Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002), IEEE Computer Society Press, 11-13 March 2002 in Budapest, Hungary.

Combining Static and Dynamic Views for Architecture Reconstruction

Claudio Riva and Jordi Vidal Rodriguez
Nokia Research Center
Software Architecture Group
P.O. Box 407, FIN-00045 NOKIA GROUP
Tel.: +358 50 483 7403, Fax.: +358 7180 36308
{claudio.riva | jordi.vidal-rodriguez}@nokia.com

Abstract

Static analysis aims at recovering the structure of a software system, while dynamic analysis focuses on its run time behaviour. We propose a technique for combining the analysis of static and dynamic architectural information to support the task of architecture reconstruction. The approach emphasises the correct choice of architecturally significant concepts for the reconstruction process and relies on abstraction techniques for their manipulation. The technique allows the software architect to create a set of architectural views valuable for the architecture description of the system.

To support our technique, we outline an environment that relies on hierarchical typed directed graphs to show the system's structure and message sequence charts for its behaviour. The main features of the environment are: visualisation of static and dynamic views, synchronisation of abstractions performed on the views, scripting support and management of the use cases. The approach and the environment are demonstrated with an example.

Keywords
Software Architecture, Reverse Engineering, MSC, Dynamic Analysis, Architecture Reconstruction.

1 INTRODUCTION

A well-accepted way to think of a software architecture is the "4+1 View" model proposed by P. Kruchten [9]. The model suggests organising the architecture descriptions in five different categories, called views: *logical view*, *process view*, *physical view* and *development view*. The fifth view, called *scenarios* or *use cases*, is used to illustrate and validate the four views. The "4+1 model" separates the description of the static and dynamic aspects of a software architecture. The logical view is primarily concerned with the functional requirements of the system. It should describe how the system is partitioned and what are the relationships among its parts. The process view is focused on the dynamic aspects of the architecture description and should describe the run-time behaviours of the system. Soni, Nord and Hofmesiter [2] also propose four different architectural views that are used to describe software architectures: *conceptual architecture*, *module interconnection architecture*, *execution architecture* and *code architecture*. The separation between static (conceptual, module and code architecture) and dynamic aspects (execution) is evident in this approach too. In our approach, we follow the definition of software architecture given in [13] and we consider the above-mentioned models as a practical way for describing the software architecture of a system.

In both the approaches, multiple views obtained from different perspectives are necessary for describing and understanding software architectures. During the forward engineering phase, the views are created by the minds of the architects who are establishing the fundaments of the software system implementation. These views are then analysed and reviewed to validate the system architecture. Once the development has started (an incremental and iterative process), the architectural views have to been maintained and synchronised with the actual implementation. Either for monitoring the development process or for validating the implementation, the architects need updated information about the system architecture. This poses the need for applying reverse engineering techniques aimed at extracting the architectural information from the actual implementation of the system.

In our previous work [17], we have developed a set of tools aimed at supporting specific reverse engineering techniques that we call *reverse architecting* or *architecture reconstruction*. These techniques and tools are mainly focused with the extraction of architectural information from the system implementation. A crucial problem that we are addressing is to combine the structural and dynamic information extracted from a software system. This paper describes our technique to solve this problem and the environment to support it.

2 RELATED WORK

The dynamic analysis aims at describing the run time behaviour of a software system. Its contribution should be considered during the reconstruction of a software architecture. Most of the methods and tools are intended to analyse only the static or the dynamic aspects of a system but not both at the same time. For example, SCED [7] or SCENE [8] are reverse engineering tools that focus on the dynamic analysis tasks.

Some attempts have been done to merge the dynamic information into static views. Systä [19] exploited Rigi [16] and SCED [7] to combine static and dynamic analysis. Her work was focused on Java source code level and program comprehension. She used the concept of horizontal and vertical abstractions to reduce the complexity and raise the abstraction level of the subject software. She also showed how the static information can guide the dynamic analysis and how to slice the static view using the dynamic data.

ISVis [5] is a tool for analysing C applications, which offers vertical abstractions and restricted horizontal abstractions. It only provides a MSC representation, and the horizontal abstractions are limited without support for static model visualisation.

Several tools use a kind of vertical abstraction to cope with message complexity. SCENE [8] uses limited size subscenarios and hyperlinks to source to aid program understanding. SCED [7] uses algorithmic constructs to express repetition like loops and subscenarios.

In the filed of tools for static and dynamic reverse engineering there is an attempt based on Dali [6] to support the static and dynamic analysis for object oriented software. Program explorer [21], supports source code, class hierarchy, invocation and object graphs. They are synchronised but the tool lacks of abstraction capabilities and is C++ domain dependent. Richner et al. [15] present a method to create views that combine static and dynamic information for object-oriented software. It is based on a logic programming language (Prolog) to query the data and on digraphs for the visualisation. R. Holt has also presented a method for manipulating software architecture data using the relational algebra of Tarski [3].

The major shortcomings of the current approaches are (1) architectural level is not the primary focus because most of the tools focus on source-level program understanding, (2) lack of automation in the extraction and analysis process, (3) lack of complete graphical visualisation of software models and (4) the tools are often coupled with a specific programming language and can be hardly used to handle generic architectural concepts like applications, servers and subsystems.

3 ARCHITECTURE RECONSTRUCTION

The description of a software architecture should communicate the essential decisions that have been taken in the design of a software system [13]. Architecture reconstruction (or reverse architecting) concerns with the task of recovering the past design decisions that have been taken during the development of a system. They comprise either decisions that have been lost (because not documented or developers have left) or are unknown (for examples, assumptions not initially take into account). The reconstruction is performed by examining the available artefacts (documentation, source code, experts) and by inferring new architectural information that is not immediately evident. Our approach can be summarised in the following four-step iterative process [17]:

1. Definition of architectural concepts

The goal is to recover and clarify the architecturally significant concepts that build the system. These concepts represent the way developers think of a system and they should become the terminology of the reconstruction process. They concern the building blocks of the system and the communication infrastructure that enables the components to communicate at runtime. In a distributed software system the architectural concepts may be applications, servers, software busses while in an operating system they may be tasks, processes, queues, shared memories, etc.

2. Data gathering

This phase gathers the relevant information for describing the software architecture of the system. We build a model of the system whose entities are instances of the concepts identified in the phase 1. A correct choice of the concepts will ensure that the model is filled with entities at the right level of abstraction. Source code is usually the most dependable for static analysis and simulation for dynamic analysis. However, documentation, software diagrams (for example, stored in CASE tools), experts can contribute to the creation of the model.

The static information is extracted by analysing the source code and searching for architectural significant elements. This can be achieved with ad hoc analysers (like presented in out previous work [17]) or with existing source code analysers (like SourceNavigator [14] and

Columbus [12]).

The dynamic information is obtained by instrumenting the source code and executing different scenarios in a simulator. We instrument the source code with ThirdEye [10] and select the scenarios among the features of the system. This choice aims at enabling the architects to understand how the features are implemented in a system.

3. Abstraction

The model of the previous phase is usually at a very low level of abstraction. The goal of this phase is to enrich the model with domain dependent abstractions that will lead to a high level view of the system. Known abstractions can be easily added to the model. Unknown abstractions have to be identified by the architects, categorised, named and then stored in the model. Such reasoning is conducted manually by the architects and then fixed in a set of abstraction rules. The abstraction process has also to produce the architectural views that will be presented in the last phase (for example, according to the 4+1 model [9]). The abstraction process is supported by a Prolog system. This allows us to formally specify the abstraction rules and the transformations that we need to apply to the model. The abstractions rules can be reused for later versions of the same system.

4. Presentation

The architects need to present the reconstructed architecture in different formats. We allow the architects to select a particular architectural view (logical, process, physical, development) and a particular format: graphs (using Rigi) for the static views and a simplified version of message sequence charts [4] for the dynamic views. We have also exploited UML visualisations with the tool Venice [20].

4 REQUIREMENTS FOR THE APPROACH

As we pointed out in Section 2, most of the approaches collect language dependent data either by parsing the source code or by tracing functions calls at run time. Such a low-level reverse engineering leads to a low-level comprehension of the subject software even after applying abstractions.

Our goal is to generate architecturally significant views of the system based on static and dynamic information. We want to allow the architects to start from a high-level view of the structure and behaviour of the subject software, and then to descend the levels of abstraction to focus on the details of the implementation or smaller parts of the system.

The high level view should not contain language programming artefacts as it is mainly intended to inform the architect about the important relationships of the system's high level components. We use directed graphs to represent the system's structure [17], and message sequence charts (MSC) [4] for describing its behaviour. The representations are based on generic elements and they allow us to describe software architectures for different domains. In the MSC we use concepts like participant and message, which can be matched for instance with subsystems at high level or function calls at source code level.

To cope with the complexity of the message sequence charts, we need two mechanisms of abstractions: *horizontal* and *vertical* abstractions. Horizontal abstractions allow us to group the participants of a MSC in order to reduce their number. As the participants are architectural entities in the static view, the horizontal grouping in the MSC has to be linked to the hierarchical grouping in the static view. Vertical abstractions consist with grouping a set of contiguous messages into one high-level message. This allows us to reduce the amount of messages and to order them according to the scenarios and sub-scenarios.

We follow an iterative process for refining the architectural model with the aid of the system knowledge. The initial raw model is a plain representation of the system without any hierarchical structure in the static view and many message repetitions in the dynamic view. To simplify the model, the user needs to switch between different views (static and dynamic) and maintain them synchronised. The synchronisation helps the user in identifying interesting relations in the structure (static view) and behaviour (dynamic view) of the system. As proposed by T. Systä [19], we group the participants of the dynamic views using the static information and we use the dynamic view to slice the static view to reduce its complexity.

The use cases are also a mechanism for compressing the dynamic view. We can break the trace into a set of hierarchical use cases, which provide a logical partition of the software behaviour by functionality. A *use case* is the ordered set of messages used to accomplish a software system or user level goal. A *scenario* is a contiguous sequence of messages that performs a concrete function but needs the context of a use case. A scenario can appear several times in the use case or others and contributes to build up and achieve the goal of the use cases. For instance, a user can 'load a file' (use case) in a tool, and the program executes many 'read record' calls (scenario), which uses some lower level calls.

We can summarise the requirements in the following points:

- Architecture and programming language independent approach.
- Generation of high-level views of the structure and

behaviour.

- Interactive manipulation and synchronisation of static and dynamic views.
- Abstraction widely used to cope with system complexity (message and participant compression, hierarchical grouping)
- Management of scenarios (and use cases).
- Support for an iterative process of data analysis

5 THE APPROACH

Our technique aims at creating architectural views based on static and dynamic information. We are seeking two key improvements for the analysis of dynamic information: (1) an effective method for managing vertical and horizontal abstractions and, (2) the integration of the static and dynamic tools in the same RE environment. A major constraint is on the independence of the RE environment from the subject system. We aim at decoupling the system domain dependent information from the generic concepts of the RE environment in order to easily customise the environment according to any subject system. Data gathering phase is system dependent.

We use the Rigi environment for the static visualisation of the system structure [17] with hierarchical typed graphs. Graphs can be customised according to the subject system with the Rigi's concept of *domain*. Rigi is an open environment that can be easily extended using the Tcl/Tk language. We have extended Rigi to support the visualisation of dynamic data. Figure 1 shows the integration scheme. The MSC visualisation tool is able to exchange data with Rigi, reacting when static abstractions are performed and vice versa. The synchronisation of both views is essential because the user need to work on the multiple views at the same time. The data from the static and dynamic analysis are represented with Prolog facts. A set of Prolog propositions is used to process the data and generated the required format for Rigi and the MSC visualisation tool.

The main features of the environment are: (1) the ability to perform horizontal and vertical abstraction on the dynamic view, (2) the integration with a static analysis tool and, (3) the support for the management of hierarchical use cases, (4) a scripting facility based on Tcl/Tk for customisation and batch processing.

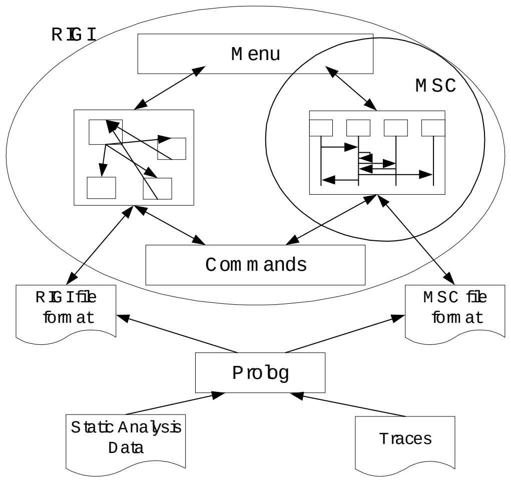


Figure 1 Integration scheme of the MSC tool in Rigi.

Figure 2 shows the different types of abstractions supported on the dynamic view. Diagram a) shows the initial MSC, diagram b) shows an horizontal abstraction over participants A and B, diagram c) shows a vertical abstraction over messages M1 to M4, and diagram d) shows the effect of both types of abstraction, resulting in a simplified and high-level MSC.

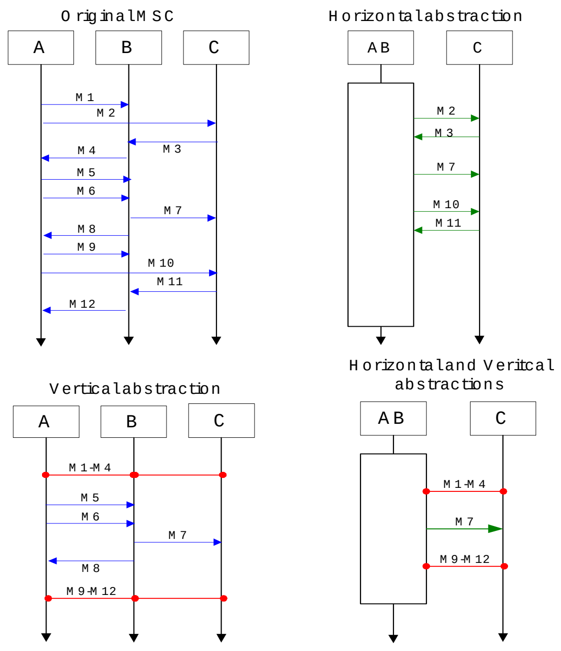


Figure 2. Horizontal and Vertical abstractions in MSC.

In the MSC file format, we have implemented a support for the management of use cases. We need this feature to cope with the complexity and size of traces. The use cases provide a logical hierarchy for partitioning the long traces that we extract. Use cases can be used to describe HMSC and the scenarios bMSC [4]. During the data-gathering phase, we organise the use cases in a tree hierarchy (nested use cases are possible) as shown in Figure 3. The hierarchy is defined during the data gathering phase, inserting bookmarks in the trace. The use cases hierarchy can be used to slice the dynamic views and provides context information of the current scenario being analysed. The abstractions applied on one scenario are also reflected overamong the other scenarios in the hierarchy.

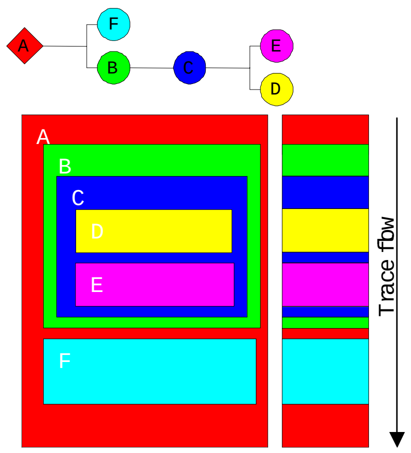


Figure 3. Three views of the Use Case hierarchy.

6 AN EXAMPLE

We demonstrate the approach with an example that is the simplification of a real case. The description follows the fours steps of the process described in Section 3.

Architectural Concepts

We start examining the architectural documentation in order to identify the architecturally significant concepts that build the system. The system is component based. Components represent computational units or system resource controllers and expose well-defined interfaces. The communication among the components is achieved with the exchange of asynchronous messages. The operating system takes care of the delivery of the messages using a software bus. There are three primitives for registering on the bus, sending and receiving messages:

register(ID) – primitive to register a component on the bus.

send(dest, msg) – primitive for sending a message to dest.

recv(src, msg) – primitive for retrieving a message from the component's queue.

When the components are initialised, they register themselves on the bus with a unique identifier that is statically assigned at compile time. The same identifier is used in the sending and receiving primitives.

One key issue of the architects is to manage the organisation of the components so that they can collaborate to implement the system's features. Each message exchange between two components creates a dependency that has to be taken into account by the architect. In a system with hundreds of components the dependency graph becomes rather complicated. For this task, the architects can find very useful to have a *logical view* that shows the organisation of the components in packages and their dependences and an *execution view* that shows how the components interact. To generate those views, we have to analyse the exchange of messages statically and dynamically. This is the focus of the next phases.

Extraction of Static Information

The extraction of the static information is carried out with a source code analyser that detects all the communication instances in the source code. The approach is similar to the one described in our previous work [17].

The output of the extraction is presented as a set of Prolog facts. Each fact defines a relationship between two elements of the model. In our case, we have created (1) the 'register' relationship between a component's directory and its name on the software bus and (2) the 'message' relationship among the sender, receiver and content of a message. Below there is a sample of the information extracted by the code analyser.

```
register('/gui/voice','VOICE_CTRL').
message('VOICE_CTRL','PROTOCOL','sel_prot').
message('VOICE_CTRL','PROTOCOL','connect').
register('/gui/data','DATA_CTRL').
message('DATA_CTRL','PROTOCOL','sel_prot').
register('/resource/protocol','PROTOCOL').
message('PROTOCOL','RADIO','setup').
register('/gui/makeCall','MAKE_CALL').
message('MAKE_CALL','DISPLAY','write').
```

For instance, the first line means that the directory '/gui/voice' contains the implementation of the component 'VOICE_CTRL'. The second line means that the component 'VOICE_CTRL' sends a message to the component 'PROTOCL' and the message is 'sel_prot'.

Abstraction

The extracted source code model is at a very low level of abstraction. The abstraction process can be divided in two steps: *composition rules* and *view selection*.

Composition Rules

This step concerns with adding the part-of relationships to the model. It is a crucial activity of the abstraction process because it creates a hierarchical structure in the model that simplifies its navigation. The composition rules specify how the source code elements are grouped to form subsystems or more abstracted entities.

Below there is an example in Prolog where we define four new components. The components are associated to the directory that contains its implementation.

```
contain('Call',X) :- register('/gui/voice',X).
contain('Call',X) :- register('/gui/data',X).
contain('Messaging',X) :- register('/gui/inst_msg',X).
contain('Messaging',X) :- register('/gui/email',X).
contain('Network',X) :- register('/resource/protocol',X).
```

The previous components are then grouped in subsystems according to their functionality.

```
contain('Services','Call').
contain('Services','Call').
contain('GUI','Services').
contain('GUI','Application').
contain('Resources','Display').
contain('Resources','Network').
```

View selection

To define a view, we need (1) to select its representation format and (2) to define the set of relationships that have to be projected in it. We use graph-based representations for the static information and message sequence charts for the dynamic information

To create the static view we need to compute the high level dependencies among the components. We represent the logical view with the typed oriented hierarchical graphs of Rigi. This is achieved by (1) defining a *grouping* relationships that describes the hierarchy, (2) defining the set of relationships of the graph, (3) compuing the transitive closure and (4) create the graph.

Below there is the Prolog code that defines the *grouping* relationship and the *relation* relationship for our simple example.

```
grouping(X,Y) :- contain(X,Y).
relation(X,Y) :- message(X,Y).
```

We can calculate the reflexive transitive closure with an auxiliary function *trans* defined below:

```
trans(R,X,Y) :- P=..[R,X,Y], call(P).
trans(R,X,Y) :- P=..[R,X,Link],call(P), trans(R,Link,Y).
trans(_,X,X).
```

We can then create the graph by defining a *hierarchy* relationship that is basically identical to the *grouping* relationship. The edges of the graph are obtained by the transitive closure. Below there is the Prolog code.

```
hierachy(X,Y) :- grouping(X,Y).
edge(X,Y):-tran(grouping,X,Item1),tran(grouping,Y,Item2),
        relation(Item1,Item2).
```

In this simple example, we have not taken in consideration the management of the types of the nodes that is done in the real case.

Visualisation.
The logical view can be visualised with Rigi as a hierarchical graph. We can also export the graph from Rigi to the Venice. Venice allows us to show the logical view in the UML notation. We use the notation proposed in [18]. Figure 4 shows the logical view of the example. The packages have been created according to the *hierarchy* relationship. The edges show the high level dependencies that exist among the packages. The user can select the level of detail for the visualisation of the edges [20].

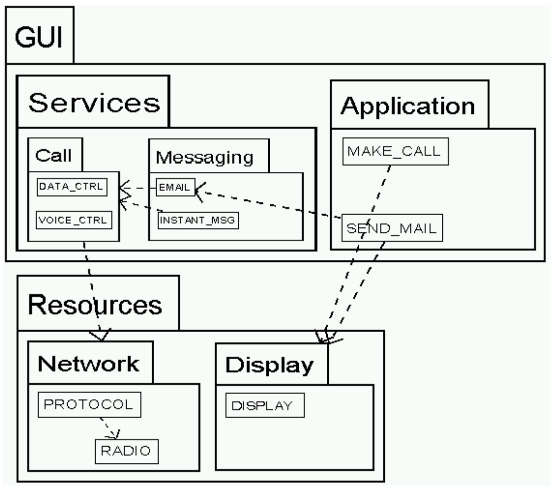


Figure 4. The component view visualised with Venice.

Extract dynamic information.
The extraction of dynamic information is conducted by (1) instrumenting the source code, (2) executing a set of usage scenarios and (3) collecting the traces. The architectural concepts of the first phase drive us in the choice of the correct instrumentation of the system. In the example, we can trace all the calls to the 'send' primitive that are architecturally significant for our system.

We use the ThirdEye [10] environment for instrumenting the code. In the example, we execute two scenarios in the system simulator. Each scenario is represented by a sequence of commands. The traces are then converted to Prolog facts. And example is given below.

```
trace('sc_start','MakeCall','','',1).
trace('msg','MAKE_CALL','DISPLAY','sel_num',2).
trace('msg','MAKE_CALL','VOICE_CTRL','setup',3).
trace('msg','VOICE_CTRL','PROTOCOL','sel_prot',4).
...
trace('msg','VOICE_CTRL','MAKE_CALL','callact',14).
trace('sc_end','MakeCall','','',15).
trace('sc_start','SendMail','','',16).
trace('msg','SEND_MAIL','EMAIL','setup',17).
...
trace('msg','PROTOCOL','DATA_CTRL','ack',29).
trace('msg','DATA_CTRL','EMAIL','con_active',30).
trace('sc_end','SendMail','','',31).
```

The ‘trace’ relationship contains the type of the trace, the sender, the receiver, a label and the time stamp. Two special events (‘sc_start’ and ‘sc_end’) delimit the begin and end of a scenario. We elaborate the traces with Prolog propositions in order to apply the abstraction rules created during the static analysis. We plan to use Prolog also to separate the different scenarios and to detect patterns in the sequence chart. The results can be exported to the Message Sequence Chart visualiser.

Navigating Static and Dynamic Views

We present an example of navigation in the architectural model with both the static and dynamic views. Figure 5 shows a graph-based representation of the logical view of Figure 4 in Rigi. The graph shows the two components at the highest level of abstraction: ‘GUI’ and ‘Resources’. The process (or execution) view is visualised in the MSC Visualiser as shown in Figure 6. The MSC is shown at the same level of abstraction as the logical view in Rigi. Therefore, only the two high level participants and messages are visualised. The messages that are exchanged among the nodes contained in the two participants are hidden. This reduces the objects to be visualised and simplifies the understanding of the MSC.

The user can navigate the two views by expanding or collapsing the nodes in the same style like in Rigi. The system takes care of maintaining the two views synchronised. We imagine the user wants to investigate the details of the ‘GUI’ component. The user can expand the GUI node for example in the logical view, as shown in Figure 7. The same effect is propagated to the MSC, as shown in Figure 8. The messages between the two components (‘Services’ and ‘Application’) of the ‘GUI’ component are visible. The same result can be achieved by selecting and expanding a node in the MSC.

The user can also apply vertical abstractions in order to hide a set of messages. In Figure 9, the user selects a set of messages. The selection can be either contiguous or not. The user can then group together the messages in one single message that represents them. In case of un-contiguous messages, multiple groups are created. The result of the grouping is shown in Figure 10. Figure 11 and Figure 12 show the views at a lower level of abstraction where the two low level elements (‘MAKE_CALL’ and ‘SEND_MAIL’) are visible.

The user can also follow a bottom-up approach by starting with a raw model and by grouping the nodes towards higher levels of abstraction. The system takes care of keeping the views synchronised.

7 CONCLUSIONS AND FUTURE WORK

Understanding a software architecture requires both static and dynamic information to be available and presented in different views. Abstraction plays a key role in reducing the complexity of the models in different dimensions (system structure, organisation of scenarios, participants and messages in the MSC).

We have highlighted that the architectural views have to be maintained synchronised in order to present the static and dynamic views at the same level of abstraction.

The end-user customisation represents a powerful mechanism to tailor the tools for the specific subject system needs. It is an improvement to cope with the wide variety of programming languages. Moreover, it makes easy to introduce new features, to aid the process of abstracting and understanding, like dynamic metrics calculation and participants clustering algorithms [1].

The use case hierarchy allows the user to select the starting point of the analysis, either from a bottom-up or a top-down approach.

The use case hierarchy allows the user to focus into more concrete use cases or scenarios. The hierarchy could be used to generate other static-dynamic representations, as in [11], or collaboration diagrams, at different levels of architecture abstraction. As well, it is useful for program comprehension, to isolate smaller scenarios, discover errors or analyse concrete behaviour.

Our future work will focus on creating a better management for the scenarios and defining a technique for detecting interaction patterns in the MSC using Prolog.

8 REFERENCES

1. Cho E. S., Kim C. J., Kim S. D., Rhew S. Y., Static and Dynamic Metrics for Effective Object Clustering, *Proceedings of the Asia Pacific Software Engineering Conference*, December 2-4 1998, Taipei, Taiwan.

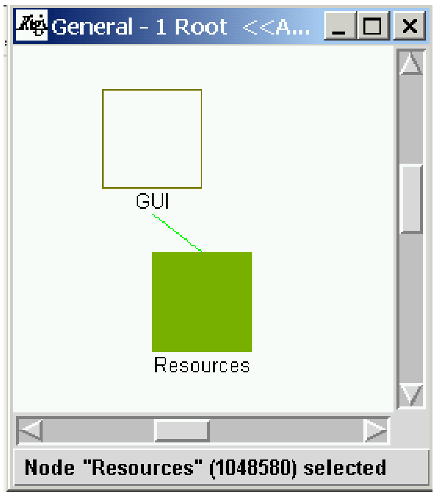
2. Hofmeister C., Nord R.L. and Soni D., Describing Software Architecture with UML, Proc. of the 1st Working IFIP Conference on Software Architecture, Kluwer Academic Publishers, 1999.

3. Holt R., Structural Manipulations of Software Architecture using Tarski Relational Algebra, Proceedings of the Working Conference on Reverse Engineering 1998, pp. 210-219, 1998.

4. ITU-T. recommendations Z.120. ITU – Telecommunication Standarization Sector, Geneva, Switzerland, may 1996. review draft Version.

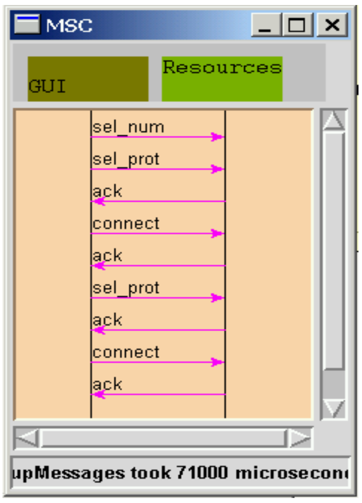
5. Jerding D., Rugaber S., Using visualization for Architectural Localization and Extraction, *Proceedings of the Working Conference on Reverse Engineering, (WCRE97)*, pp. 56-65, October 1997, Amsterdam, Netherlands..

6. Kazman R., Carrière S. J., View Extraction and View Fusion in Architectural Understanding, *Proceedings of the fifth International Conference on Software Reuse (ICSR5)*, pp.290-299, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.
7. Koskimies K., Männistö T., Systä T., Tuomi J.: Automated support for modeling of OO software. IEEE Software, January/February 1998, 87-94.
8. Koskimies K., Mössenböck H., SCENE: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs, *Proceedings of the 18th International Conference on Software Engineering, (ICSE '96)*, IEEE Computer Society Press, pp 366-375, 1996.
9. Kruchten P.B., The 4+1 View Model of architecture, IEEE Software, 12(6):42-50, 1995.
10. Lencevicius R., Ran A., Yairi R., ThirdEye – Specification-Based Analysis of Software Execution Traces, Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, 4-11 June, 2000, page 772.
11. Leue,S., Mehrmann L., Rezai M., Synthesizing software architecture descriptions from Message Sequence Chart specifications, Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98), 1998, Page(s): 192 - 195
12. R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss and M. Tarkiainen. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems, Proc. of SPLST 2001, Szeged, Hungary, pp. 16-27, June 15-16, 2001. (ed. T. Gyimothy)
13. Ran A., "ARES Conceptual Framework for Software Architecture" in M. Jazayeri, A. Ran, F. van der Linden (eds.), Software Architecture for Product Families Principles and Practice, Addison Wesley, 2000.
14. RedHat Source Navigator, http://sources.redhat.com/sourcenav/
15. Richner T., Ducasse S., Recovering high-level views of object-oriented applications from static and dynamic information, Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99), Oxford, 1999, Page(s): 13 -22
16. Rigi: a visual tool for understanding legacy systems, University of Victoria, http://www.rigi.csc.uvic.ca/
17. Riva C., Reverse Architecting: an Industrial Experience Report, Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000), Brisbane, Australia, 23-25 November, 2000.
18. Riva C., Xu J. and Maccari A., Architecting and Reverse Architecting in UML, Workshop on Describing Software Architecture with UML, International Conference on Software Engineering 2001 (ICSE), Toronto, May 2001.
19. Systä T., Static and Dynamic Reverse Engineering Techniques for Java Software Systems, Ph.D Thesis, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-4, 2000.
20. Venice, http://www.cs.Helsinki.FI/group/venice/
21. Danny B. Lange and Yuichi Nakamura. Program Explorer: A Program Visualiser for C++. *In Proc. of the USENIX Conference on Object-Oriented Technologies*, June 1995.



Node "Resources" (1048580) selected

Figure 5. The component view in Rigi.



upMessages took 71000 microsecon

Figure 6. The MSC in the MSC visualisarer.

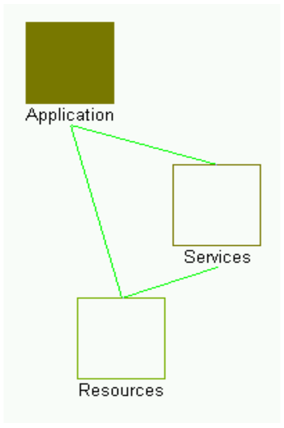


Figure 7. Expand node.

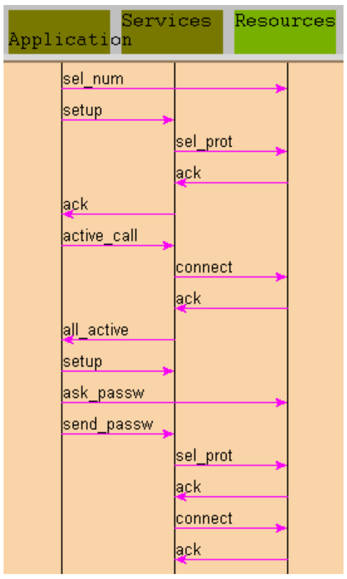


Figure 8. Expand node.

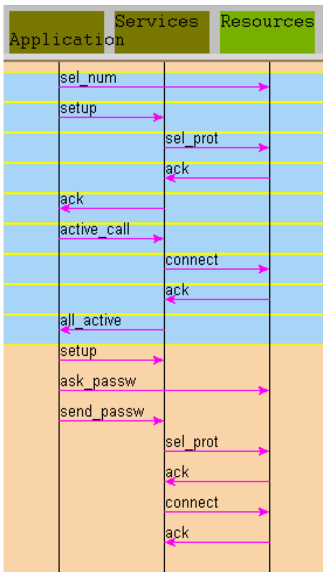


Figure 9. Selection in MSC.

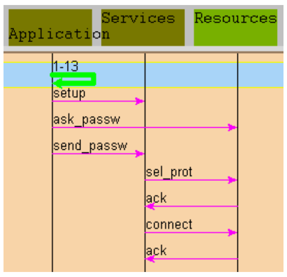


Figure 10. Grouping.

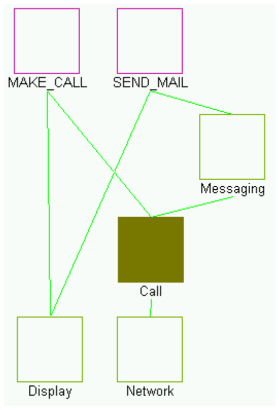


Figure 11. Low level.

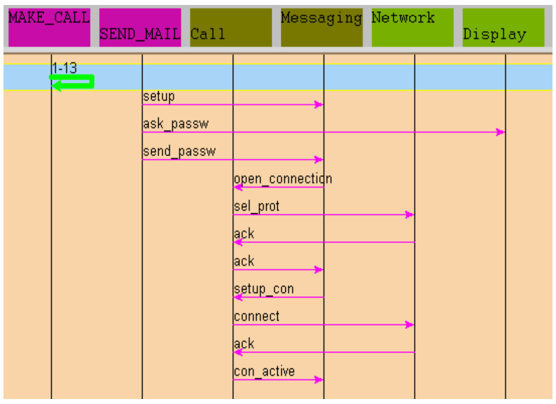


Figure 12. Low level details in MSC.