Using Graph Patterns to Extract Scenarios

Jingwei Wu, Ahmed E. Hassan and Richard C. Holt
Software Architecture Group
Department of Computer Science
University of Waterloo
Waterloo, ON, Canada
{j25wu, aeehassa, holt}@plg.uwaterloo.ca

Abstract

Scenario diagrams are useful for helping software developers to understand the interactions among the components of a software system. In this paper, we present a semi-automatic approach to extracting scenarios from the implementation of a software system.

In our approach, the source code of a software system is represented as a graph and scenarios are specified as graph patterns. A relational calculator, Grok, is extended to support graph pattern matching. Grok, as extended, is used in our analysis of the Nautilus open source file manager. Multiple scenarios are extracted and analyzed. These scenarios have helped us to understand Nautilus's architecture.

1 Introduction

In reverse engineering, it is common to extract software architectures from source code and system documentation [1, 2]. An extracted architecture is often represented in box-and-arrow diagrams. These diagrams help us understand the system's architecture, but do not express the order of component interactions in the execution of the system. Software engineers use scenarios to illustrate sequences of interactions among the components within a software system [3, 20].

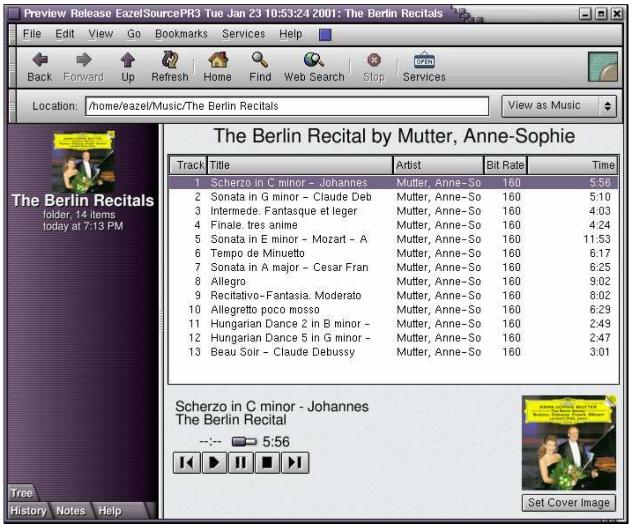
Scenarios provide a good mechanism for understanding software systems and validating software architectures [3]. Many benefits and uses of scenarios have been discussed. For example, in the 4+1 view model proposed by Kruchten, the scenario view consists of a small set of critical use cases, which illustrate how the other four views (logical, process, development and physical) are interrelated [4]. To gain a better understanding of a software system, we need to extract a useful set of scenarios. The major resources we can use to do the extraction include the requirement specifications, the system's implementation, and the system's runtime behavior.

Scenario extraction is a challenging problem. In many cases, the software system must be instrumented and must run in a profiler which traces function calls and data accesses. The system's source code or execution environment need to be modified to extract scenarios. Such modifications may not be possible or may be too time consuming to implement. In this paper, we present an approach that requires no changes to the source code of a software system or its execution environment. This approach is based on the static analysis of source code. We use graph patterns to extract scenarios from the information gathered from source code. The scenarios that we consider describe sequences of interactions among the components in a software architecture.

We demonstrate our approach through a case study on an open source file manager, Nautilus [6], which is developed as an integral part of the GNOME [7] desktop environment. Nautilus takes the Windows Explorer metaphor, but it has more elegant features. Figure 1 shows a music view embedded in the Nautilus window. We consider Nautilus a good candidate for a case study for the following reasons:

1. Its architecture has little documenation;
2. It is a moderately large system (290 KLOC);
3. It is built on the GNOME component technology and the CORBA technology, and it has an interesting mechanism of communicating with componentized views (e.g., Music view and Photo Album view).

The rest of this paper is organized as follows. Section 2 gives an overview of the process we used to do the analysis of Nautilus. Section 3 describes an extension to Grok [9] for graph pattern matching. Grok is a tool we used to do scenario extraction. Section 4 describes how graph patterns were used to extract scenarios for a software system. Section 5 discusses why we used graph patterns and also presents lessons learned in our work. Section 6 considers the related work. Section 7 concludes the paper.



Preview Release EazelSourcePR3 Tue Jan 23 10:53:24 2001: The Berlin Recitals

Location: /home/eazel/Music/The Berlin Recitals

The Berlin Recitals
folder, 14 items
today at 7:13 PM

The Berlin Recital by Mutter, Anne-Sophie

Track	Title	Artist	Bit Rate	Time
1	Scherzo in C minor – Johannes	Mutter, Anne–So	160	5:56
2	Sonata in G minor – Claude Deb	Mutter, Anne–So	160	5:10
3	Intermede. Fantasque et leger	Mutter, Anne–So	160	4:03
4	Finale. tres anime	Mutter, Anne–So	160	4:24
5	Sonata in E minor – Mozart – A	Mutter, Anne–So	160	11:53
6	Tempo de Minuetto	Mutter, Anne–So	160	6:17
7	Sonata in A major – Cesar Fran	Mutter, Anne–So	160	6:25
8	Allegro	Mutter, Anne–So	160	9:02
9	Recitativo–Fantasia. Moderato	Mutter, Anne–So	160	8:02
10	Allegretto poco mosso	Mutter, Anne–So	160	6:29
11	Hungarian Dance 2 in B minor –	Mutter, Anne–So	160	2:49
12	Hungarian Dance 5 in G minor –	Mutter, Anne–So	160	2:47
13	Beau Soir – Claude Debussy	Mutter, Anne–So	160	3:01

Scherzo in C minor - Johannes
The Berlin Recital

--:-- 5:56

Figure 1. The Nautilus application

2 Overview of Analysis

This section explains how our analysis of the Nautilus system was carried out in two major stages: architecture extraction and scenario extraction (see Figure 2). In the second stage, scenario extraction, which is central to this paper, we relied on the knowledge gained during architecture extraction. We used the Portable Bookshelf (PBS) toolkit [10] to automate much of our analysis.

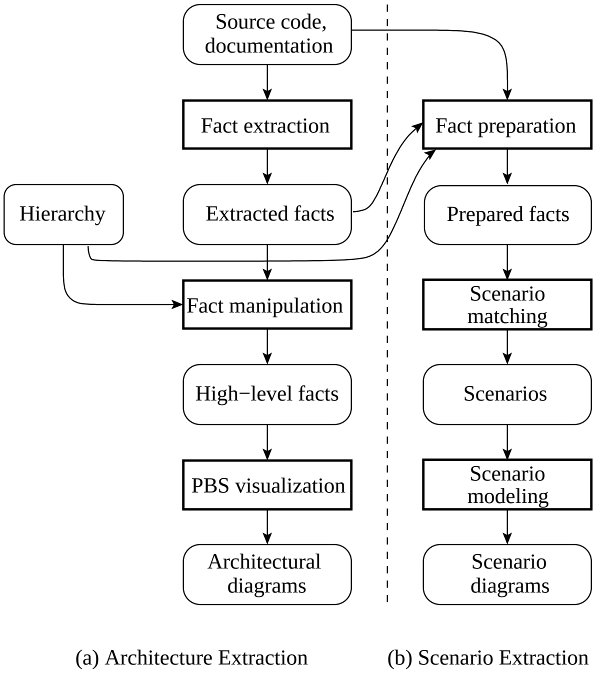
2.1 Architecture Extraction

In the first stage, architecture extraction, we followed these steps, which are supported by the PBS toolkit:

(1) **Fact Extraction**. We used a C source code extractor called *cfx* [10] to extract facts from Nautilus, such as "file `x.c` defines function `f1`" and "function `f1` calls function `f2`". The extracted facts were stored in a plain text file, which is referred to as a *factbase*.

(2) **Fact Manipulation**. We used a fact manipulator called *Grok* [9, 10] to manipulate the extracted facts and the containment facts, stored in a *hierarchy* file, to produce high-level facts, such as "subsystem `P` depends on subsystem `Q`". The hierarchy is a tree-structured decomposition of Nautilus. As we understood more about the organization of Nautilus, we modified the hierarchy file to reflect such an understanding. So, the hierarchy file is the starting point of iterations during architecture extraction.

(3) **PBS Visualization**. We used the PBS *layouter* to determine the coordinates for each box so that the PBS *visualizer* can display them on the screen [10]. The *visualizer* supports manual layout manipulation, for example, moving and resizing.



(a) Architecture Extraction

(b) Scenario Extraction

Figure 2. Overview of analysis

Based on these analysis steps, we derived an architectural view of Nautilus, whose highest level of abstraction is shown in Figure 3. This level has five subsystems: (1) the *Window* subsystem is responsible for normal file management and the strategy of linking and embedding views dynamically; (2) the *Initializer* subsystem does the loading and initialization of views at the request of the window; (3) the *Proxy* subsystem maintains references to various view components, forwarding the service request of the window to views and reporting the status of views back to the window; (4) the *Views* subsystem contains various view components such as *Image Viewer*, *Music View*, and *Mozilla* (a wrapper application of the Mozilla web browser); and (5) the *Library* subsystem contains the routines that are used to develop views linkable to Nautilus and the routines that are only used internally by Nautilus. The first four subsystems are grouped in a dashed box, which has an arrow to the Library subsystem. This indicates that these four subsystems all depend on the Library subsystem.

The architectural diagram in Figure 3 shows the modular organization of Nautilus, but it does not reveal the system's dependency on the underlying CORBA technology. To understand the dependency of Nautilus on CORBA, we extracted a set of scenarios involving CORBA.

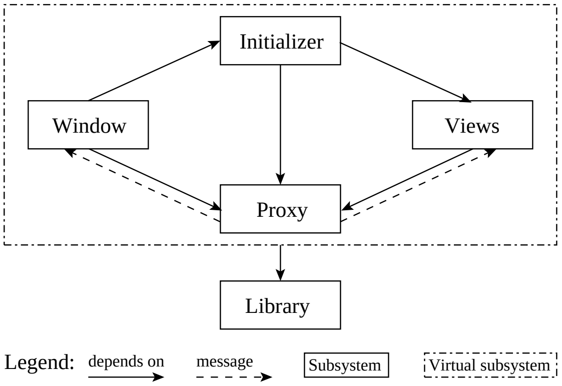


Figure 3. Software architecture of Nautilus

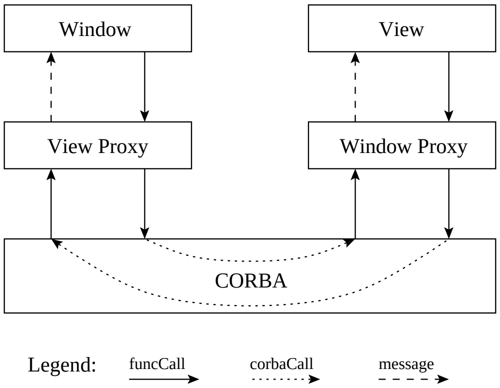


Figure 4. Layered Communication in Nautilus

2.2 Scenario Extraction

This section will explain our process, illustrated in Figure 2(b), for extracting scenarios. Before discussing each analysis step, we first describe layered communication in Nautilus.

The layered structure in Figure 4 shares the same style as the Open System Interconnection (OSI) reference model for network communications [8], but it has less layers. The top layer is the *Window-View* layer, containing the Nautilus window application and various view components. The middle layer is the *Proxy* layer, containing the view proxy and the window proxy. The proxy layer relays routine calls from the Window-View layer to the CORBA layer and messages from the CORBA layer to the Window-View layer. The bottom layer is CORBA, which does the low-level communication between the proxies. In fact, we conjectured this layered communication architecture based on our understanding of Nautilus from the first stage. We needed a method to validate it.

The directions of the two paths in Figure 4 imply two communication scenarios:

> **Window to view scenario:** The window invokes function calls on the view proxy. Then, the view proxy translates these calls to CORBA calls to the window proxy which sends a set of messages to the actual view component.
>
> **View to window scenario:** A view component invokes function calls on the window proxy. Then, the window proxy translates these calls to CORBA calls to the view proxy which sends a set of messages to the window.

The objective of our analysis was to extract these two scenarios and to use them as a means to validate the layered communication architecture. Also, we hoped to produce a set of scenario diagrams as enhanced versions of the architectural diagrams of Nautilus (e.g., Figure 3 and Figure 4). During the extraction, we followed three steps:

(1) **Fact Preparation**. We used scripts to extract message and `corbaCall` facts (see Figure 4), which were not produced automatically by our fact extractor (cfx). These new facts, with the facts extracted by cfx and the facts stored in the containment hierarchy, were prepared for scenario extraction and merged into a new factbase.

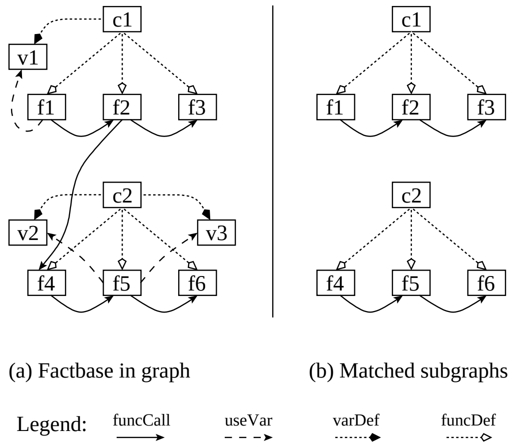
(2) **Scenario Matching**. We applied graph patterns to describe those two communication scenarios. Then, we used Grok, which was extended to support graph pattern matching, to extract these scenarios from the prepared factbase.

(3) **Scenario Modeling**. We produced a set of diagrams by applying UML interaction diagrams [5] to model the scenarios extracted at the previous step. We chose collaboration diagrams instead of sequence diagrams as the modeling method. The reason for our choice will be explained in Section 5.

Before discussing these three steps in detail, we explain how Grok is used for pattern matching in the second step of scenario extraction.

3 Enhancing Grok with Graph Patterns

Grok is a relational calculator that supports a scripting language [10]. In the PBS toolkit, Grok is used to manipulate code-level facts to produce architectural abstractions and aggregations by way of algebraic calculations [9]. We have extended Grok to have graph patterns to match structures (graphs) in factbases.



(a) Factbase in graph

(b) Matched subgraphs

Figure 5. Matching graph patterns within a graph, which represents a factbase.

3.1 Example Factbase

We will start with an example factbase to illustrate how we store facts about programs. In this factbase, facts are represented as ASCII triples in Rigi Standard Form (RSF) [12]. For example, in the following RSF triples, `funcDef c1 f1` means that class `c1` defines function `f1`, and `funcCall f1 f2` means that function `f1` calls function `f2`. The following factbase can be considered the typed graph shown in Figure 5(a).

```
varDef     c1   v1
varDef     c2   v2
varDef     c2   v3
funcDef    c1   f1
funcDef    c1   f2
funcDef    c1   f3
funcDef    c2   f4
funcDef    c2   f5
funcDef    c2   f6
useVar     f1   v1
useVar     f5   v2
useVar     f5   v3
funcCall   f1   f2
funcCall   f2   f3
funcCall   f4   f5
funcCall   f5   f6
funcCall   f2   f4
```

3.2 Graph Patterns

Given a factbase extracted from a source program, we can define a graph pattern to model a particular structure occurring in the factbase, for example, a path, a cycle, or, more generally, a subgraph. In general, the match of a graph pattern is a set of subgraphs, each representing an occurrence of that pattern.

In Grok, a graph pattern is defined by a set of pattern variables and a set of constraints (rules) to be applied to those pattern variables. We can think of pattern variable as abstract graph nodes, and pattern rule as abstract graph edges or abstract graph attributes. Thus, a graph pattern is essentially an abstract graph. Grok requires that a properly defined graph pattern must be a connected graph.

We now explain how to match an example structure described as follows:

> Within a class C, function F calls function G, and function G calls function H.

This structure can be modeled as a graph pattern using the following Grok script:

```
pattern1 := {funcDef  C  F;
             funcDef  C  G;
             funcDef  C  H;
             funcCall F  G;
             funcCall G  H}
```

The definition of `pattern1` is composed of five pattern rules, which are expressed as RSF triples. When being used to define a graph pattern, each RSF pattern rule implicitly declares two pattern variables. For example, the rule `funcCall F G` declares two pattern variables, `F` and `G`, which must have a relation `funcCall` between them. When `pattern1` is applied to the graph in Figure 5(a), it matches the two subgraphs in Figure 5(b). The match of `pattern1` will be stored in a n-ary relation where the column names are actually pattern variables (see Table 1). Each row in the table corresponds to a matched subgraph in Figure 5(b).

C	F	G	H
c1	f1	f2	f3
c2	f4	f5	f6

Table 1. Match of Pattern1

We can reuse a pattern match by expanding it with new pattern rules. For example, `pattern2` matches subgraphs from Table 1, in which function `G` uses a variable `V` declared in the same class as `G`.

```
Pattern2 := {Pattern1:
             useVar  G  V;
             varDef  C  V}
```

The match is shown in Table 2.

C	F	G	H	V
c2	f4	f5	f6	v2
c2	f4	f5	f6	v3

Table 2. Match of Pattern2

We can apply constraints to select a subset of the matched instances of a pattern. For example, we may want to select those pattern instances, in which the name of function G matches the regular expression "f.*" and the name of variable V must be "v2". The match is shown in Table 3.

```
Pattern3 := {Pattern2:
             (G like "f.*") and
             (V equal "v2")}
```

C	F	G	H	V
c2	f4	f5	f6	v2

Table 3. Match of Pattern3

Since scenarios capture the interactions among a set of components, we can use graph patterns to model such interactions and to extract the scenarios as subgraphs from a factbase extracted from a source program.

4 Scenario Extraction

In this section, we give a detailed description of the three steps of scenario extraction (fact preparation, scenario matching, and scenario modeling). We will show our experience of using graph patterns to extract scenarios.

4.1 Fact Preparation

In Section 2.2, we identified two communication scenarios: one goes from the window to the view, and the other takes the reverse direction (see Figure 4). In either direction, the communication passes through the two proxies. As can be seen in Figure 4, three relations are involved: `funcCall`, `corbaCall`, and `message`. Our fact extractor (*cfx*) produced `funcCall` facts automatically, but it does not generate facts for `corbaCall` and `message`. We had to prepare these last two before starting scenario extraction.

To prepare `message` facts, we studied the implementation of the view and window proxies. We found that both proxies use the signal routine `gtk_signal_emit` provided by the GTK+ Signal [11] to send a message. Figure 6 illustrates the role GTK_Signal plays in the window-to-view communication. We used *grep* to locate the occurrences of `gtk_signal_emit` from the Nautilus source code and created `message` facts by scripting. For example, a message fact may look like `message gtk_signal_emit music_view_load_location_callback`. This fact means that the GTK_Signal signals the music view to load a location (e.g., a file directory of songs).

To prepare `corbaCall` facts, we relied on the CORBA naming convention specified by GNOME. The relation `corbaCall` represents the mapping from CORBA stub routines to their corresponding skeleton routines. According to the CORBA naming, if we prefix a stub routine (e.g., `report`) with `_ORBIT_skel_`, we will get its skeleton routine (e.g., `_ORBIT_skel_report`). Correspondingly, these two routines together form a `corbaCall` fact, `corbaCall report _ORBIT_skel_report`. Using Grok scripts, we prepared all the `corbaCall` facts.

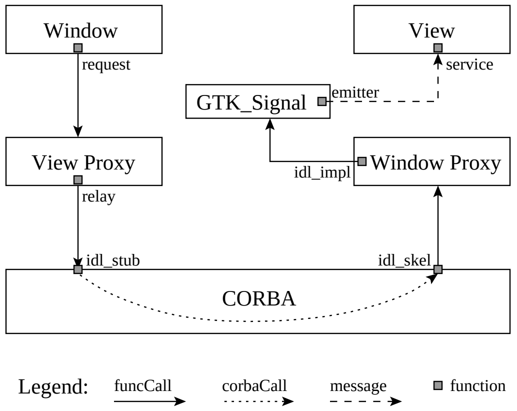


Figure 6. Window-to-view communication

After preparing the `message` and `corbaCall` facts, we merged them with the facts extracted by cfx and the facts stored in the containment hierarchy into a new factbase. We also simplified this factbase by abstracting many minor details inside subsystems.

4.2 Scenario Matching

Given the prepared factbase from Nautilus, we were ready to apply graph patterns to extract the communication scenarios for Nautilus. Along the path from the window to the view is a chain of six relations: `funcCall`, `funcCall`, `corbaCall`, `funcCall`, `funcCall`, and `message` (see Figure 6). A similar chain exists for the view-to-window scenario. To define a pattern for this chain of six relations, we used seven pattern variables: `request`, `relay`, `idl_stub`, `idl_skel`, `idl_impl`, `emitter`, and `service`, as illustrated in Figure 6. In this case, each pattern variable represents a function. Using Grok, we specified this chain as the `scenarioChain` pattern.

```
% Scenario chain in either direction
scenarioChain := {
  funcCall  request relay;
  funcCall  relay idl_stub;
  corbaCall idl_stub idl_skel;
  funcCall  idl_skel idl_impl;
  funcCall  idl_impl emitter;
  message   emitter service
}
```

The match of scenarioChain contains both classes of communication scenarios, either window-to-view or view-to-window. To select the scenarios in each direction, we used a small set of CORBA stub routines as the selection standard. For the window-to-view scenarios, the view proxy has access to five stub routines. For the view-to-window scenarios, the window proxy has access to ten stub routines, but only four of them are shown in the following scripts. Also, the signal emitter must be gtk_signal_emit.

```
% The set of stubs used by view proxy.
viewStubs := {
  "Nautilus_View_stop_loading",
  "Nautilus_View_load_location",
  "Nautilus_View_title_changed",
  "Nautilus_View_history_changed",
  "Nautilus_View_selection_changed"
}

% The set of stubs used by window proxy.
windowStubs := {
  "Nautilus_ViewFrame_report_load_underway",
  "Nautilus_ViewFrame_report_load_compplete",
  "Nautilus_ViewFrame_report_load_failed",
  "Nautilus_ViewFrame_set_title",
  ...
}

% Pattern for window-to-view scenarios
windowToView := {scenarioChain:
  (idl_stub in viewStubs) and
  (emitter equal "gtk_signal_emit")
}

% Pattern for view-to-window scenarios
viewToWindow := {scenarioChain:
  (idl_stub in windowStubs) and
  (emitter equal "gtk_signal_emit")
}
```

Sets and patterns were used to extract all the scenarios defined by windowToView and viewToWindow. These extracted scenarios are quite useful for explaining how the Nautilus window collaborates with various views through CORBA and validating the layered communication architecture of Nautilus. Due to size limitations, we choose to show only three scenarios.

The scenario load_location is an instance of pattern windowToView. Table 4 gives this match. It represents a sequence of function calls going from the Nautilus window to the music view when a location change is made by a Nautilus user.

Upon being signaled to load a new location, the music view starts to inform the Nautilus window of the status of its loading process. Two of the matched instances of pattern viewToWindow are found relevant to this loading process: report_load_underway (see Table 5) and report_load_complete (see Table 6).

The three scenarios in Tables 4-6 all use the function music_view_load_location_callback. We looked at the source code of this function, and found that nautilus_music_view_load_uri was called after nautilus_view_report_load_underway but before nautilus_view_report_load_complete. With this new information, we were able to construct a story to connect all these three scenarios: (1) the Nautilus window notifies the music view to load a new location; (2) the music view gets this notification and reports "load is underway"; (3) the music view starts the actual loading and reports "load is complete" after the loading is done.

load_location (window-to-view)	
variable	**matched function**
request	update_view
relay	nautilus_view_frame_load_location
idl_stub	Nautilus_View_load_location
idl_skel	_ORBIT_skel_Nautilus_View_load_location
idl_impl	impl_Nautilus_View_load_location
emitter	gtk_signal_emit
service	music_view_load_location_callback

Table 4. Scenario load_location

report_load_underway (view-to-window)	
variable	**matched function**
request	music_view_load_location_callback
relay	nautilus_view_report_load_underway
idl_stub	Nautilus_ViewFrame_report_load_underway
idl_skel	_ORBIT_skel_Nautilus_ViewFrame_report_load_underway
idl_impl	impl_Nautilus_ViewFrame_report_load_underway
emitter	gtk_signal_emit
service	load_underway

Table 5. Scenario report_load_underway

report_load_complete (view-to-window)	
variable	**matched function**
request	music_view_load_location_callback
relay	nautilus_view_report_load_complete
idl_stub	Nautilus_ViewFrame_report_load_complete
idl_skel	_ORBIT_skel_Nautilus_ViewFrame_report_load_complete
idl_impl	impl_Nautilus_ViewFrame_report_load_complete
emitter	gtk_signal_emit
service	load_complete

Table 6. Scenario report_load_complete

4.3 Scenario Modeling

After the extraction of the communication scenarios for Nautilus, we used UML collaboration diagrams [5] to model them. The three collaboration diagrams in Figure 7 show their corresponding scenarios as annotations to the layered communication architecture of Nautilus (see Figure 4). In each scenario diagram, numbers show the order in which the interaction steps occur.

For a better understanding, we will take the scenario `load_location` as an illustrative example and describe it in detail. A Nautilus user enters a location, */home/eazel/Music/The Berlin Recitals*, in the location bar (see Figure 1). The Nautilus window catches this event, and invokes a call `update_view` on itself (step 1). A message `nautilus_view_frame_load_location` is sent to the view proxy (step 2), which then transforms the received message to `Nautilus_View_load_location` (step 3) and sends it through the underlying CORBA (step 4). The window proxy receives the incoming message (step 5), and then it invokes a call `gtk_signal_emit` on the GTK_Signal (step 6), which signals the music view to use function `music_view_load_location_callback` to load the new location, *The Berlin Recitals* (step 7). After the completion of the callback function, a music view for *The Berlin Recitals* is displayed to the user.

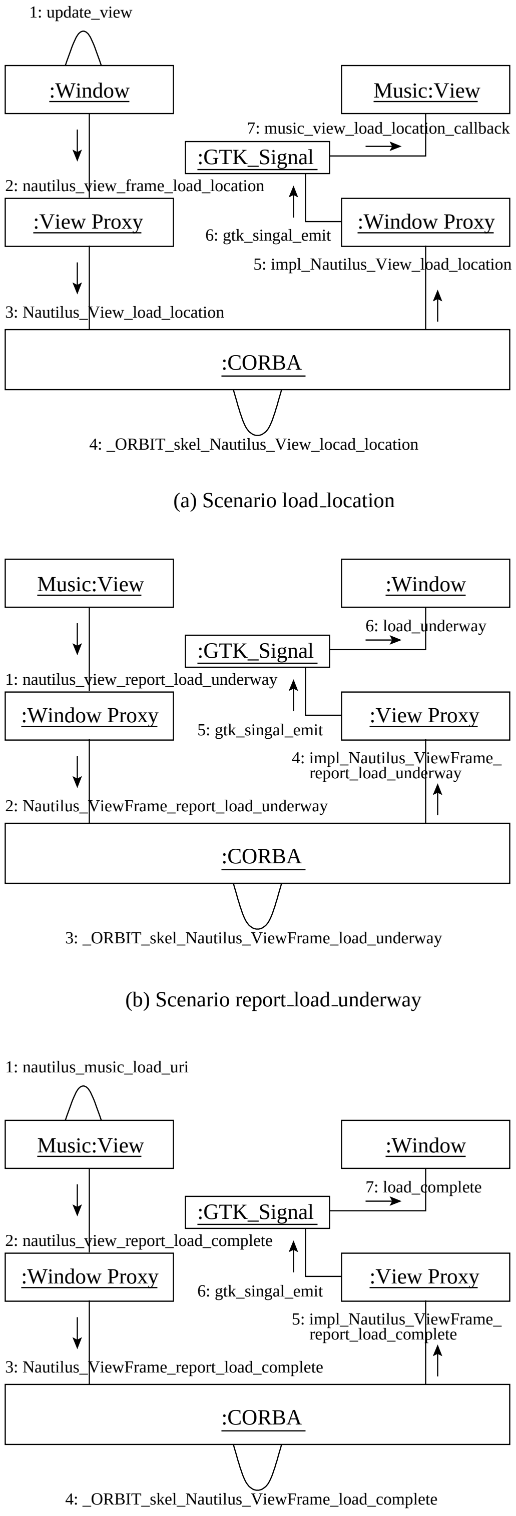
The other two scenario diagrams in Figure 7 illustrate the two view-to-window scenarios, which are caused by `music_view_load_location_callback`. They are straight forward, and we will omit the description of them in this paper.

For each scenario that captures the communication flow from the window to the view (e.g., Figure 7(a)), there must be several response scenarios, which describe the communication flow from the view back to the window (e.g., Figure 7(b) and Figure 7(c)). In both directions, only a small set of interface routines (see `viewStubs` and `windowStubs`) are used for communication. So, we can extract a fairly small set of communication scenarios.

By modeling these extracted scenarios, we produced a set of scenario diagrams, which are used as annotations for Nautilus. These diagrams improved our understanding of Nautilus by showing the interactions between the subsystems of Nautilus and the support technology CORBA.

5 Discussion

Graph patterns provide software engineers with a flexible approach to explore the structural aspects of a software system. One particular use of graph patterns is to extract scenarios from the implementation of a software system, as demonstrated in this paper.


1: update_view
:Window
Music:View
7: music_view_load_location_callback
:GTK_Signal
2: nautilus_view_frame_load_location
:View Proxy
6: gtk_singal_emit
:Window Proxy
5: impl_Nautilus_View_load_location
3: Nautilus_View_load_location
:CORBA
4: _ORBIT_skel_Nautilus_View_locad_location
Music:View
:Window
6: load_underway
:GTK_Signal
1: nautilus_view_report_load_underway
:Window Proxy
5: gtk_singal_emit
:View Proxy
4: impl_Nautilus_ViewFrame_
report_load_underway
2: Nautilus_ViewFrame_report_load_underway
:CORBA
3: _ORBIT_skel_Nautilus_ViewFrame_load_underway
1: nautilus_music_load_uri
Music:View
:Window
7: load_complete
:GTK_Signal
2: nautilus_view_report_load_complete
:Window Proxy
6: gtk_singal_emit
:View Proxy
5: impl_Nautilus_ViewFrame_
report_load_complete
3: Nautilus_ViewFrame_report_load_complete
:CORBA
4: _ORBIT_skel_Nautilus_ViewFrame_load_complete

(a) Scenario load_location

(b) Scenario report_load_underway

(c) Scenario report_load_complete

Figure 7. Modeling communication scenarios

By extending Grok for graph pattern matching, we are able to investigate many different kinds of structural interactions in a software system (involving function calls, event interactions, etc.) within the PBS architecture recovery framework. This is not meant to imply that Grok, as extended, is the only means for extracting scenarios by way of graph patterns. There are other approaches, for example, SQL queries and Prolog unification.

5.1 SQL Queries

We could have used Structured Query Language (SQL) instead of Grok to match graph patterns. This is because that facts extracted from a source program can be stored in a relational database, and because Grok's graph pattern rules can be translated to SQL query statements. Why did we choose to develop our own tool for extracting scenarios with graph patterns? There are three reasons:

1. Graph patterns are more intuitive for specifying graph-like structures, whereas SQL is designed for table-oriented computations.

2. Plain text files and semi-structured files are used extensively during the pipelined analysis supported by PBS. It is desirable to have a tool to work on those files directly rather than submit queries to a database.

3. Grok has powerful support for algebraic operations. It only took us a small amount of time to extend Grok for graph pattern matching through the reuse of the algebraic operations inside Grok.

5.2 Prolog Unification

Prolog is a powerful programming language used for solving problems that involve objects and relationships between objects [13]. In Prolog, the process of matching graph patterns is referred to as *unification*. For example, the `pattern1` described in Section 3 can be expressed in a Prolog clause as follows:

```
pattern1(C, F, G, H) :-
                funcDef(C, F),
                funcDef(C, G),
                funcDef(C, H),
                funcCall(F, G),
                funcCall(G, H).
```

The graph patterns expressed in Grok are quite similar to those in Prolog. The main difference is that the syntax of graph patterns in Grok is more consistent with the data formats supported by the PBS toolkit: RSF and TA [10]. Our goal was to make graph pattern matching an integral part of Grok, and we were able to achieve this by extending the Grok language with algebraic operations or Prolog unification [14, 15].

5.3 Lessons Learned

In our analysis of Nautilus, we extracted (using *cfx*) and prepared (using scripts) a factbase for Nautilus based on the static analysis of source code. Though this factbase contained no runtime facts about the dynamic behavior of Nautilus, it was quite useful for understanding the layered communication architecture of Nautilus by extracting scenarios.

Driven by the need to understand a software system, a software engineer may choose different extraction methods, either static methods (e.g., parsing and scripting) or dynamic methods (e.g. debugging and profiling). Runtime facts extracted during code execution can be used to validate the ordering of interactions in a scenario that is extracted using graph patterns. On the other hand, facts extracted in static scenario analysis can provide useful guidelines for dynamic extraction.

One of the troubles we had with Nautilus was identifying a useful set of scenarios, because substantial knowledge about the important tasks of the system were required. Our experience showed that such knowledge could be accumulated during architecture extraction. In our analysis, the *trial-and-error* method was applied to extract useful scenarios. We carried out many iterations of defining patterns before we got useful matches.

After we extracted a set of useful scenarios, we chose to model them using UML collaboration diagrams, instead of sequence diagrams. This is because a collaboration diagram (e.g., Figure 7(a)) can have a direct layout mapping to a software architecture (e.g., Figure 4). Scenarios can be used as annotations for software architectures, augmenting programmers' understanding of a software system. Many difficulties we had in understanding Nautilus's architecture were mitigated by the extraction of useful scenarios. For example, we found that the initialization of view components exposed a similar mechanism as the one shown in Figure 4.

6 Related Work

To our knowledge, no previous work has been done to extract scenarios using graph patterns, but a variety of studies [16, 17, 18, 19, 20] are closely related to our work.

Guo, Kazman *et al.* applied SQL queries to extract structural patterns [16]. In their approach, the facts extracted from a source program are stored in a relation database, and a pattern recognition plan is developed as a set of SQL query statements. They focused on recovering design patterns employed in the implementation of a software system, for example, the Presentation Abstraction Control (PAC) pattern. Their method for describing a recognition plan is quite similar to the way we define a graph pattern. To carry out a pattern recognition plan, they need to translate it into SQL queries.

Jerding *et al.* used code instrumentation to extract the call trace from the execution of a program [17, 18]. Their pattern recognition algorithm is based on substring matching heuristics, which can be used to locate interesting execution scenarios from voluminous trace information. They also developed a prototype visualizer to model execution scenarios using a variation of Message Sequence Charts.

In [19], Systä used a debugging tool to trace runtime events during the execution of object-oriented programs. A large amount of event trace information was produced and then reduced to a reasonable amount by a CASE tool for extracting behavioral patterns. In her approach, behavioral patterns can be extracted using a string-based matching algorithm.

Pal described a technique of lifting low-level call sequences to illustrate dynamic interactions at the component level [20]. He first chose a set of scenarios. Then, for each of these scenarios, he used *gdb* to trace call sequences by defining breakpoints during debugging. The traced call sequences were abstracted up to the component level.

7 Conclusions

We presented an approach to extracting scenarios based on static information recovered from the source code of a software system. The approach requires no modifications of the source code and doesn't require profiling the executable. The approach is semi-automated. Graph patterns are used to describe the interactions between the components and a relational calculator is used to locate the matching patterns in the source code of the software system. We have shown through the Nautilus case study that scenario extraction was useful for improving a programmer's understanding of a software system's architecture.

References

[1] G. C. Murphy, D. Notkin and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proc. of the 3rd ACM Symposium on the Foundations of Software Engineering*, pp. 18-28, Washington, D.C., October 1995.

[2] I. T. Bowman, R. C. Holt and N. V. Brewster. Linux as a Case Study: It Extracted Software Architecture. In *Proc. of the 21st International Conference on Software Engineering*, Los Angeles, California, May 1999.

[3] R. Kazman, G. Abowd, L. Bass and P. Clements. Scenario-Based Analysis of Software Architectur. *IEEE Software*, 13(6):47–55, 1996.

[4] P. Kruchten. The “4+1” View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[5] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2000.

[6] Nautilus. http://nautilus.eazel.com, June 2001.

[7] GNOME. http://www.gnome.org, January 2002.

[8] H. Zimmermann. OSI Reference Model – the ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980

[9] R. C. Holt. Software Architecture Abstraction and Aggregation as Algebraic Manipulations. In *Proc. of CASCON'99*, Toronto, Canada, November 1999.

[10] The Portble Bookshelf. http://www.swag.uwaterloo.ca/pbs, September 2001.

[11] The GTK+ Online Reference. http://www.gtk.org, January 2002.

[12] H. Müller, O. Mehmet, S. Tilley and J. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, Vol. 5(4), pp. 181–204, December 1993.

[13] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*, 4th edition. Springer-Verlag, 1994.

[14] GNU Prolog, version 1.2.8. http://pauillac.inria.fr/ diaz/gnu-prolog, October 2001.

[15] XPCE/SWI Prolog, version 5. http://www.swi-prolog.org, April 2002.

[16] G. Y. Guo, J. M. Atlee and R. Kazman. A Software Architecture Reconstruction Method. In *Proc. of the 1st IFIP Working Conference on Software Architecture*, pp. 15–33, San Antonio, Texas, February 1999.

[17] D. Jerding, J. Stasko and T. Ball. Visualizing Interactions in Program Executions. In *Proc. of ICSE'97*, pp.360–370, Boston, Massachusetts, May 1997.

[18] D. Jerding and S. Rugaber. Using Visualizatin for Architectural Localization and Extraction. In *Proc. of the 4th WCRE*, pp.56–65, Amsterdam, Netherland, October 1997.

[19] T. Systä. Understanding the Behavior of Java Programs. In *Proc. of the 7th WCRE*, Brisbane, Australia, November 2000.

[20] C. Pal. A Technique for Illustrating Dynamic Component Level Interactions Within a Software Architecture. In *Proc. of CASCON'98*, pp. 134–146, Toronto, Canada, November 1998.