methods. The main contribution of this paper is to investigate the effect of common existing collapsing strategies on architectural views with particular focus on multi-collapses.

Considering these strategies, it is important to understand that collapsing is not a purpose in itself but rather requires an initial concept for the abstractions to be generated. In other words, collapsing requires a goal-driven approach. The goals are determined by the stakeholders requiring the reconstruction. Their goals are mainly driven by quality attribute scenarios, or impact scenarios. For the purpose of this paper we assume a goal-driven approach as, for example, outlined in [2], [13], and [14]. The scenarios require appropriate architectural views of the existing system with particular notations and styles [4].

The work described in this paper is the result of an architecture reconstruction that we applied for a Satellite Tracking Agency (STA). The reconstruction was carried out on the Satellite Tracking System (STS) in a multi-language context of C, C++, and Fortran. During the application of collapsing strategies to generate architectural views we detected the need for multi-collapses. Multi-collapses are entities - such as functions and variables - that are not uniquely assignable to a particular architectural element, such as a layer. The major disadvantage in the occurrence of multi-collapses is the resulting uncertainties because collapsing strives to assign elements uniquely in top-down hierarchies, such as a module hierarchy consisting of a system, sub-systems, layers, modules, etc.

The uncertainties include the question for ownership of multi-collapses and associated responsibilities for allocation, initialization, and de-allocation of resources. However, multi-collapses also have advantages, for example, the visualization of a system from a data perspective, where all elements that access or define a variable are collapsed into the corresponding data container. In this case, a function that accesses several variables will be collapsed into several data containers.

Multi-collapses are either the result of applying incorrect collapsing strategies or an excellent starting point for software analysis to gain better understanding of the existing software or particular aspects, such as cross-cutting concerns. Based on this assertion we implemented collapsing and visualization support for multi-collapses. The principles of multi-collapses are widely applicable and can be implemented in many reconstruction tool environments. In this case we used the reconstruction environment ARMIN (Architecture Reconstruction and MINing) [10].

The remainder of the paper is organized as follows. Section 2 introduces several collapsing strategies involving multi-collapses in a simple example setting. The case study for the Satellite Tracking Agency in Section 3 outlines a set of architecture views and contains a discussion of the related collapsing strategies. Section 4 continues with a brief description of an implementation approach to multi-collapses in an architecture reconstruction environment. Finally, we outline our future work in Section 5 and summarize the results in Section 6.

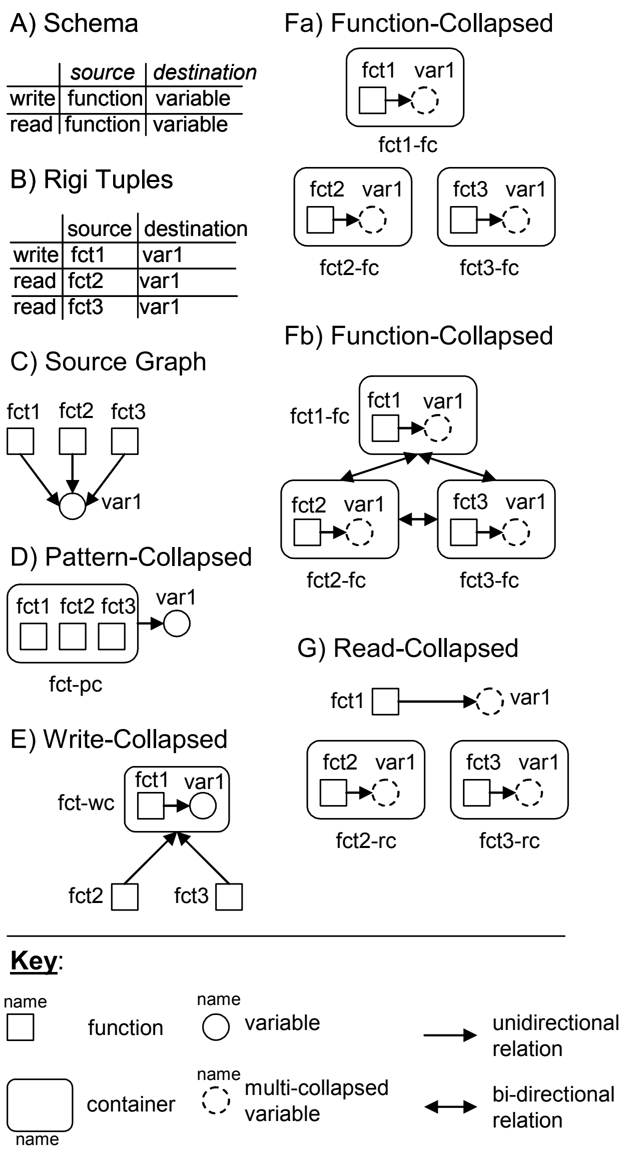
2 COLLAPSING

Collapsing is an essential mechanism in architecture reconstruction. We will demonstrate this mechanism and the various facets of it by introducing a simple example, as illustrated in Figure 1. The example consists of a schema (Figure 1-A), extracted source facts (Figure 1-B), a source graph (Figure 1-C), and several graphs that are generated from the source graph as a result of particular collapsing strategies (Figure 1-D, 1-E, 1-F, and 1-G). We will explain and discuss Figure 1 in the following sub-sections.

Laying the Foundation

Figure 1-A: Schema. The schema consists of the relation types that are extracted from sources of the example. Bowman, et al., [3], van Deursen and Riva[17], the Dagstuhl Middle Model [5] and others have outlined schemas for identifying what entities and relations should be extracted from a system to assist the process of architecture reconstruction. In this case, we use a simple subset of relations consisting of write and read relations between source and destination. The source is a function and the destination is a variable.

Figure 1-B: Rigi Tuples. The relation types, as defined in the schema, are extracted from the source. The extracted facts are represented in Rigi standard format [11]. The write relation in Figure 1-B represents an extracted write access of a function with the name *fct1* on a variable with the name *var1*. The facts are extracted from sources by parsing and analyzing the sources. For a detailed treatment of the extraction process we refer, for example, to [8].



	source	*destination*
write	function	variable
read	function	variable

	source	destination
write	fct1	var1
read	fct2	var1
read	fct3	var1

Figure 1: Collapsing Example

Figure 1-C: Source Graph. The Rigi tuples from Figure 1-B can be used to generate a graph. The graph G = (N, R) contains the extracted source facts and their relations, where the nodes N represent entities, such as functions (rectangles) and variables (circles), and the relations R represent write and read edges (directed arrows) between the nodes.

Four Collapsing Strategies

The remaining five graphs in Figure 1 introduce collapsing strategies to aggregate detailed source facts into higher levels of abstraction. The aggregated source facts are merged into containers that are represented as rounded rectangles in Figure 1.

Figure 1-D: Pattern-Collapsed. All entities of type function in source graph G, beginning with the pattern "fct", are collapsed in a container. The result of this strategy is graph G^D, where the relations of each function to variable *var1* are aggregated in a relation between the new container, for example *fct-pc* (*pc* is an abbreviation for pattern collapsed), and *var1*. A motivation for this particular case could be the aggregation of all functions that share coherent functionality, for example all functions of a user interface component.

Figure 1-E: Write-Collapsed. Entities are collapsed along a relation type, in this case the *write* relation. The destination entity (3rd item in the tuple) for a given relation type and source entity (2nd item in the tuple) is aggregated in a new container. The entities *fct1* and *var1* are collapsed into a container named *fct-wc* (*wc* is an abbreviation for write-collapsed). As a result, the relations of *fct2* and *fct3* to *var1* are redirected to *fct-wc*. A motivation for this particular case could be the segmentation of variables and functions to form a cohesive block in a reengineering environment.

Figure 1-Fa: Function-Collapsed. All descendants of the entity type function that are children and optionally grandchildren etc., are collapsed into containers. The relation type is insignificant. The trigger for this collapsing is the source (2nd item in the tuples). All tuples of this example have a function as their source. Consequently, the unique collapse of *var1* into exactly one container is not possible. Instead, *var1* is cloned into the containers *fct1-fc*, *fct2-fc*, and *fct3-fc* (*fc* is an abbreviation for function-collapsed), as well as the relations of the functions to *var1*. We name collapsing of entities and relations **multi-collapsing** when there is no unique assignment to a container possible. The term **multi-collapses** refers to those entities. The multi-collapses in Figure 1-Fa are illustrated as dashed circles. A further interesting characteristic of multi-collapses in Figure 1-Fa is the lack of relations between the containers. The resulting graph G^{Fa} pretends that the containers have no relations among each other. Obviously, the relation of the functions to *var1* can be resolved inside of each container.

Figure 1-Fb: Function-Collapsed. An alternative collapsing strategy to Figure 1-Fa would be to add a relation between an entity and each instance of the multi-collapsed item. Figure 1-Fb illustrates this alternative by adding relations from the functions to each *var1* instance, which eventually produces a fully connected graph between the containers. This alternative leads to an explosion of relations in settings where there are large amounts of data, with the consequence of producing cluttered graphs. The reduction of relations can be useful for aggregations where multi-collapses and their relations are negligible on the hierarchy level of the resulting graph.

Figure 1-G: Read-Collapsed. This strategy produces a graph with multi-collapses in the containers *fct2-rc* and *fct3-rc* (*rc* is an abbreviation for read-collapsed), as well as in the resulting graph G^G. The relation between *fct1* and *var1* in each container cannot be resolved because it is unclear to which container the relation should go. Both effects occur:

(1) Introduction of multi-collapses inside and outside of containers

(2) Disappearance of relations between containers

The relation between *fct1* and *var1* inside of the containers cannot be resolved because it is unclear to which container the relation should go. As already discussed in Figure 1-Fb, an alternative for adding a multi-collapse in the resulting graph would be the introduction of two relations from *fct1* to both containers, which does not scale well for large amounts of data and reduces the understanding of the resulting graph. The collapsing strategy in Figure 1-G illuminates the write relation but hides the read-relations.

Container Types and Names

The Figures 1-D, 1-E, 1-Fa, 1-Fb, and 1-G use containers for the aggregated source facts. Interestingly, the type "container" is not part of the schema as illustrated in Figure 1-A. The container is implicitly assumed as a built-in container type for entities and relations. The disadvantage of this approach is that containers have no explicit types. For example, the container in Figure 1-D could be referred in follow-up aggregations as a container of type layer, because the architects of the system envisioned a particular set of coherent source artifacts as a layer. Containers of type layer could be collapsed in further aggregations into containers of type subsystem.

An additional issue is the assignment of names to containers. Manually assigning names does not scale for large systems. A pragmatic solution is the generation of unique names that are coupled with the collapsing strategy, such as *fct2-rc* in Figure 1-G, where *rc* denotes the read-collapsed strategy.

Multi-Collapsing

There are three principal ways to collapse entities and relations: collapsing along

- pattern matching operations (see Figure 1-D)
- relation types (see Figure 1-E and 1-G)
- entity types (see Figure 1-F)

Collapsing along patterns allows the aggregation of entities with certain characteristics, such as matching of regular expressions, or operations known from RPA (Relation Partition Algebra [6]). Further useful collapse operations are possible. For example, collapse operations along runtime properties, such as execution time, to separate time critical paths from non time-critical paths. For the purpose of this paper it is sufficient to subsume them under pattern matching.

We identified three characteristics of multi-collapses:

1. Multiple occurrence of entities
2. Disappearance of relations between containers
3. Uncertainties with respect to ownership and responsibilities

As mentioned in the Introduction, collapsing strives to assign elements uniquely in top-down hierarchies, such as a module hierarchy consisting of a system, sub-systems, layers, modules, etc. In most cases it is the effort to reconstruct decompositions of a system that consists of elements with unique responsibilities. Multi-collapsed elements challenge this effort by raising the question for ownership and associated responsibilities, such as for allocation, initialization, and de-allocation of resources. A common strategy to prevent multi-collapses in module hierarchies is the assignment of entities to separate containers, such as libraries or using different aggregation strategies. For example, a public library function is typically used by several functions. The wrong collapsing strategy would be to aggregate the library function inside of the caller functions. However, sometimes multi-collapses are unavoidable, sometimes intentionally desired, and sometimes an excellent starting point for further analysis. Consider the following cases:

1. The visualization of a system from a data perspective, where all elements that access or define a variable are collapsed into a data container. In this case, a function that accesses several variables will be collapsed into several data containers. In this case, multi-collapsed functions are a good starting point to analyze information hiding implementations.
2. The visualization of a call-graph, where all information related to a particular function is collapsed into its function container, such as the function defined by a file or an accessed variable. Of course, a file could define several functions. Therefore, the collapsing strategy produces a file that is multi-collapsed into several function containers. The call-graph shows the function containers and the call relations between them. The sub-graphs of each function container show the related entities that use or are used by the function. In this case, the collapsing strategy allows the analyst to navigate hierarchical functions from the top level graph, and the exploration of dependencies in the sub-graphs of each function container.

We will refer to these examples in the discussion of architectural views in the case study below. For now, it is sufficient to capture the result that multi-collapses have advantages as well as disadvantages depending on the architectural views to be generated.

Goal-Driven Collapsing

The simple collapse strategies of Figure 1 illustrate how similar operations produce different resultant graphs and different conclusions when viewing these graphs. The collapsing strategy illuminates an aspect, such as write-relations in Figure 1-G, or hides an aspect, such as the read-relations in Figure 1-G. It is therefore essential to develop a concept about views and their interpretation for a particular system before performing collapsing operations. We have often had the experience, that the development of a schema is driven by the capabilities of particular extractors for particular languages. However, the development of a schema is intertwined with the development of a collapsing strategy to achieve reconstruction goals. Reconstruction goals are often motivated by a top-down perspective, such as the investigation of change or impact scenarios.

3 CASE STUDY

The first time we consciously detected multi-collapses as a useful mechanism was during a case study for the Satellite Tracking Agency (STA). The STA supports efforts to develop, acquire, and deploy satellite-tracking systems. In this case, the STA wanted to better understand the architecture of one of its legacy systems, the Satellite Tracking System (STS), to be able to port the system to a new environment. The STS consists of about 500KLOC, which is a mixture of C, C++, and Fortran that currently runs in a Silicon Graphics environment. The system has been in operation for many years and certain people know parts of the system for which they are responsible very well, though no one knows the architecture of the entire system, thus the need to apply architecture reconstruction techniques.

The STS is a classified system and access to the system and any information about it is tightly controlled. Consequently, the reconstruction of the real system was performed by the developers of the STA. The architectural views and the collapsing strategies outlined in this paper were developed on a sanitized version of the information extracted from the system: the developers manipulated the extracted source artifacts by sanitizing the entity names. These collapsing strategies were applied by the developers on the real STS system to generate a set of architectural views. Although this time-consuming process produced a lot of overhead in effort and communication, it enabled STA to perform architecture reconstructions on further system versions by themselves.

We will focus in this case study on the usage of architectural views, multi-collapses, and useful collapsing strategies for generating the appropriate views. The concept of viewtypes and their associated styles is taken from [4]. In the following we will introduce the initial STS concept, the schema, and the source extraction, before we describe the view generation.

Initial Concept

Architecture reconstruction is hard to achieve without any initial concept about the architecture of the existing system. Typical ways to obtain the initial concept are interviews with the architects and reading of documentation. Using interviews, we sketched the concept as illustrated in Figure 2.

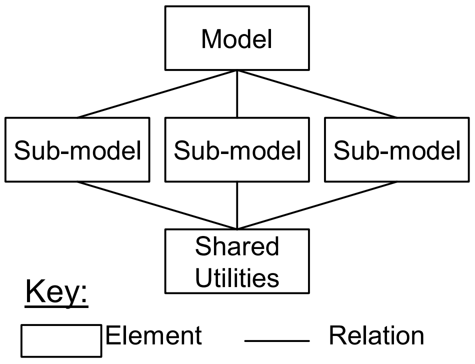


Figure 2: Initial Concept

In this concept a model relates to a set of sub-models that have shared utilities. The model is more abstract than a sub-model. For example, Weather could be a model which contains sub-models for different types of weather conditions. The initial concept does not have to be perfect. Figure 2 leaves a couple of open questions: How many models exist? Does a model refer exactly to three sub-models? What is the relation between models? The initial concept and terminology should reflect the current thoughts of the developers, or other stakeholders, about their system. The concept does not necessarily have to conform to the as-implemented system, or provide a consistent terminology.

In addition to Figure 2, the concept comprises a cyclic executive that dispatches models in a particular sequence. The communication between the models is done via message passing.

Based on the interviews and the initial concept we identified the architectural viewtypes Allocation, Module, and Component-and-Connector (C&C) with their associated viewstyles which we summarized in Table 1.

Viewtype	Viewstyle	Comment
Allocation	Implementation	File structure of the implementation
Module	Decomposition	Partition into manageable pieces
C&C	Shared-Data	Producer consumer
	Communicating-Processes	Component communication

Table 1: Views to be generated

The Schema

Major parts of the schema for the STS reconstruction are illustrated in Figure 3. The system consists of models, which consist of files, etc. Note that the schema notation is different than the table presentation in Figure 1-A. The schema in Figure 3 contains information about the multiplicity of relations. For example, a file is assigned to exactly one directory, whereas a directory might contain several files. Collapsing files into directories should not, according to the schema, generate multi-collapses. One exception from this rule would be the occurrence of symbolic links, which were not used in the STS case. The occurrence of multi-collapses would hint to either false positives of the source extractor or incorrect assumptions in the schema. However, collapsing directories into files would generate multi-collapses as long as there is more than one file in a directory.

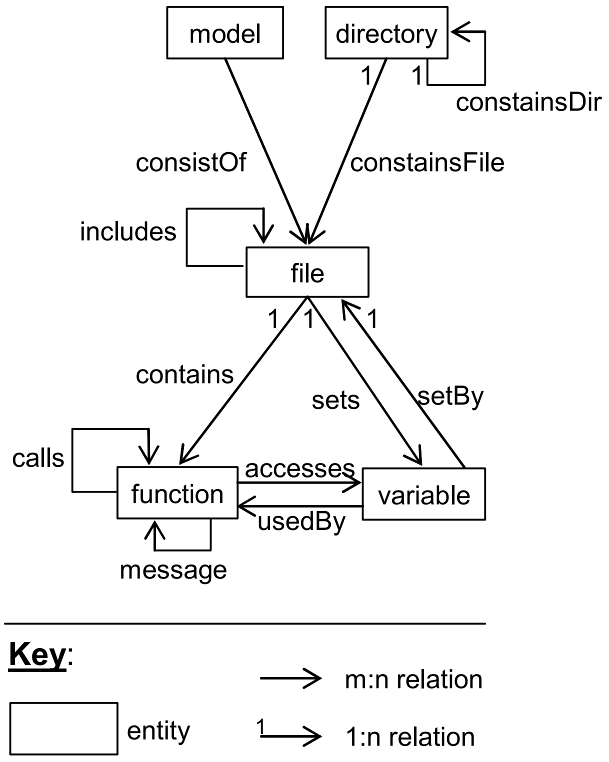


Figure 3: STS Schema

Multi-collapses do not occur when destinations are collapsed into sources in a 1:n relation between source and destination. This rule is easy to follow with entity and relation collapsing strategies. Pattern-collapsing requires more attention because the collapsing criteria, formulated in pattern(s), have to filter an element for exactly one container to avoid multi-collapses. Figure 3 shows that unique collapses are typically possible along containment relations (*containsDir*, *containsFile*, *contains*) and defines (*sets*, *setBy*) relations.

Models do not have a 1:n relation with file in the schema of

Figure 3. The reason is that, for example, particular include files or library files are used in several models.

The schema defines relations that build the foundation for the reconstruction. Other relations are possible but were not considered in this case because they did not add significant architectural information to the identified documentation viewtypes. The selected entities and relations are extractable from the C/C++ as well as from the Fortran sources.

Source Extraction
The developers of the STA used the commercial tool "Understand for C++ and Fortran" [15] to extract the source information according to Figure 3. The extracted information had to be converted into the Rigi Standard Format with a simple Perl [19] script. The Rigi tuple format is widely used among reconstruction tools.

Allocation Viewtype
The allocation viewtype targets the interaction of the software architecture with its environment. The software architecture is mapped onto a file system, onto hardware structures, and onto project management structures. Consequently, the styles of the allocation viewtype are implementation, deployment, and work-flow assignment.

Every architecture reconstruction effort requires an implementation viewstyle. This style identifies the file structure and the associated file versions relevant for the reconstruction. The file structure is analyzed, parsed, compiled, and related to runtime information of the models. The source extraction process is already performed with an initial concept and a schema in mind. Source extraction already selects and compresses particular information worth extracting for subsequent collapsing steps.

Presentation: Presenting all relations in one graph produces a source graph as illustrated in Figure 1-C. The resultant cluttered graph of around 10000 entities and 31500 relations is not particularly useful. However, the entities and relations can be filtered by reconstruction tools in a way that only file and directory entities and their relations (*containsDir* and *containsFile*) become visible. Applying a hierarchical layout on the graph produces a view of the directory and file hierarchy. Other allocation views are possible. For example, the file and variable information in the source graph can be filtered to present files that define global variables.

The allocation viewtype provides the source foundation for the Module and C&C viewtypes. Therefore, it is a good rule of thumb to check parts of the source graph for consistency with the source in order to validate the source extraction process. This is especially the case if several source extractors are used.

Module Viewtype
Modules in an architecture reconstruction context comprise logical elements, sometimes referred to as conceptual design elements, and implementation language elements, sometimes referred to as concrete design elements:

- Logical elements – are constituted by the architects and designers of the system. Examples are layer, client-server, subsystem, program, etc.
- Implementation language elements – are defined by the selected implementation language. Examples are packages, files, classes, objects, functions, etc.

Typically, there is a mapping between logical elements and implementation language elements. For example, a layer might be mapped to a Java package. It is a common trend to erase informal mappings by introducing top-down design languages, such as particular domain languages, and by providing bottom-up richer abstractions in implementation languages. However, logical elements might be mapped onto several implementation languages. In this case a more general abstraction of the implementation language model is required.

The fundamental style of a module viewtype is the decomposition style. The modules in a decomposition style follow a hierarchical organization principal. For example, a program is composed of packages, a package is comprised of files, and a file defines functions and variables. Functional decomposition avoids ambiguous ownerships or uncertain responsibilities for elements in a module hierarchy. A module might be used by other modules; however, the ownership is typically unique. Therefore, collapsing strategies for functional decomposition use containment relations, such as *is-part-of*, *contains*, or *defines* relations. Entities (elements) are uniquely aggregated. Consequently, the occurrence of multi-collapses hints to a collapsing strategy that should be reviewed carefully.

Figure 3 illustrates containment relations for the STS system: *consistOf*, *containsFile*, *containsDir*, and *contains*. The containment relation for a variable is unclear. The *sets* relation doesn't deduce a containment statement. Therefore, variables would be root elements in a containment hierarchy of the decomposition style. The organization used the *sets* relation as a combination of a contains and write access to local variables. These combinative relations are difficult to use in a collapsing strategy for decomposition styles, and should be avoided where possible. For non-functional-decomposition styles combinations might be useful, for example, an aggregation of read and write relations to an access relation in a uses style.

As mentioned before, all entity types of a schema have a unique containment hierarchy, that is: for all entity types, except the root type, exists a 1:n relation to a parent entity. This is not the case in Figure 3. There are two root types: directories and models. A collapsing strategy that aggregates variables and functions into files, files into directories, and directories into models is not possible, because there is no relation between directories and models. Instead of aggregating files into directories, we aggregated directories into files before the files were aggregated into models. The obvious reason is that the models are the primary interest in the architecture module view. The consequence is that some directories occur as multi-collapses in the file containers. An alternative strategy is that directories are not considered in the module view. The disadvantage is the missing directory information in the graph navigation during the view analysis.

Presentation: Figure 4 illustrates a drill down view of the generated module view containing the files, functions, and variables of the sub model eKXvoCYLzFpxL in a grid layout. The relations between the entities are calls, include, and accesses relations. The dashed rectangles in Figure 4 are the multi-collapsed entities. Whereas the function, variable, and file collapsing can be performed using entity collapsing strategies, the model aggregation had to follow a pattern strategy. The pattern-collapsing strategy used particular knowledge about the system, such as naming conventions for files in particular models. As a rule of thumb, containment hierarchies should be provided by the design specification and guidelines for the implementation language. A tedious process is the manual assignment of files to models, which typically is a sign of creating an artificial design in the absence of an explicit design.

With the help of the generated module view it is now possible to review the initial concept that we illustrated in Figure 2. The previously mentioned open questions, such as the relation of models to sub-models, can now be answered and further analyzed, such as for reengineering purposes.

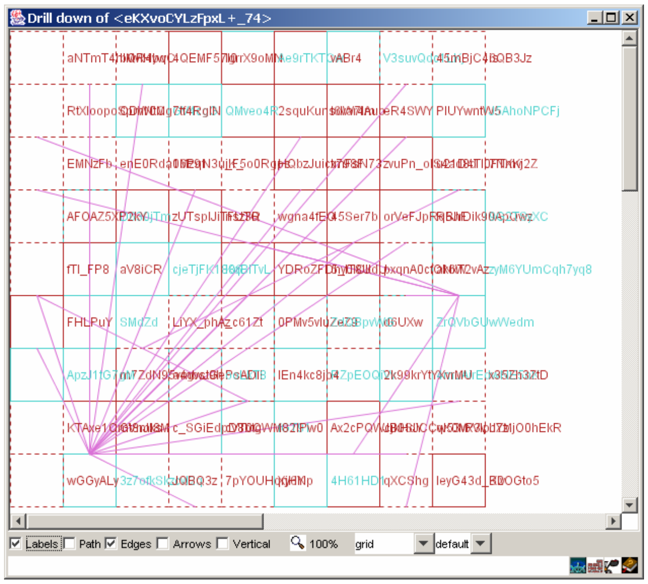


Figure 4: Sub-graph of Sub Model eKXvoCYLzFpxL

According to the definition of a decomposition style [4], Figure 4 does not provide a pure decomposition style. It is a combination of a decomposition style with a uses style because calls and accesses relations provide in many cases uses-information. The filtering of these non-containment relations can easily produce a pure decomposition style view. Finally, the functional decomposition style might be mixed with a layered style by organizing the modules according to a layer rule, for example, models, sub-models, and shared utilities.

C&C Viewtype

C&C views define presentations consisting of elements that have some runtime presence, such as threads, clients, shared data storage, and information flows. The authors of [4] propose six C&C styles: pipe-and-filter, shared-data, publish-subscribe, client-server, peer-to-peer, and communication-processes style. We used in the STS case study the shared-data and communicating-processes style.

Shared-Data Style

A decomposition style along system functionality is not the only way to present system partitioning. Understanding decomposition can also be achieved by collapsing objects or data. In this sense, a module is not necessarily a functional package but rather a cluster of, typically cohesive, information. In the STS case study, the data is shared between models implemented in different programming languages. The shared data is not structured, such as a hierarchy of objects, or provided with an explicit access pattern that is used by the models, such as a common access interface with discovery mechanisms for data modifications. In this case the shared data is simply a global data space.

The components in the shared-data style are the global data space with its data and the data accessors. The connector relations are the *setBy* and *usedBy* relations of the schema in Figure 3. Interestingly, the *sets* and *accesses* relations are not useful because they do not have a variable as a source. Using the sets and accesses relations would result in hiding the data inside of files and functions. However, in this case the STA wanted to visualize the system from a data perspective, and not from a function, file, or directory perspective.

Presentation: The components of the C&C view are the variables; connectors are the *setBy* and *usedBy* relations. The collapsing strategy consists of two steps:

1. Aggregate functions, files, and directories into variables
2. Aggregate variables along model boundaries that were identified in the module view

Again, considering only the first step would result in a cluttered graph. The second step tries to logically arrange variables in clusters with model boundaries. The analysis showed the presence of functions as multi-collapses, which is not surprising. The data is not organized in a repository with particular access functions but rather shared as a non-structured global data space. The multi-collapsed functions are useful initial indicators to search for conflicting variable accesses and conflicting assumptions about the variable content between the models. They are "initial" indicators, because, of course, encapsulated data in set and get methods do not necessarily solve conflicting situations on a logical level.

The shared data style was of particular interest for the STS developers. One of the intentions is to reengineer the system in the future. An initial start is to identify coherent pieces and get and set methods for individual data pieces. The identification of an object-oriented data model might be a goal further down the road, where data objects with their associated methods become the main form of interaction with the data.

The shared-data style example shows that multi-collapses can be a useful instrument to analyze existing code and are not an undesirable side-effect as opposed to the decomposition style of the module viewtype.

Communicating-Processes Style
Information between models is communicated via messages. Messages are sent via particular functions that are offered by the infrastructure of the STS. One reconstruction task was to analyze the message flow between the models.

The style to describe the message flow between components is the communicating-processes style. The components of this style are potentially in parallel executable units of concurrency, such as threads, processes, and tasks. The connectors enable the information exchange between the units of concurrency.

The messages in the STS case are transmitted between functions. The associated relation *message* is described in Figure 3: *message function function*. The disadvantage of this tuple-format is that the message itself is missing. Therefore, we annotate the tuple-format with an additional attribute:

message fct1 fct2 _msg:MSG1

The attribute *_msg:* signifies that a particular message, in this case *MSG1*, is passed between *fct1* and *fct2*. There might be several messages exchanged between *fct1* and *fct2*. In this case we either have to add a further line with the additional message or we have to use a comma separated list of messages. Still, we do not know the content of the message MSG1. A static solution provides the following annotation:

sets file1 MSG1 _line:150 _col:8

The attributes *_line:* and *_col:* provide the information where the message *MSG1* is defined in file1. In case where messages are differently treated than variables, a new relation might be added, such as

defines_message file1 MSG1 _line:150 _col:8

Another approach is the tracing of messages between components at runtime by code instrumentation. The tracing approach is especially useful for message sequence and message content analysis for particular system usage scenarios. However, for the purpose of the STS system it was sufficient to analyze the type of messages between models.

Presentation: The developers of the STS system wanted to have the message functions as components and the messages themselves as connectors in the C&C view. The collapsing strategy consists of the following steps:

1. Remove functions that do not send or receive messages
2. Aggregate directories, files, and variables into functions
3. Aggregate functions along model boundaries that were identified in the module view

Step 3 was an additional step to illustrate the message flow on a more abstract level of models. However, step 2 was sufficient because it turned out that only a few application functions of the STS send or receive messages.

4 AN IMPLEMENTATION FOR COLLAPSING
In this section, we will explain an implementation approach for multi-collapsing. We incorporated the approach in our architecture reconstruction tool ARMIN [10]. Key elements of the implementation are:

- A two-dimensional matrix
- The collapsing algorithm
- Scripting elements

In addition, graph operations, such as adding and deleting of nodes and edges, creation of sub-graphs, and search operations, are required. In the following we will explain the purpose of the matrix, the collapsing algorithm, and the basic elements for the scripting language.

Matrix
The matrix contains the data about which elements should be collapsed in which containers. The matrix consists of two columns where the first column contains the name of the containers to be generated, and the second column lists the elements that should be collapsed in the container. Two example matrices are listed in Figure 5. Matrix *M_1* contains the elements to generate a graph as illustrated in Figure 1-D, and matrix *M_2* contains the elements to generate the graph of Figure 1-G.

$$\begin{array}{rcc} & \text{Column 1} & \text{Column 2} \\ & \downarrow & \downarrow \\ M\_1 = & \text{fct\_pc} & \text{fct1, fct2, fct3} \end{array}$$

$$M\_2 = \begin{bmatrix} \text{fct2\_rc} & \text{fct2, var1} \\ \text{fct3\_rc} & \text{fct3, var1} \end{bmatrix}$$

Figure 5: Matrix Examples

The matrices *M_1* and *M_2* consist of typed elements. For example, *fct1* is not a string element in the matrix but rather an element of type function with the name "fct1". This ensures that elements with identical names but different types are distinguishable. Elements with identical names and equal types have to be distinguishable by different scopes, such as a C++ name scope, or a file-path as a name prefix.

Collapsing Algorithm

We will introduce the collapsing algorithm by explaining the generation of the graph in Figure 1-G. The collapsing is performed in four steps which are illustrated in Figure 6.

Figure 6-Step1: In this step the entities from the source graph are collected in the way as described above:

- The containers to be generated are listed in column 1 of the matrix
- Column 2 contains all elements of the source graph that should be collapsed in the corresponding container

Figure 6-Step 2: A new graph G' with the containers that are given in the first matrix column is created.

Figure 6-Step 3: The third step traverses the nodes of the existing source graph and analyzes the nodes in the following way:

- Nodes that appear more than one time in the second matrix column are marked as multi-collapses and cloned in each corresponding container. For example, the bold ***var1*** elements in *M_2* of Figure 6-Step 1.
- Nodes that are elements in the second matrix column are moved into the corresponding container of the first column.
- Nodes that are already multi-collapses from earlier collapse operations remain multi-collapses.
- Remaining nodes that do not appear in the matrix, are moved into the new graph. For example, the element *fct1*.

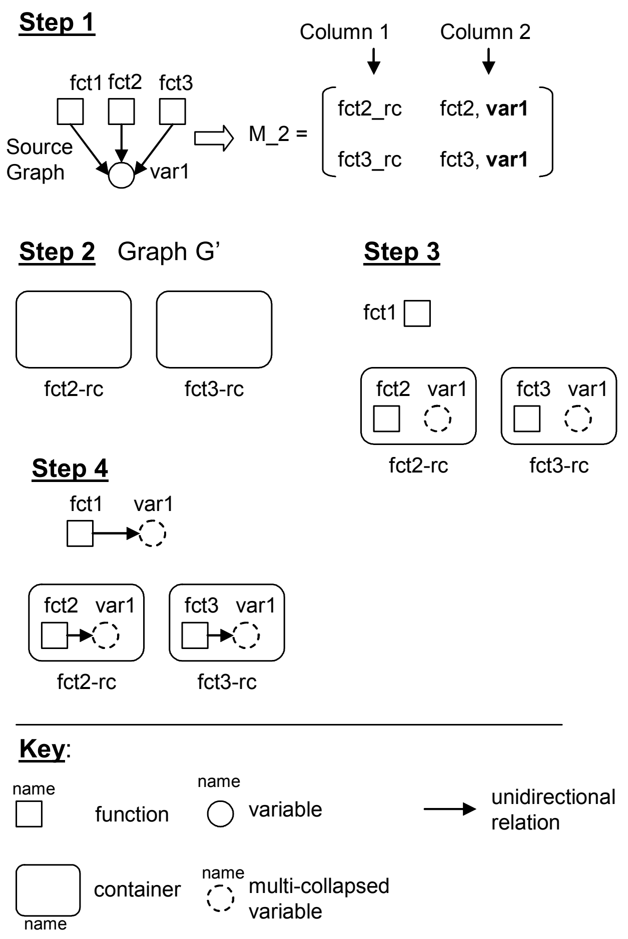


Figure 6: Example for Collapsing Algorithm

Figure 6-Step 4: The last step traverses the edges of the existing source graph and creates or collapses edges accordingly. Three particular cases are analyzed:

- Edges with multi-collapses as destination nodes guarantee that the multi-collapse exists in the corresponding graph of the edge's source and creates edges accordingly. For example, the element *var1* is added in Figure 6-Step 4.
- Edges with multi-collapses as source and destination guarantee that the destination multi-collapses exist in all graphs with the source multi-collapse. Edges are created accordingly.
- Edges with multi-collapses as source nodes guarantee that the multi-collapse exists in the corresponding graph of the edge's destination and creates edges accordingly.

The last step ensures that multi-collapses are present in the resulting graph and all new sub-graphs where they are referred to, either as source, destination, or both.

Our current experience resulting from the case study is that the collapsing algorithm might turn into a performance bottleneck for large graphs with many multi-collapses. This is caused by a potential explosion of nodes and edges. One mitigation strategy we used in the implementation is a lazy sub-graph allocation regarding necessary visual information. However, the structural information has to be generated for all sub-graphs.

Scripting Elements

The matrix contains the data for the collapsing algorithm, as previously mentioned. The various collapsing strategies require a rich set of operations to generate the matrix. A manual generation of the matrix does not scale for larger graphs. For example, the STS example generated a matrix with approx. 9000 rows (number of functions) and two columns when collapsing variables into functions. For automatic generations we implemented a descendant function. This function identifies children and optionally grandchildren etc. of entities in the tuple file and arranged them in a matrix. For example, the command sequence of a scripting language to generate matrix *M_2* of Figure 6-Step 1 could be:

```
1: $M_2 = descendants (system.types.read);
2: collapse($M_2);
```

Line 1 advises the interpreter of the scripting language to collect all destinations of read relations into its corresponding source and transfer the result into the matrix variable $M_2. Line 2 executes the collapsing by transforming the source graph into the graph illustrated in Figure 6-Step 4.

Complex collapsing patterns might require several operations to generate the matrix, such as concatenation, list, or matching operations.

A useful further feature of the scripting language is the access to graphs. For example, the command sequence above could be written as:

```
3: $M_2 = $GSource.desc(system.types.read);
4: $GNew = $GSource.collapse($M_2);
```

Line 3 obtains the destinations of all read relations into its corresponding source in the graph $GSource. Line 4 advises the interpreter to store the collapsing results in the graph $GNew. This construction allows the storage and combination of several graphs.

5 FUTURE WORK

Our future work plans include exploring other case studies to further investigate collapsing strategies that involve multi-collapses and their impact on generating and analyzing architectural views. We also want to investigate how the presence of multi-collapses affects the complexity of a container and the difficulty in understanding a container. We would like to determine if there are metrics such as the number of multi-collapses within a container that would give a guide to the complexity of a container. Also we want to investigate how the presence and number of multi-collapses relate to quality attributes, for example how the presence of multi-collapses affect the modifiability of a container or the system overall.

6 CONCLUSIONS

We have outlined strategies for collapsing information during the process of building abstractions during architecture reconstruction. We have identified situations in which collapsing will require the need to have multi-collapses. These multi-collapses can be very useful in understanding a system or particular aspects as they allow the information relevant to a container to be included within the container rather than having that information outside of the scope of the container.

Multi-collapses also reduce the clutter within the architectural views that are generated and assist the understanding of the system by allowing better hierarchical views of the system to be generated. The results from the case study show that the Satellite Tracking Agency was able to produce very useful views of the architecture of their system that allowed them to better understand and communicate about their system.

As a result of this work and the architectural views that STA are now able to generate, the developers at STA have a better understanding of their system and are now in a position to move forward with their work on porting the system, which is currently ongoing. They now have views of the STS that show the various components of the system and dependencies between those components.

REFERENCES

1. Bass, L.; Clements, P., and Kazman K., Software Architecture in Practice, 2nd ed. Reading MA: Addison Wesley, 2003.
2. Bengtsson, P. and Bosch, J., "Scenario-based Software Architecture Reengineering", Proceedings of the 5th International Conference on Software Reuse (ICSR5), IEEE, pp. 308-317, 2-5 June, 1998.
3. Bowman, T.; Holt, R. C.; & Brewster, N. V. "Linux as a Case Study: Its Extracted Software Architecture." 555-563. Proceedings of the 21st International Conference on Software Engineering. Los Angeles, CA, May 16-22, 1999. New York, NY: ACM Press, 1999.
4. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R. and Stafford, J., Documenting Software Architectures: Views and Beyond, Addison Wesley, 2002.
5. The DAGSTUHL Middle model project: http://scgwiki.iam.unibe.ch:8080/Exchange/2.
6. Feijs, L. M. G. and Krikhaar, R. L., Relation Algebra

with Multi-Relations, *International Journal of Computer Mathematics*, 70, pp 57-74, 1999.

7. P. J. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, The Portable Bookshelf, *IBM Systems Journal*, Vol. 36, No. 4, pp. 564-593, November 1997.

8. Kazman, R., O'Brien, L. and Verhoef, C, Architecture Reconstruction Guidelines 2nd Edition, CMU/SEI-2002-TR-034, Software Engineering Institute, Pittsburgh, USA.

9. Lakhotia, A., A Unified Framework for Expressing Software Subsystem Classification Techniques, *Journal of Systems and Software 36, 211-231*, 1997.

10. O'Brien, L., and Stoermer, C., Architecture Reconstruction Case Study, Technical Note, CMU/SEI-2003-TN-008, 2003.

11. Müller, H. A.; Mehmet, O. A.; Tilley, S. R.; & Uhl, J. S. "A Reverse Engineering Approach to System Identification." *Journal of Software Maintenance: Research and Practice 5*, 4 (December, 1993): 181-204.

12. Müller, H. A. and Uhl, J. S., "Composing Subsystem Structures Using (K,2)-Partite Graphs." *Proceedings of Conference on Software Maintenance (CSM 1990)*, (San Diego, California, November 26-29, 1990), pp. 12-19, IEEE Computer Society Press (Order Number 2091), November 1990.

13. Stoermer, C., O'Brien L., and Verhoef, C., Moving Towards Quality Attribute Driven Software Architecture Reconstruction. *Proceedings of the 10th Working Conference on Reverse Engineering* (WCRE), Victoria, Canada, November 2003.

14. Tahvildari, L., Kontogiannis, K. and Mylopoulos, J., *Quality-driven software re-engineering*, Journal of Systems and Software, vol. 6. Issue 3, June 2003.

15. Scientific Toolworks Inc., Understand for C++ and Fortran: http://www.scitools.com

16. Selby, R. W. and Basili, V. R., "Error Localization During Software Maintenance: Generating Hierarchical Descriptions from Source Code Alone." *Proceedings of Conference on Software Maintenance - 1988*, (Phoenix, AZ, October 24-27), pp.192-197, November 1988.

17. van Deursen, A. and Riva, C., Software Architetcure Reconstruction Tutorial at the International Conference on Software Maintenance, 3-6 October, 2002, Montreal, Canada.

18. Van Deursen, A, and Kuipers, T., Identifying Objects using Cluster and Concept Analysis. *In Proceedings of the ICSE, 246-255,* 1999.

19. Wall, L., Christiansen, T., & Orwant, J., Programming Perl, 3rd ed., O'Reilly.