Scenario visualization in scenario-based specification mining

Korolev, D.M.

Award date:
2015

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

MASTER THESIS

Scenario visualization in scenario-based specification mining

Author:
D. Korolev

Supervisor:
dr. D. Fahland

Committee:
prof.dr.ir. J.J. van Wijk
dr. J. Schmaltz
dr. D. Fahland

Eindhoven, March 30, 2015

Abstract

Maintenance of software systems requires not only introducing changes, but also understanding of the existing code. The latter can be complicated due to the system specification, which is often missing, incomplete or outdated. Scenario-based specification mining helps to solve this problem by automatically discovering specifications using system logs. The output consists of a number of live sequence charts (LSCs), each of which represents an individual scenario of system usage.

In this thesis, we present a technique for interactive visualization of mined scenario-based specification. We identify different use case and formulate corresponding requirements. Basing on them, we suggest a solution that consists of two separate views, which complement each other. The first view helps to see inter-dependencies between scenarios by depicting a specification as a graph where nodes represent scenarios and edges represent an execution flow. The second view allows observing a particular scenario in the form of an LSC extended with context information. Users are provided with different kinds of interaction, which include several filtering modes and adjustable generalization. To support the developed technique, a software tool has been created and used to validate the approach on a case study.

Contents

1. Introduction....1
1.1. Focus....1
1.2. Organization....1
2. Preliminaries....2
2.1. Live Sequence Charts....2
2.2 Scenario-based specification mining....3
3. Problems in visualizing scenario-based specification....4
3.1. Problem statement....4
3.2. Goals....4
3.3. Use cases....5
3.4. Literature review....5
3.5. Requirements....8
4. Scenario graph....9
4.1 Showing static or dynamic information....9
4.2. Showing events or scenarios....11
4.3. Handling scenario fragmentation and overlap....12
4.4. Dynamic features of a scenario graph....14
4.5. Building a scenario graph....15
5. Detailed view....21
5.1. Showing LSCs in a scenario graph or in a separate area....22
5.2. Handling scenario fragments visualization....23
5.3. Tracing user stories....24
5.4. Handling AND nodes....27
6. Filtering....30
6.1. Scenario-based filtering....30
6.2. Object-based filtering....34
7. Generalization....39
8. Implementation and evaluation....45
8.1. Implementation....45
8.2. Evaluation....46
9. Conclusion and future work....57
Bibliography....58
Appendix A....59

1. Introduction

Software specification is a very important part of any software product, which describes functionality and technical aspects of a software system. When a change should be made to a software product, system specification that defines relations between software modules can help to understand the consequences of that change. Incomplete or outdated specification is a major factor in the maintenance cost, which is the most expensive stage of the software lifecycle. At this, 50% of the maintenance cost is due to comprehending or understanding an existing code base [1]. Thus, updating specification automatically can potentially save a lot of resources. For this purpose there were created a number of approaches. Some of them use source code to mine specification, while the others use logs that are created when the system is being used.

One of the latter approaches is called scenario-based specification mining and produces a set of scenarios that occur in a system in the form of live sequence charts (LSCs). Since usually the number of mined scenarios is usually high, they should be somehow visualized to make the system analysis more effective.

1.1. Focus

The focus of this thesis is on interactive visualization of mined scenario-based specifications. The goal is to help the user to comprehend a system behavior and to gain insights into a system structure, which are difficult to get when analyzing scenarios one by one.

We take the set of mined scenarios as an input and visualize them in such a way, that the user can track the execution flow and discover inter-dependencies between scenarios. We present a view that is a graph showing the order in which the scenarios appear during the system execution. In this graph, the user can also see choices in the system and places where scenarios overlap. For every scenario we add context information to help the user to understand how and where within the system this scenario is used. To easier navigate through complex systems, we implement filtering mechanisms. Also, generalization is used to be able to eliminate possible incompleteness of logs.

1.2. Organization

This thesis is structured as follows. Chapter 2 contains preliminary information concerning live sequence charts and scenario-based specification mining. In Chapter 3 we formulate use cases of how the visualization can be used and on the basis of these use cases we define corresponding requirements. Also, in this chapter we make a literature review. Chapter 4 explains the idea of a scenario graph, which is one the most important notion of our approach. Chapter 5 discusses ideas behind a detailed view, which represents a scenario for the selected node from the scenario graph with added context information. Chapter 6 presents two modes of filtering we use: scenarios-based and object-based ones. Generalization is introduced in Chapter 7. In Chapter 8 we describe the implementation issues and discuss evaluation of our approach at the example of logs obtained for CrossFTP tool. Finally, Chapter 9 contains conclusions and suggestions for future work.

2. Preliminaries

2.1. Live Sequence Charts

Live Sequence Chart (LSC) is an extension of the Message Sequence Charts (MSC), which are a well-established visual formalism for the description of inter-working of processes or objects [2]. MSC is a popular description of specification with visual representation of involved objects as vertical lines, and messages between them as arrows going from the source to the target line, according to their occurrence order [3]. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU [4]. The standard defines the MSC syntactic constructs supported by formal semantics.

Despite the widespread use of MSCs, several fundamental issues have been left unaddressed. One of the most basic of these is that MSCs do not provide means to distinguish *mandatory* and *possible* behavior. To address this issue, Damm and Harel in [5] introduced the notion of LSC which allows distinction between possible and necessary behavior on the level of the entire chart and for the separate events, objects, and conditions.

The basic elements together with their graphical representation for instances have been adopted from MSCs:

- *Instances/process/objects* are represented with vertical line with its name on top of it;
- *Messages* between objects are represented with arrows going from the source instance to the target one, and their vertical position (order) corresponds to their actual order.

A tuple of the form <source object, target object, message> describes an *event*.

Since the main idea of LSC is to make a distinction between possible and necessary behavior, most LSC elements can be designated to belong to either one category or the other. This distinction is also expressed graphically, which contributes largely to the easy understanding of LSC specifications. Mandatory elements are depicted by solid lines, possible ones by dashed lines or filled with different colors.

More advanced constructs, like conditions, if-then-else, loops, etc. can also be expressed in LSC language (see [6]). A typical LSC consists of a pre-chart and a main chart, which are differentiated graphically by different color of the elements, or by shape of their frame. The intended semantics is – "if the pre-chart is satisfied in a run of the system, then following its completion the main chart must also be satisfied" [8].

In reality, a specification of the behavior of a system may often be characterized not only by linear invariants but also by possible choices [8]. To capture such behavior in LSC G. Sibay et al. extended the notion of LSC and defined an "existential conditional LSC" in [9]. Such a branching LSC specifies the semantics of per-chart – main chart relation as following: "when the pre-chart has occurred, the system must be able to perform the main chart". In the current thesis by LSC we mean existential conditional LSC.

2.2 Scenario-based specification mining

Software specification is a document describing the software system, and it becomes very important artifact when there is a need to make changes. Unfortunately, specification may be incomplete, or become outdated when software system evolves. In this case, an automated recovery of software specification can be extremely useful for maintenance of the system.

Scenario-based specification mining is one of the methods to gather the system specification. In this work we use the system specification mined with technique described in [8]. The input for such a method is

- a set of execution *traces* (sequences of events) that have been recorded while the application was used,
- and two parameters, which are commonly used metrics in data mining: a *support* threshold sets how often a particular LSC has to occur in the log, and a *confidence* threshold sets the fraction of times the property specified in the discovered LSC has to hold (allowing the LSC to be violated on some occurrences).

The output consists of a set of *scenarios* (sequences of events) in form of existential conditional LSCs that is correct and complete with regard to the support and confidence.

3. Problems in visualizing scenario-based specification

This chapter discusses problems that are faced when scenario-based specification shall be visualized. First, we explain how mined specifications can be used and which questions can be answered by using them. Then, we make an overview of how the problem of specification visualization is addressed in other works. Finally, we perform requirements elicitation for the tool for visualizing scenario-based specifications.

3.1. Problem statement

Traditionally, system specifications are supposed to be created by domain experts before the implementing stage. However, a specification may change later while developing and maintaining a system. Since a specification is the main artifact that describes system purpose, functionality and possible behavior, it should be clear and understandable for any domain expert who uses it. This is usually the case when a specification is created manually. However, things may be different when a specification is mined automatically since it contains a lot of information, which should be somehow processed to be interpreted by a human. Therefore, to help domain experts to analyze automatically mined specifications, visualization techniques should be used.

One of techniques for mining specifications consists in discovering scenarios from logs that are created while a system is being used. The result of this technique is a set of scenarios that are represented as a sequence of events. There can be inter-dependencies between scenarios, which are difficult to depict while observing scenarios separately. Also, by looking at a mined scenario we cannot determine its place in the system, i.e. we lose the information about occurrences of this scenario. So, we distinguish two problems:

- visualization of a single scenario in the context of a system;
- visualization of dependencies between scenarios.

3.2. Goals

Within this project we aim to enable users to analyze mined scenario-based specifications by:

- providing techniques for visualization of a single scenario in the form of LSC extended with context information;
- providing techniques for visualization of dependencies between scenarios based upon execution flow and scenario overlapping;
- providing a tool with support of suggested visualization techniques allowing different kinds of user interaction.

Extending LSCs with context information leads to better understanding of the role of a scenario in the whole system and decreasing the amount of time needed for analysis of a specification. Revealing dependencies between scenarios allows gaining insights into the structure of a system. With the help of interaction techniques it becomes possible to navigate through specifications focusing on those parts of a system that are interesting for analysis.

A proper visualization provided to the user results in better system comprehension, which in its turn reduces the amount of efforts needed to accomplish testing of a system and to introduce changes in it.

3.3. Use cases

With respect to usual tasks that arise while analyzing scenario-based specifications, we define possible use cases and formulate related questions to be answered for every use case.

1. Track the execution flow. Tracking an execution flow helps to discover patterns of system usage. It also allows answering the following question: *In what order scenarios occur? Which scenarios occur at the beginning and which ones occur at the end?*

2. Observe individual scenarios in details. *Which scenarios occur more often than others? Where are all the occurrences of a particular scenario? What are possible predecessors and ancestors of a particular scenario? Are they always the same or different ones?*

3. Discover relations between scenarios. *Which scenarios overlap? In case scenarios overlap on some events, which are these events? What are the common objects for overlapping scenarios and how do they interact with each other?*

4. Observe the system in terms of selected scenarios or objects. *Which scenarios/objects are more important (i.e. used more often) than the others? In which scenarios are selected objects used?*

5. Discover loops and the same behavior. *Are there loops in the system? Are there states in the system where the possible continuations are the same?*

3.4. Literature review

Visualization of interdependencies between scenarios is one of the main challenges in understanding scenario based specifications. While each separate scenario can be visualized in the term of LSC, the visualization of relationships between them, such as sequential order, parallelism, or branching behavior is not a trivial task. There are several approaches of the visualization interdependencies between scenarios, which are focused on different aspects and can be considered as a complement to each other.

David Harel and Itai Segall introduced a program called SIV (for scenario inter-dependency visualization) in [7]. SIV represents a specification as a graph where each node represents a scenario and edges between nodes represent various types of dependencies, such as possible causality and synchronization relations (see *Figure 1*). It also allows filtering and aggregation for exploration of interesting aspects of the specification. The weakness of such a type of visualization is that it allows only a static view of a scenario-based specification.

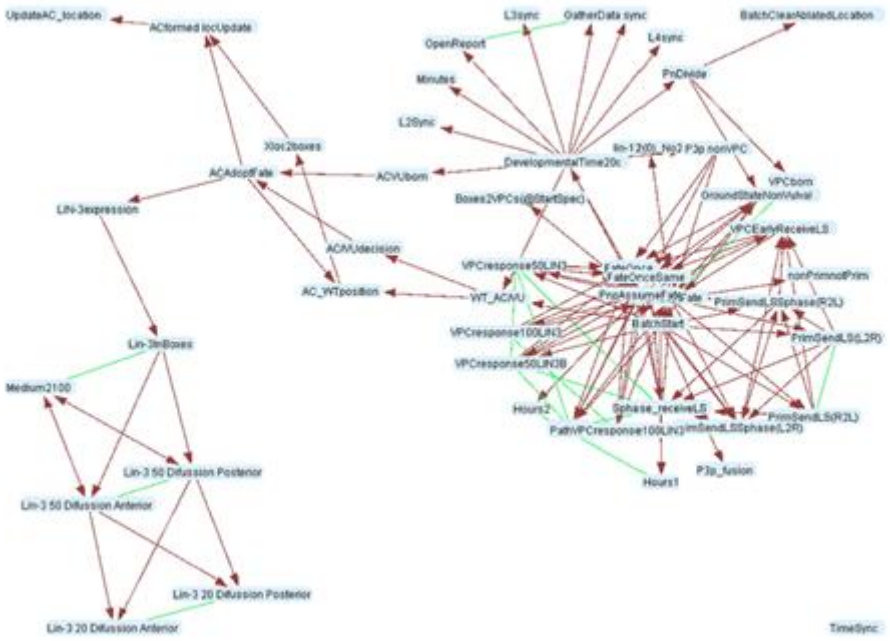


Figure 1. Visualization interdependencies between scenarios in SIV [7]

To compensate the drawback of SIV, David Harel together with Shahar Maoz offered a new technique for analyzing, visualizing, and exploring the execution traces in [10], which was implemented in a tool called Tracer (see *Figure 2*). Tracer takes as input a scenario-based behavioral model and an execution trace of the system, and uses a hierarchical Gantt chart to follow scenarios throughout execution. Tracer supports interactive trace exploration through different types of navigation, filtering, and comparisons.

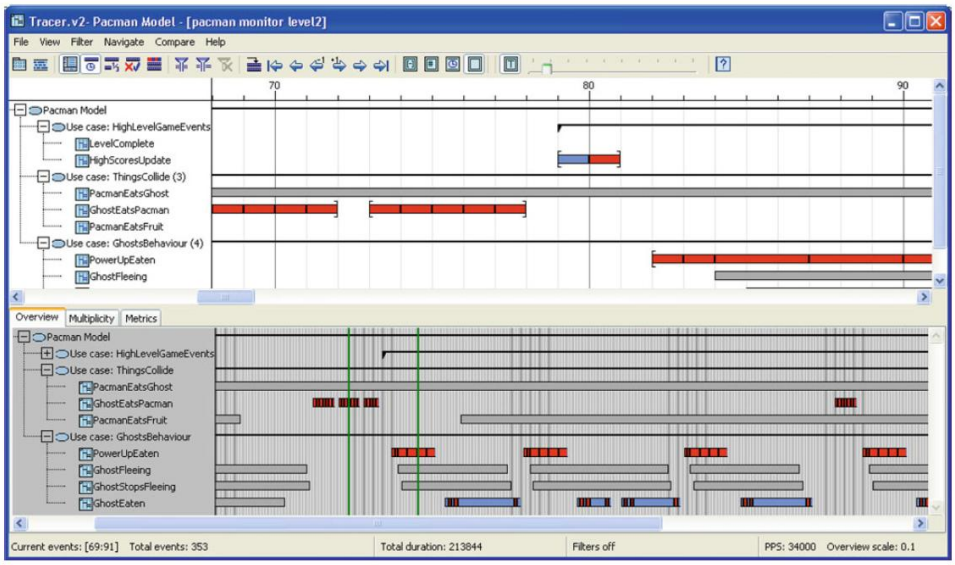


Figure 2. The Tracer main view and Overview [10]

In [11], Gordon et al. propose semantic navigations for scanning multiple connected LSCs to understand how they interact. The full specification is visualized with a huge multi-chart LSC (see *Figure 3*). The method uses weighted messages to create a semantic order that enables

semantic zooming and scrolling of interesting parts of a chart, while hiding/merging the rest. Unfortunately, such type of scenarios visualization is not suitable for the specifications with branching-time scenarios.

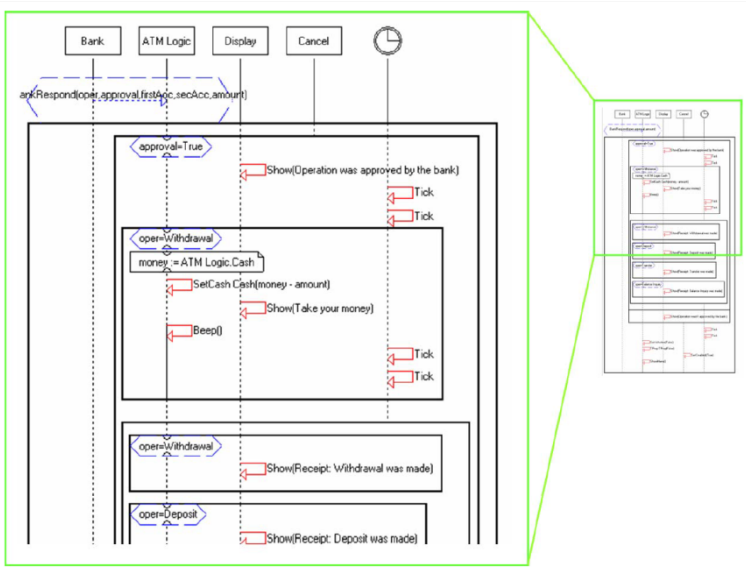


Figure 3. Multi-chart LSC [11]

Another way of visualization interdependencies between scenarios is presented in [12]. The system considered as a set of modules, called behavior threads (or b-threads). The b-threads can request events, wait for events and forbid events. The proposed visualization tool combines visualization of individual behaviors, information about bidding at synchronization points, and the resulting event trace. It supports navigation and filtering techniques, allowing investigation of the system behavior efficiently. The visualization tool uses a table-like display to visualize the run of a behavioral program (see *Figure 4*). Behavior threads are grouped by class and depicted in columns ordered by priorities. Synchronization points are represented by rows ordered by time.

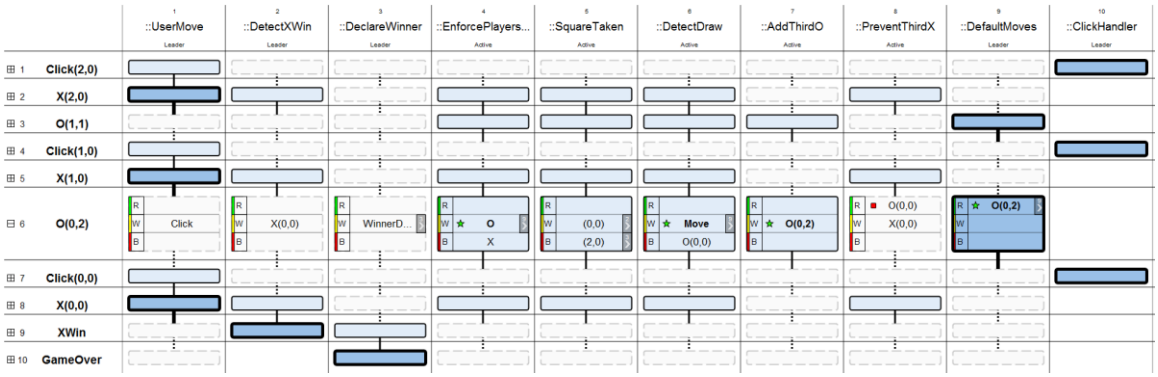


Figure 4. Visualizing a game of Tic-Tac-Toe [12]

3.5. Requirements

Basing upon the list of formulated use cases, we form a list of requirements for the tool that should be met.

1. The user should be able to see an execution flow based on scenarios in the form of a graph where nodes represent scenarios and edges represent an execution flow. Such a view can be constructed using an execution tree.

2. The user should be able to view a single scenario in details: an LSC corresponding to the selected node should be presented along with context information.

3. When analyzing some node in details, the user should understand which nodes are predecessors and which are successors of the selected node. It should be also possible to quickly switch to those nodes (thereby travelling along some trace).

4. It is likely that the user would often want to switch between the graph view and LSC view because while analyzing a scenario graph, it is desirable to be able to zoom in some scenario, and vice versa, when looking at some LSC, it is important to see its place in the whole system. So, it should be possible to see those two views at the same time.

5. It should be possible to generalize a scenario graph. Since the information from logs is almost always incomplete, some patterns found in the topology of the tree may be the result of such incompleteness, rather than a feature of a system. So, the user should be able to abstract such information and generalize an execution tree to make it smaller and more understandable. However, sometimes a complete generalization may lead to a situation when valuable information about system behavior is lost, so the user should be able to adjust a generalization rate.

6. To concentrate on particular scenarios (for example, on those that are used most often), it should be possible to filter out scenarios, thereby changing and simplifying the scenario graph. Filtering should be also applicable to objects.

4. Scenario graph

This chapter describes a notion of scenario graph, which is a projection of mined scenarios onto the execution tree. First, we discuss why it is important to take into account not only static information, but dynamic data as well. Then, we explore certain design decisions that meet requirements and use cases formulated in Chapter 3. Finally, we describe how a scenario graph is constructed and list corresponding algorithms.

As we have shown in Chapter 3, the user should not see a scenario in isolation, but in relation to other scenarios. Here, we look at different ways how context information can be added to scenarios. The following example will be used throughout this chapter to illustrate information discussed. Consider *Figure 5* where three scenarios are presented: scenario A consists of events a, b, e; scenario B consists of events b, c, f; and scenario C consists of events b, d, h. Blue color indicates that an event is in a pre-chart of a scenario, and red events compose main-chart. Note that scenario A has event b in its main-chart, while this same event is included in pre-charts of scenarios B and C.

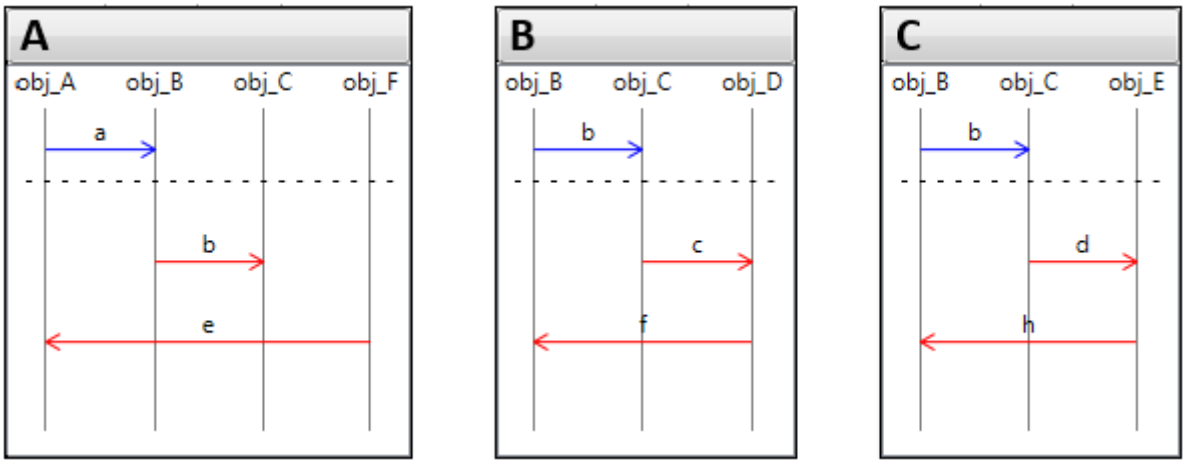


Figure 5. Scenarios A, B, and C

4.1 Showing static or dynamic information

When the specification has been mined, it is difficult to analyze the system by looking through scenarios one by one because there exist dependencies between them, which should be revealed and highlighted with the help of visualization techniques. A first idea that may come to one's mind is to use static information to build a graph of dependencies, where nodes represent scenarios and an edge is drawn between two scenarios A and B if the main-chart of scenario A partially or totally contains the pre-chart of scenario B. Such an approach is used in [6]. By applying this approach to the example introduced in *Figure 5*, we get the result depicted in *Figure 6*.

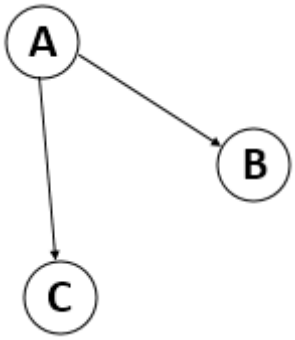
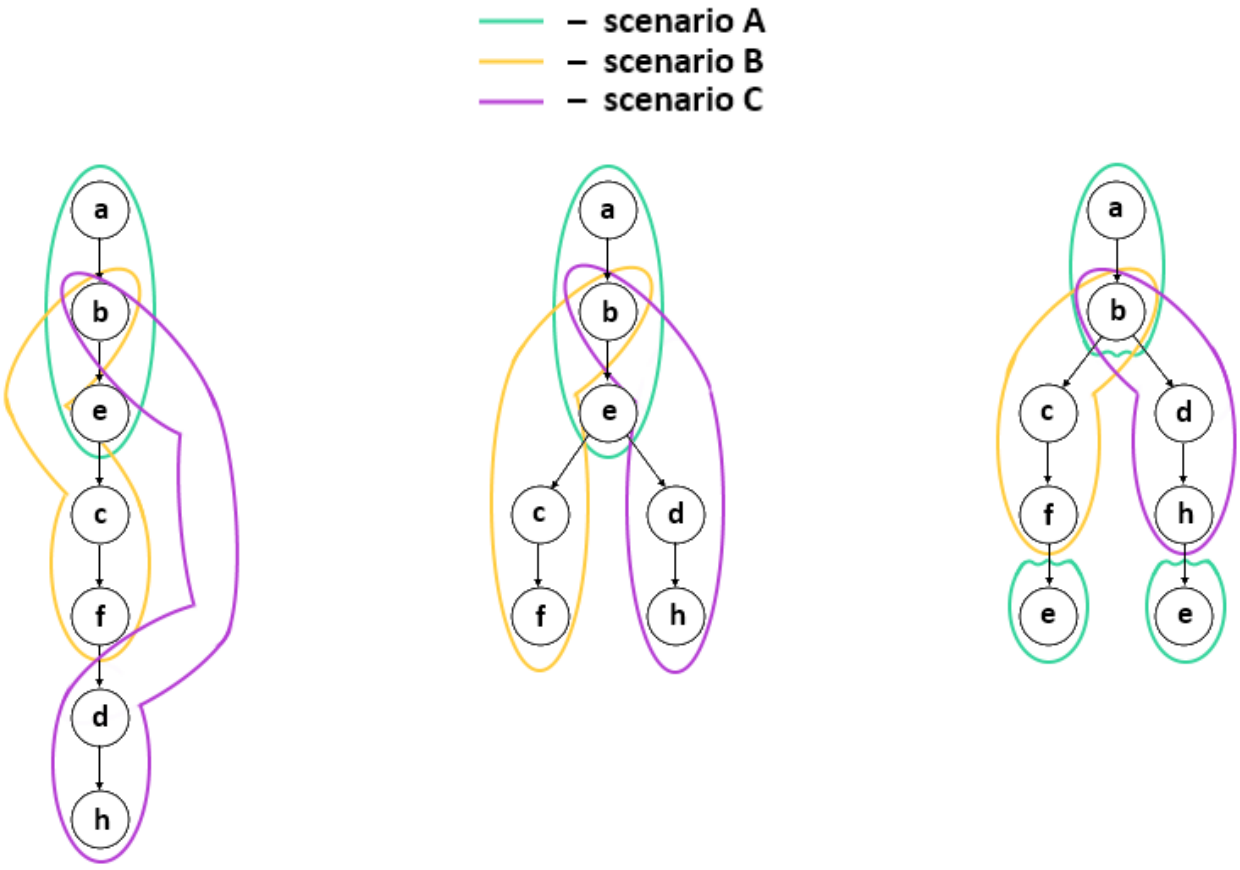


Figure 6. Static dependencies between scenarios A, B, and C.

In spite of the fact, that such a visualization immediately allows to see inter-dependencies between scenarios, this information is static and it does not show several crucial properties of the system and its usage, e.g. we cannot see if scenarios B and C are be executed only after scenario A has been finished or they may begin during execution of scenario A, thereby fragmenting it. We also cannot say if either scenarios B and C are both executed or there is a choice between them. Consider three different execution trees depicted in *Figure 7*.



(a) *(b)* *(c)*

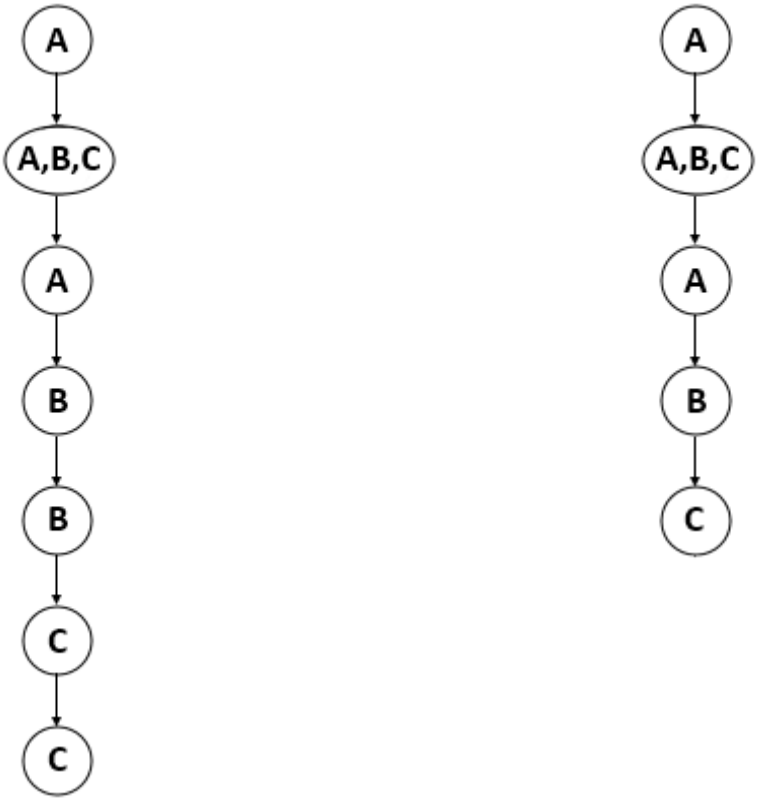
Figure 7. Possible execution trees consisting of scenarios A, B, C

Every tree represents different system behavior: (a) scenarios execute in a sequence: after an occurrence of scenario A, scenario B is executed, followed by execution of scenario C; (b)

scenario A is fully executed, and then there is a choice between scenarios B and C; (c) after scenario A is partly executed, either scenario B or scenario C occurs, and each branch ends with the final event of scenario A. By using static information shown in *Figure 6*, we cannot distinguish between these system behaviors. To be able to tell one behavior from another, we need to take into account dynamic information, such as execution trees.

4.2. Showing events or scenarios

Due to the fact that in an execution tree nodes correspond to events, such trees are usually large ones. To make visualization more understandable, we would like to use a higher level of abstraction and make a scenario to be a first-class citizen rather than an event. To do so, we project scenarios on an execution tree, such that nodes in the resulting tree correspond to scenarios rather than events as in the initial execution tree. *Figure 8a* shows the result obtained after applying such a projection to the tree depicted in *Figure 7a*. At this point, the number of nodes has not changed in comparison with the initial tree, while our goal is to reduce this number. To do so, we fuse sequential nodes if: (a) they have same labels; (b) they are from the same scenario occurrence. The result is shown in *Figure 8b*.



(a) *(b)*

Figure 8. Scenarios projected to the execution tree

Some events in an execution tree may be not covered by any scenario. In this case, we fuse such sequential nodes and represent the resulting node as a cloud node to denote that this node consists of events that have not been combined in a scenario.

4.3. Handling scenario fragmentation and overlap

According to the execution tree shown in *Figure 7a*, scenario A occurs only once. However, in *Figure 8b* there are three nodes labeled with this scenario. It happens because scenarios not just follow each other, but they also may overlap, thereby interrupting other scenario occurrences. Thus, scenarios may be fragmented. By looking at *Figure 8b*, we can no longer say how many times scenario A occurred. Another important piece of information lost after abstracting the execution tree concerns tracking progress of scenario execution: we now cannot say if scenario A just began before starting scenarios B and C or it was almost finished by then. We depict the information about scenario progress by adding a rectangle next to each node that shows which part of a scenario corresponds to this node. Such a rectangle consists of $n + m$ sectors, where n is a number of events in the pre-chart of the corresponding scenario, and m is a number of events in its main-chart. In case, an event is associated with a node, the corresponding sector will be filled with either blue or red color (depending on the fact if this event is a part of the pre-chart or of the main-chart). *Figure 9* shows the result obtained after adding progress indicators to the tree depicted in *Figure 8b*.

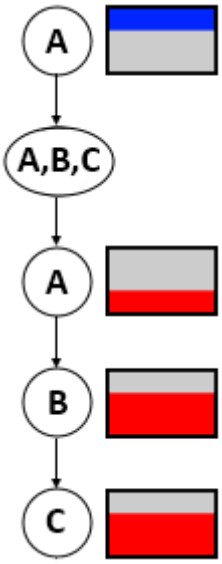


Figure 9. Adding progress indicators to nodes

Now, we can see where scenarios are fragmented and where different parts of scenarios occur. However, there are nodes where several scenarios overlap, e.g node A, B C in *Figure 9*. To be able to add scenario progress indicators to such nodes, we need to split them into multiples ones, such that one node corresponds to one scenario. After that, we can add progress indicators to every node as shown in *Figure 10*.

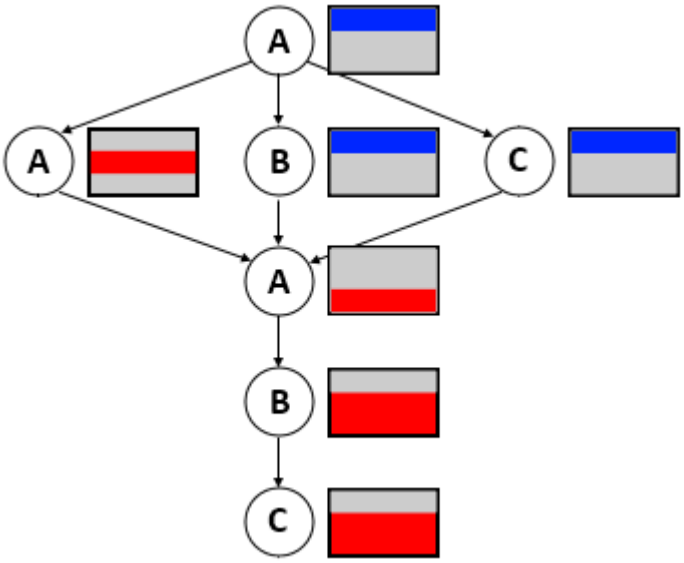


Figure 10. Splitting nodes where several scenarios overlap

Now, we can clearly see which parts of a scenario are covered by every node. We can also see an execution flow: an arrow from one node to another is a good intuitive indicator of a sequence. So, if there is an edge between nodes A to node B, then it means that first a part of scenario A occurs followed by execution of a part of scenario B. However, we need to introduce additional notation to be able to distinguish between parallel and alternative executions. We speak about p*arallel overlapping* when two scenarios occur simultaneously, while *alternative overlapping* occurs when there is a choice between multiple scenarios, and only one of these scenarios can be chosen as continuation. To distinguish between these two options, we introduce two new types of nodes in a scenario graph. They are AND nodes and OR nodes. AND nodes show that several scenarios overlap, while OR nodes indicate branching, i.e. a choice between several possible continuations. *Figure 11* shows how *Figure 10* looks like after adding an AND node where scenarios A, B and C overlap.

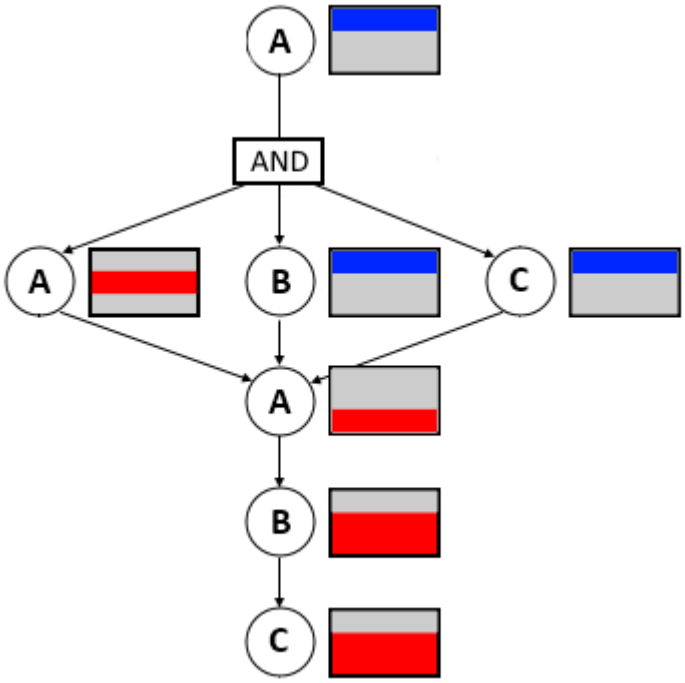


Figure 11. Scenario graph after inserting AND node

4.4. Dynamic features of a scenario graph

In the previous subsections we explained what a scenario graph is and how we can build it. At this, all the features we introduced so far had a static nature. However, having a static picture is often not enough to properly analyze a scenario graph. One such situation deals with scenario fragmentation because of which it may be difficult either to trace the full occurrence of a scenario or to tell one scenario occurrence from another one. When there are few nodes in the scenario graph, it may be easy to track different scenario occurrences by looking at progress indicators, but things get worse when the execution tree is large and the fragmentation rate is high. To solve this issue, we introduce the feature of selecting a node. When some node is selected, all the nodes that belong to the scenario occurrence of the selected node are highlighted. Figure *Figure 12* shows the scenario graph where the root node is selected. Blue frames indicate all the fragments of this scenario occurrence. To better understand the role of a particular scenario within the system, it is important to see when this scenario occurs. That is why when some scenario is selected, we also highlight all other occurrences of this scenario.

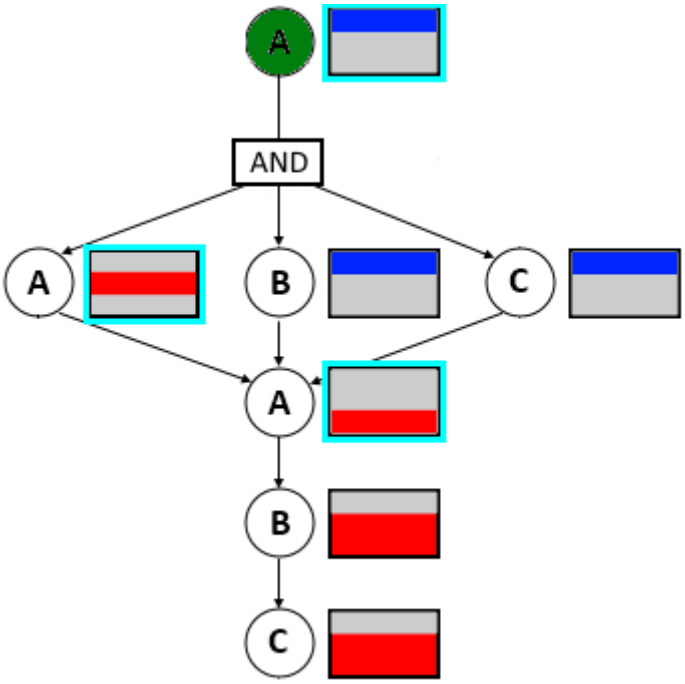


Figure 12. Scenario graph with selected node A and highlighted nodes corresponding to the same occurrence

Another dynamic feature, which we implement over a scenario graph, is zooming and panning, so the user can zoom in some part of the scenario graph and observe it by dragging the canvas. The user can also change the position of a node by dragging it and thereby changing the automatically computed layout.

4.5. Building a scenario graph

The design decisions that we made to build a scenario graph have been discussed and motivated in the previous subchapters. In this subchapter, we list the algorithms that are used to transform an execution tree into a scenario graph.

First, we use information about which event of the execution tree is part of which scenario. The way this transformation is performed will be explained using LSCs from the example depicted in *Figure 5* and the execution tree, which is shown in *Figure 13*.

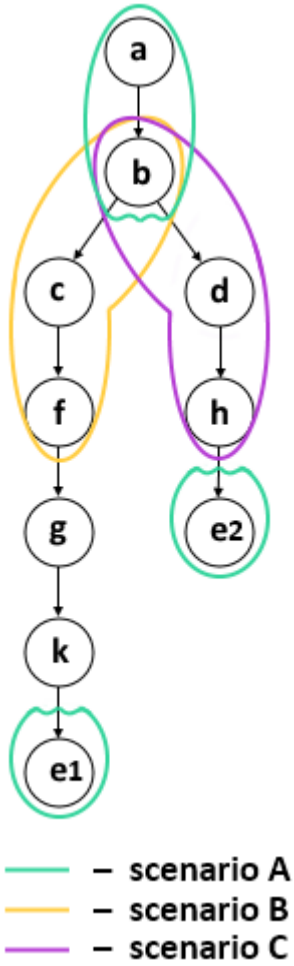


Figure 13. The execution tree

We can see that there are three scenarios: A, B and C. They all overlap on event b. For scenario A, event b is included in its main-chart, while this event is in the pre-charts of both scenarios B and C. Note that events g and k are not covered by any scenario.

The first step in transforming an execution tree into a scenario graph is creating a mapping between nodes of the execution tree and scenarios. A scenario is mapped on some node if this scenario includes the event, which the node represents. *Figure 14* shows nodes with attached information obtained after mapping. Coverage information is omitted to make a figure easier to understand.

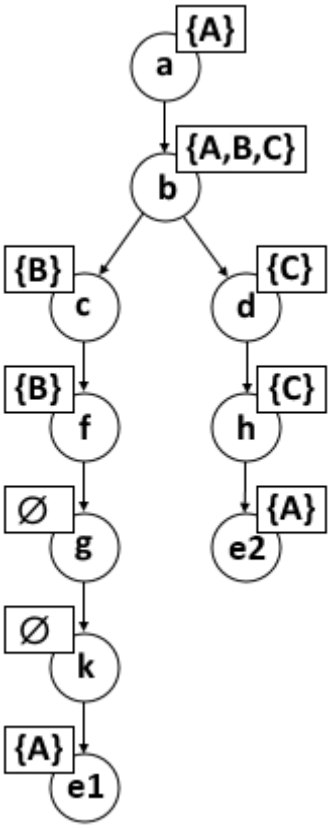


Figure 14. Mapping between nodes of the execution tree and scenarios

Since scenarios may overlap on events, several scenarios can be mapped on a single node. Thus, in *Figure 14* we can see that three scenarios A, B and C correspond to event b. Alternatively, some events may be not covered by any scenario, e.g. events g and k in *Figure 14*.

The second step in transforming the execution tree into the scenario graph is replacing event nodes with scenario nodes. We partition the execution tree into subtrees, where each subtree consists of one or more nodes that are labeled with same scenarios. At that, an out-degree of each node that is not a leaf of a subtree equals 1. *Figure 15* shows how the execution tree from the example is divided into subtrees. Each red ellipse represents a subtree.

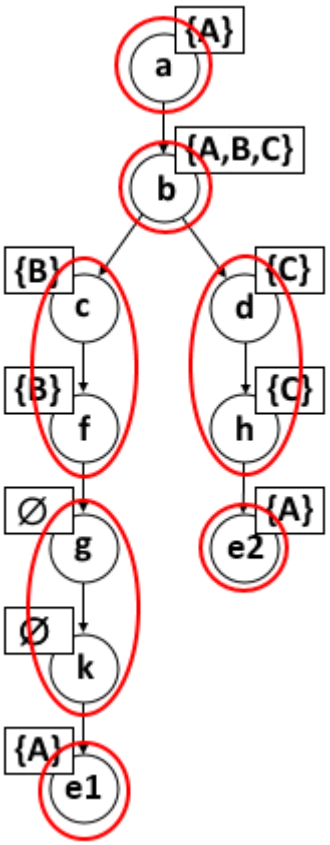


Figure 15. Partitioning the execution tree

Then, we compute the contraction of the execution tree under created partition, i.e. each subtree becomes a node and there is an edge between two nodes if two corresponding subtrees have been directly connected in the execution tree.

After this transformation is performed, there are two major differences between the initial execution tree and the resulting tree: (1) in the execution tree each node always corresponds to one event, and in the obtained tree a node may correspond to multiple events; (2) in the obtained tree, to each node there are attached scenarios that cover all events to which the node corresponds. *Figure 16* shows the result obtained after performing this transformation to the tree from the example.

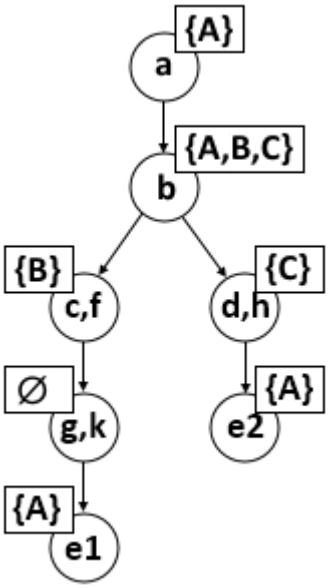


Figure 16. The execution tree after contraction based upon computed partitions

The next step is inserting OR nodes that indicate choices between several possible continuations. We insert an OR node if some node has more than one outgoing edges. At the example there is only one such node, which is node b. Inserting OR nodes leads to the result shown in *Figure 17*.

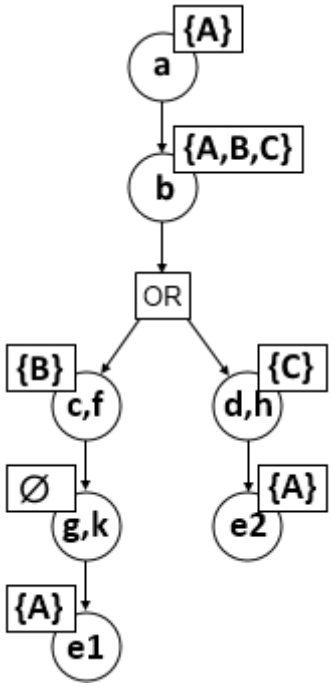


Figure 17. Scenario graph after inserting OR nodes

In a scenario graph that we want to construct, each node corresponds to maximum one scenario, so the next step in building the scenario graph consists in replacing nodes that correspond to multiple scenarios with several nodes, i.e. one node for each scenario. *Figure 18* shows the scenario graph after inserting AND nodes and changing labels to scenario names.

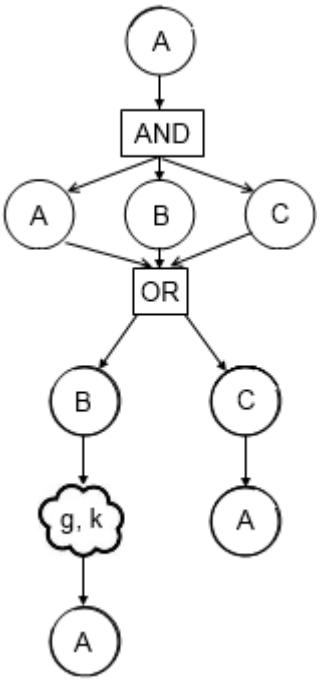


Figure 18. Scenario graph after inserting AND nodes

5. Detailed view

In Chapter 4, we have discussed how an execution tree can be transformed into a scenario graph. One of the key steps in this transformation was abstracting from events and making a scenario to be a first class citizen. Therefore, a scenario graph does not depict detailed information concerning scenarios, but we still must show such information somewhere else. We do it by introducing a so-called *detailed view* that is an LSC with context information attached.

The content of this chapter will be explained using the scenario graph and scenarios A, B, C and D that are depicted in *Figure 19*.

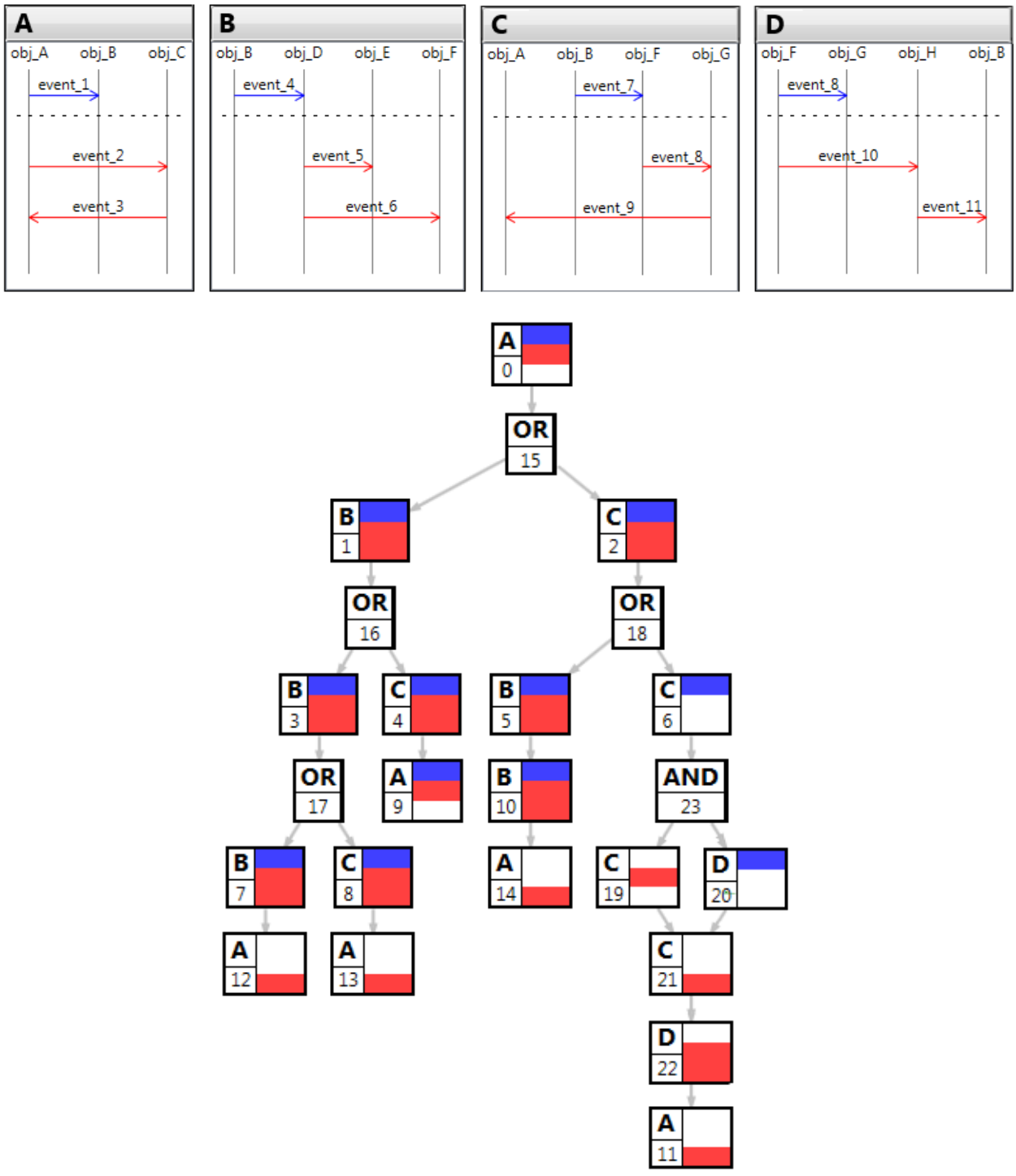


Figure 19. LSCs of scenarios A, B, C, D and the scenario graph used in Chapter 5

5.1. Showing LSCs in a scenario graph or in a separate area

In Chapter 4, we presented a feature of selecting a node in a scenario graph. When a node is selected, not only other nodes are highlighted in the scenario graph, but also the detailed view shows the LSC that corresponds to the selected node.

One of use cases formulated in Chapter 3 is to understand how a system is actually used by tracking paths travelled by users. To do so, it is convenient to traverse a scenario graph starting from the root: the user sees a scenario, its predecessor nodes and possible continuation. If the user is interested in which objects communicate within this scenario or which events happen there, the user might want to open a corresponding LSC and analyze it. When the user is done with a scenario, he chooses an ancestor node, i.e. a possible continuation, and continues studying a user path. From this point of view, a possible design decision is to expand the selected node directly in the scenario graph and show the LSC that corresponds to this node as shown in *Figure 20*.

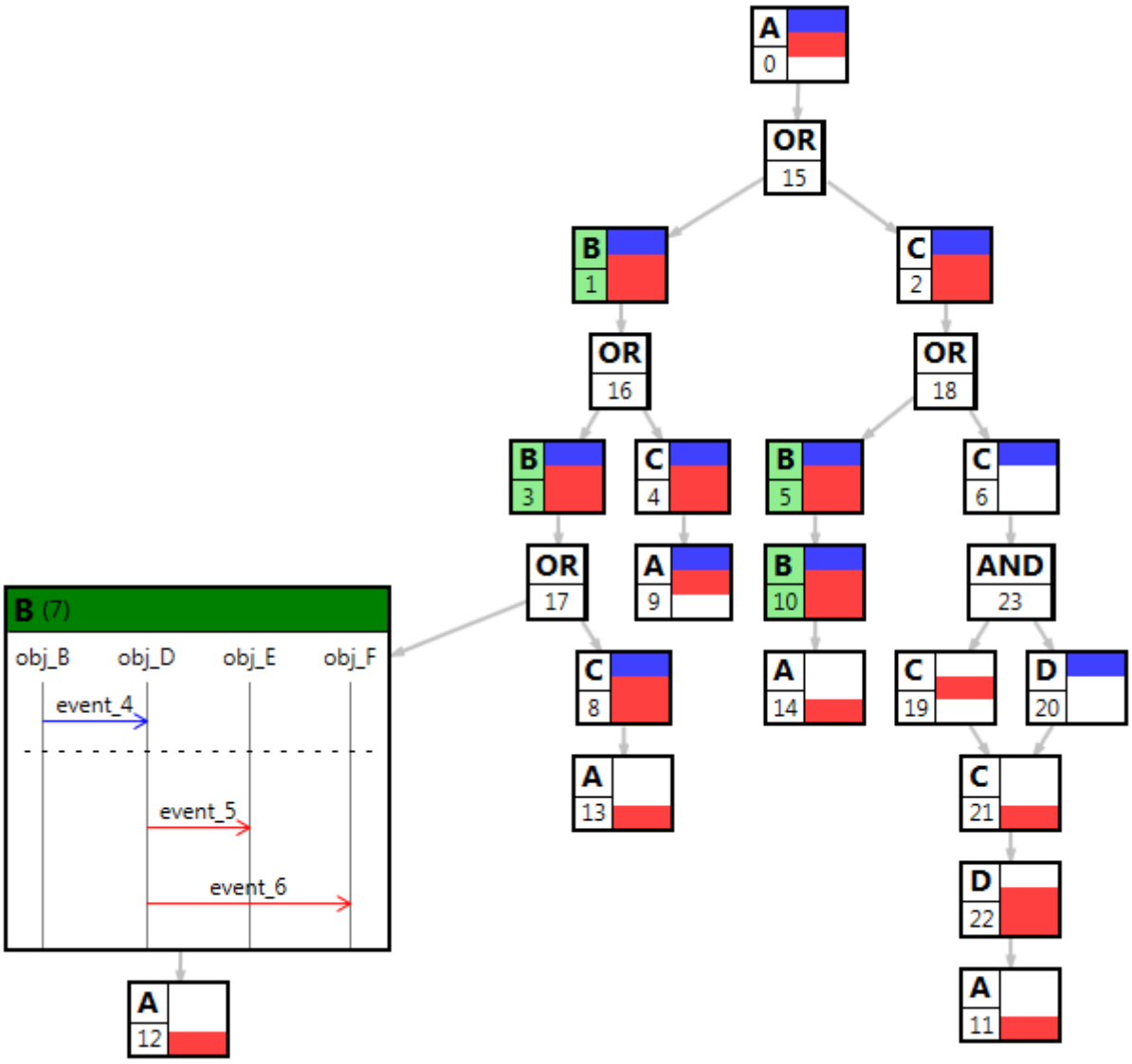


Figure 20. An expanded LSC inside the scenario graph

Such a decision has a clear advantage: the user can immediately see the context of the chosen scenario, i.e. its place in the whole graph, predecessor and ancestor nodes. For small systems that consist of simple scenarios this solution works well, but problems may appear when scenarios are large, i.e. they contain many interacting objects and a big number of events. In this case, showing a big LSC leads to the situation when the diagram takes much space on the screen and the context information such as the rest of the scenario graph cannot be visible any longer. Even in case an LSC is not that big, it is still takes quite a lot of the screen space and the user cannot see a big picture, i.e. the whole scenario graph. To resolve these issues, we separate LSC representation from a scenario graph as shown in *Figure 21*: in the upper part a scenario graph is displayed, while in the bottom part there is a detailed view of the LSC corresponding to the selected node. Between these two areas we place a splitter, so that the user can set the space on the screen that each of these areas takes.

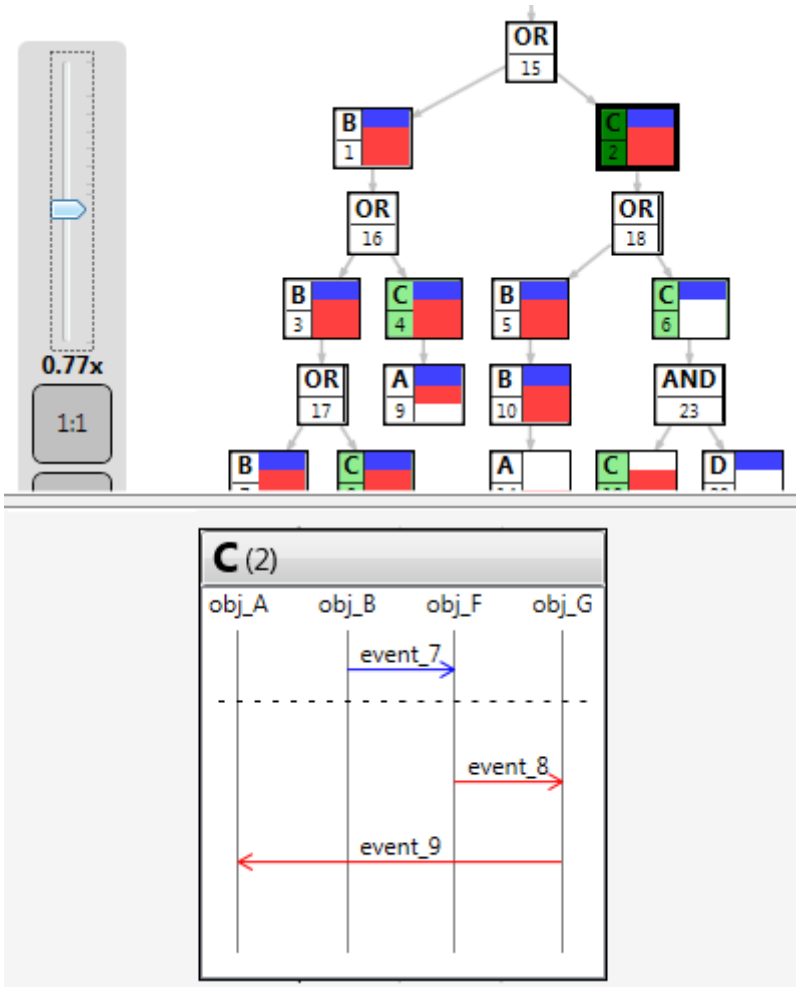


Figure 21. The scenario graph and the detailed view separated by the splitter

5.2. Handling scenario fragments visualization

In a scenario graph we presented in Chapter 3, every node except AND, OR and folded nodes corresponds to one scenario. When the user selects such a node, the corresponding LSC is displayed in the bottom area. However, some nodes depict not whole scenarios, but their fragments, which should be also visualized in some way. One solution to do this is to display only those events that form the corresponding scenario fragment along with objects which are

used in the filtered events communication. This design decision is a reasonable one in respect to an execution tree and a scenario graph, i.e. we show to the user exactly those events that are incorporated into the selected node. Consider *Figure 22* that shows the detailed view after selecting node 19. Since only one event is included in this node, all other events that form scenario C are filtered out, as well as objects: obj_A and obj_B.

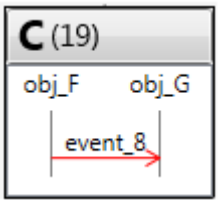


Figure 22. The detailed view for the scenario segment.
Events and objects that are not included into the segment are filtered out.

However, such visualization does not provide important information about a scenario and may even mislead the user as he can see only those objects that take part in the shown events. As a result, the user might conclude that some object is not used in the scenario while in fact it is an important actor for this scenario. Also, in case when a scenario is always fragmented in a scenario graph, the user will never see the full LSC of such a scenario, which makes analyzing even more difficult. Thus, we can conclude that by showing only those events that form a scenario fragment we lose a lot of crucial scenario context. To keep this context, we show the full LSC of a scenario where events that are not included in the scenario segment are faded out. *Figure 23* shows how the detailed view for node 19 looks with the context information added.

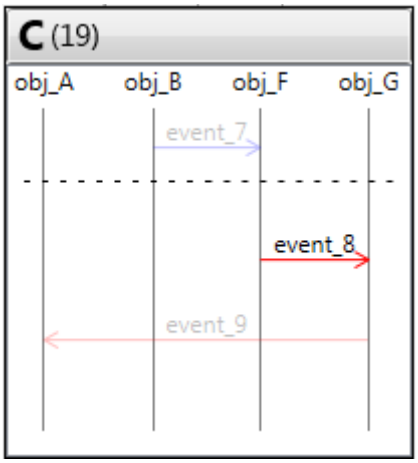


Figure 23. The detailed view for the scenario segment.
Events that are not included are faded out.

5.3. Tracing user stories

Since we have separated the detailed view from the scenario graph, there is a lack of context now, i.e. by looking at the detailed view of the selected node we cannot immediately name predecessors and ancestors of this node, which is an important feature for tracking user stories. On one hand, the user can refer to the scenario graph to get this information, but on the other hand such switching between views always distracts the attention. Additionally, in the scenario graph it can be difficult to quickly determine predecessors and ancestors due to the layout

algorithm, which produces different renderings for every graph and there is no guarantee that it will be always good for our purposes. That is why we make a decision to add more context information to the detailed view, namely predecessors and ancestors of the selected node. We place predecessor nodes at the top of the LSC and ancestor nodes below it, so they can be easily interpreted as execution flow, which goes from top to bottom. This context information is represented not by entire LSCs, but only by their abstractions that are these LSCs' names. The user can click such an abstraction and the corresponding LSC will be shown instead of currently displayed one, as if the node has been selected in the scenario graph. Thus, it is possible to start for instance from the root node and by choosing between possible continuations traverse the scenario graph to understand how the users have worked with the system. If an LSC is preceded/succeeded by AND/OR nodes in the scenario graph, we also include these nodes as context information. *Figure 24* illustrates this approach at the example of node 2.

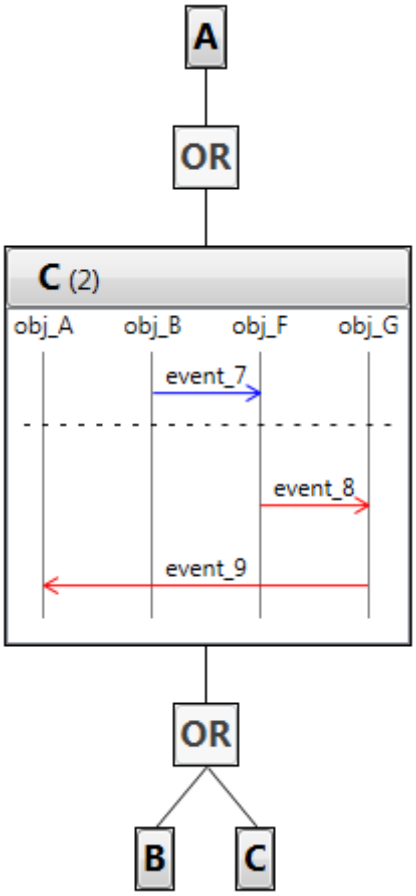


Figure 24. Predecessors and successors of node 2 added to the detailed view

Scenarios may occur in multiple places in the system, i.e. sets of predecessors and ancestors may differ for different occurrences of some scenario. When looking at the detailed view of the selected scenario, the user should be able to see where else this scenario occurs. To form a complete idea of scenario usage, we decided to add to the detailed view not only predecessors and ancestors of the currently selected node, but for the selected scenario form sets of all possible predecessors and ancestors among all nodes that correspond to this scenario. Consider the example shown in *Figure 25*. We can still see the preceding and succeeding nodes of the currently selected node 2: they are clickable and indicated by black color. But now we can see that there are different options of which scenarios can go before or after scenario C. Options that

correspond to nodes that cannot be visited from the currently selected node are drawn with grey color. If the user is interested in some predecessor/successor, he can pint at the grey node to find out identifiers of nodes that have that predecessor/successor.

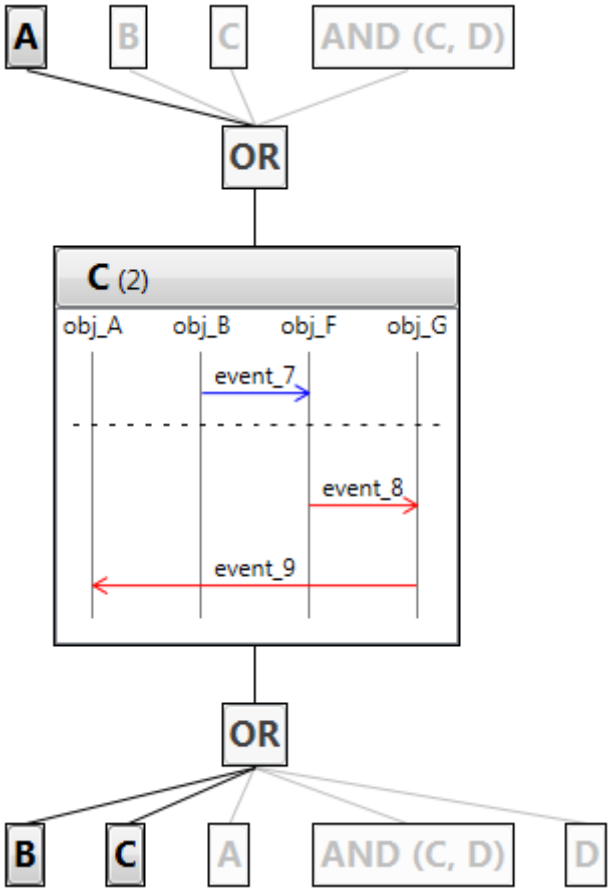


Figure 25. All the possible predecessors and successors of nodes that correspond to scenario C

Sometimes, patterns of scenario usage can be found in the system. The example of such a pattern: execution of scenario A always followed by execution of scenario B. By observing the scenario graph, it might be difficult to detect such patterns because a number of scenario occurrences can be large as well as the scenario graph itself. So, the user would need to pan the scenario graph to find all scenario occurrences, which is very uncomfortable and the information that is not shown at the screen can be easily missed or forgotten. To solve this problem, we present features for traversing the scenario graph in two additional ways.

First, we divide nodes that represent the same scenario into groups on the basis of predecessor and ancestor nodes. So, two nodes belong to one group if they have equal sets of predecessors and ancestors. We add buttons for switching between different groups; when switching to a different group, one of the nodes from this group will be selected in the scenario graph and in the detailed view. Each group can be thought as a way how a scenario can be used in the system. So, this traversing mechanism allows switching between user stories that include a certain scenario. Second, in one group there can be several nodes, so we add a feature of switching between such nodes. Using this feature, the user can see places in the system where a certain scenario occurs in the same way. *Figure 26* shows four buttons for these two types of traversing the scenario graph. The left pair of buttons is for switching between groups, while the right one is for switching

within the currently selected group. Numbers that are next to the buttons indicate how many different groups exist for the selected scenario and how many nodes are inside the selected group respectively. There are 5 nodes that correspond to scenario C in the scenario graph (see *Figure 19*) and they can be divided into 4 groups basing upon their predecessors and successors. So, two of those nodes are belong to one group and we can switch between them using the right pair of buttons. *Figure 26* shows that nodes in this group have scenario B as a predecessor and scenario A as a successor. By looking at the scenario graph, we can see that those two nodes are 4 and 8.

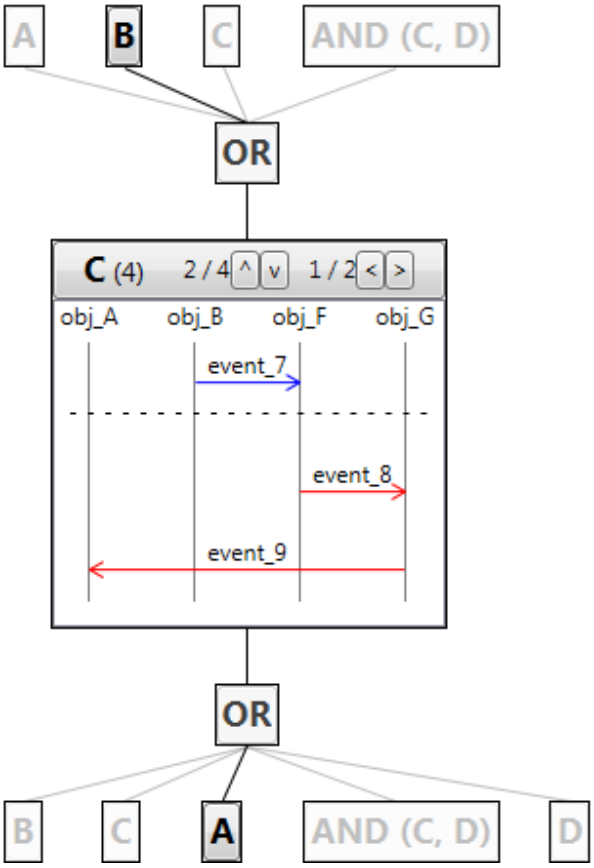


Figure 26. Button for different kind of traversing the scenario graph

5.4. Handling AND nodes

AND nodes represent situations when several scenarios overlap on one or more events. Since some events are common for overlapping scenarios, we can conclude that such scenarios are logically connected. Therefore, there is a high probability that they share same objects. This is a good motivation to have an ability to see LSCs overlapping at an AND node not individually, but combined at one diagram. In principle, not only overlapping scenarios may be logically connected and have many same objects, but we restrict out tool so it can only combine overlapping scenarios. Motivation of such a decision is twofold: first, it prevents users from generating large meaningless diagrams where combined LSCs have no logical connections between each other; second, the chosen approach consists in studying a scenario graph and analyzing the system by zooming in certain parts with the help of a detailed view. For this reason, we always show LSCs, which correspond to nodes from the scenario graph. If we allow the users to combine two any scenarios, the resulting diagram will not relate to any node from the scenario graph that is against the chosen approach.

At the detailed view, we could display only those events that are included in the corresponding AND node. *Figure 27* illustrates the result of this design decision for node 23 from the scenario graph shown in *Figure 19*. Scenarios C and D overlap by event event_8, so the combined diagram consists only of this event and two objects that communicate by this event.

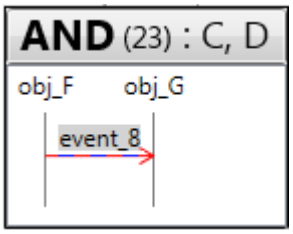


Figure 27. Visualization of the AND node with only overlapping events included

However, as well as in case with scenario nodes, such a solution omits the context information, which is of a big interest: since there is a logical connection between overlapping scenarios, the user wants to see the whole flow of events that form those scenarios. That is why we create a combined LSC by placing at one diagram all events and objects that form LSCs that overlap at an AND node. In case an event or an object is included in several scenarios, we do not duplicate it at a combined LSC. The order of how events appear at a combined diagram coincides with their occurrence sequence found in the execution tree. At a combined LSC, the user should be able to distinguish events by which scenarios overlap. To achieve it, we fill backgrounds of such events with grey color. We also keep the coloring information for events: events that form pre-charts remain blue, while events that are included in main-charts are red. However, when combining several scenarios, some events may be a part of one scenario's pre-chart and a part of another scenario's main chart at the same time. For such events, we draw an arrow, which is colored in a dashed manner where red and blue segments alternate. *Figure 28* displays the final result for the detailed view of node 23. Note that execution of scenario C began earlier than execution of scenario D, so its pre-chart event goes first at the combined LSC.

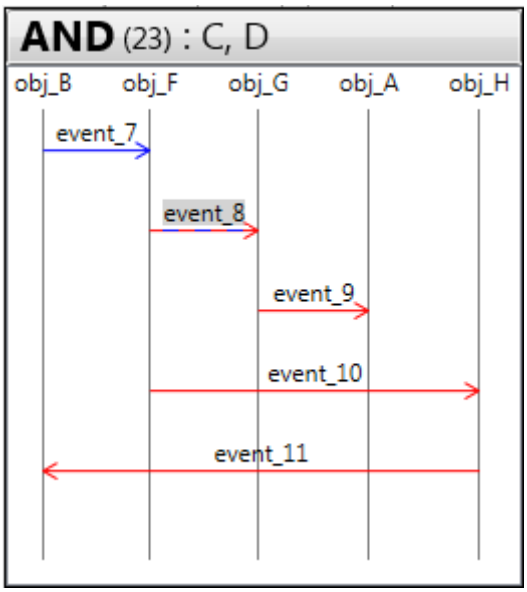


Figure 28. Final visualization of the AND node

When analyzing a combined LSC, the user need a feature for identifying to which LSCs some event belongs. To meet this requirement, we add tooltips that are enabled by right-clicking at events names. Such tooltips contain a list of scenarios that include the corresponding event. In *Figure 29* we can see that event_9 occurs only in scenario C, while event_8 is the part of both scenarios C and D.

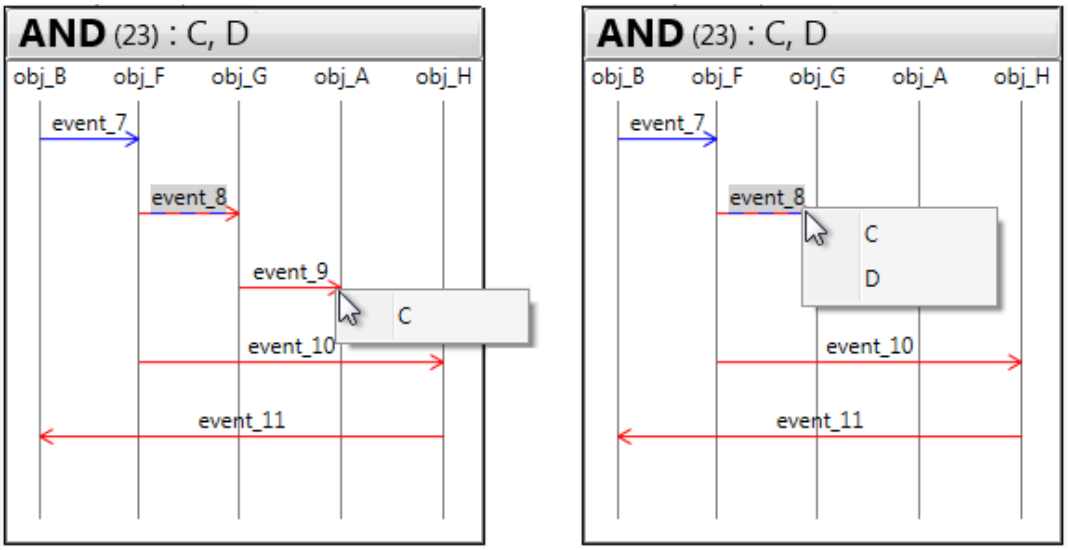


Figure 29. Tooltips for finding out to which scenarios an event belongs

The more overlapping scenarios an AND node contains, the bigger the combined LSC becomes. When studying inter-relations between scenarios, the user might want to filter the list of scenarios that form a combined LSC, thereby concentrating at some of them. We provide the user with such an opportunity by allowing choosing the "active" scenarios. First, the user right clicks at some event thereby showing the tooltip. Then, it is possible to select only those LSCs the user is interested in. After that, all the events that are not included in the selected LSCs are faded out. *Figure 30* shows how this approach works. There, only scenario D is selected, so events belonging to scenario C are faded out. At this, event_8 is not faded out since both scenarios C and D overlap on this event and at least one of them is selected.

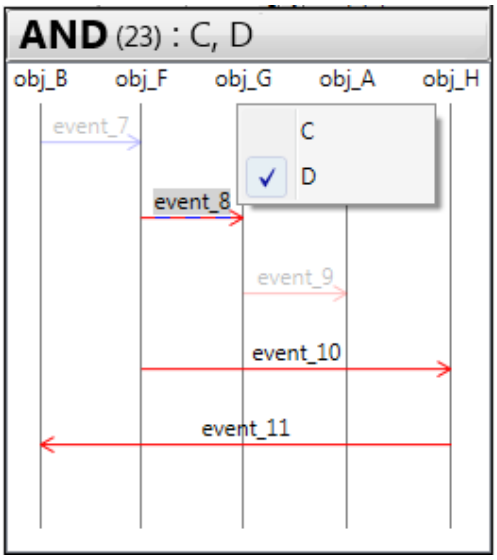


Figure 30. The AND node with scenario C faded out

6. Filtering

In Chapter 3 we defined a number of use cases that help the user to concentrate on the most popular scenarios that occur in the system. Another use cases deal with the situation when the user wants to select the set of objects to see in which scenarios the selected objects take part. To help the user to accomplish such tasks, we present the mechanism, which allows filtering out scenarios from the scenario graph and objects from individual scenarios.

The scenario graph depicted in *Figure 31* will be used throughout this chapter. Scenarios A, B, C and D are the same that are used in Chapter 5, while the scenario graph is a different one. We can see that scenario A occurs three times, scenario B occurs four times, scenario C occurs two times, and scenario D occurs only once.

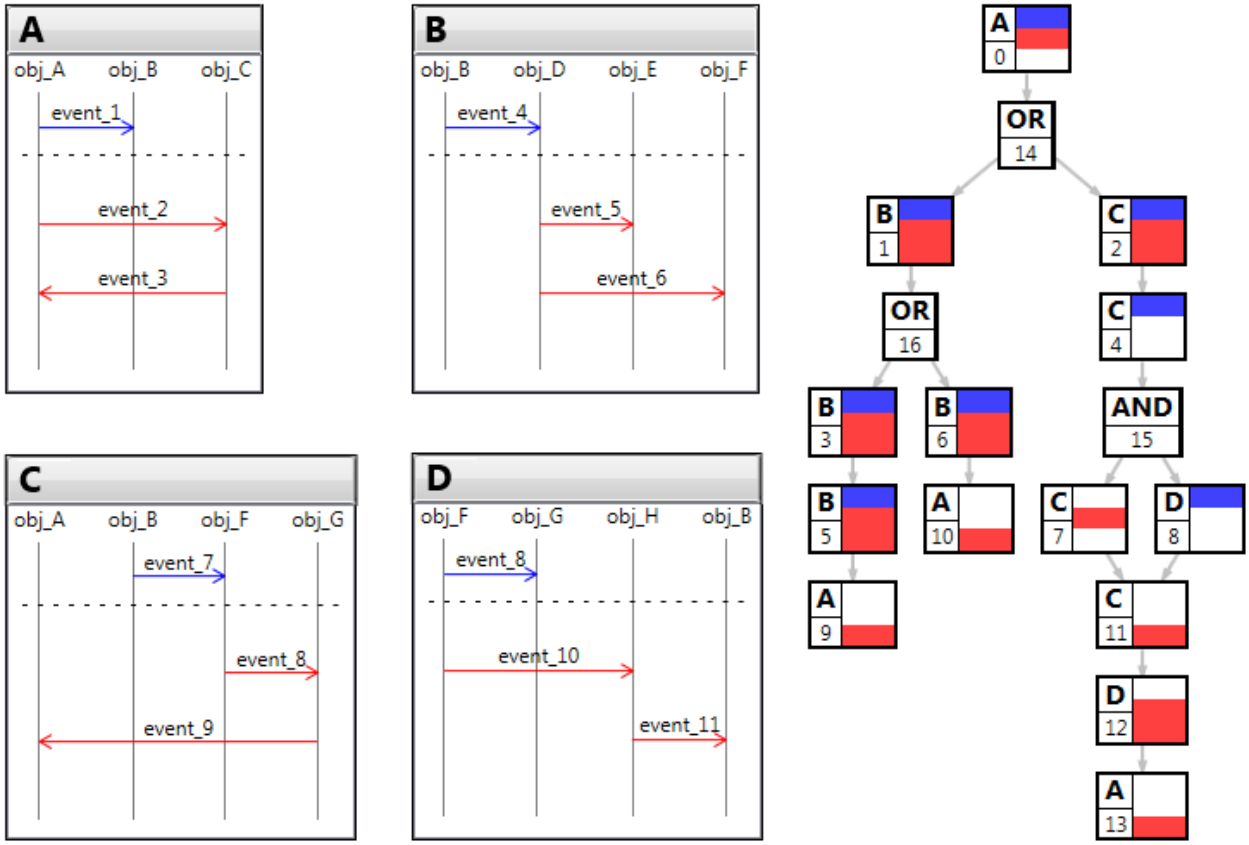


Figure 31. LSCs of scenarios A, B, C, D and the scenario graph used in Chapter 5

6.1. Scenario-based filtering

By looking at *Figure 31* we can quickly say which scenarios occur more often than the others. For instance, it is clear that scenario B occur multiple times while scenario D occurs only once. However, scenario graphs for real-life systems are a lot bigger than the one we use as the example. Therefore, they cannot fit into one screen, and the user should drag the canvas to observe the entire graph. For this reason, it can be difficult to say which scenarios occur most often. To solve this problem, we add a table, which contains statistic information of scenarios usage. We add to this table the occurrence number for every scenario, so that the user can distinguish between often and seldom occurring scenarios. We also depict in this table support

and confidence parameter values for each scenario. So, such a table consists of the following columns:

- O – shows how many times the corresponding scenario occurred;
- S – shows the value of the support parameter for the corresponding scenario;
- C – shows the value of the confidence parameter for the corresponding scenario.

For the given example, such a table looks like depicted in *Figure 32*, where the table is sorted descendant by column O. The table can also be sorted by any other column.

Name	O	S	C	F
B	4	1	1	☑
A	3	3	1	☑
C	2	1	1	☑
D	1	1	1	☑

Figure 32. The table for scenario-based filtering

Scenario D occurs once, so it is probably the least important scenario for the user. To filter it out, the user should tick a checkbox next to this scenario in the table. Doing so results in replacing all nodes that corresponds to scenario D with small dot nodes. We do not just throw away such nodes because it may mislead the user since he cannot see anymore that something happens in the system while it does. At the same time, dot nodes indicate that some scenarios occur but we do not see them because they have been filtered out. The scenario graph after disabling scenario D is shown in *Figure 33*.

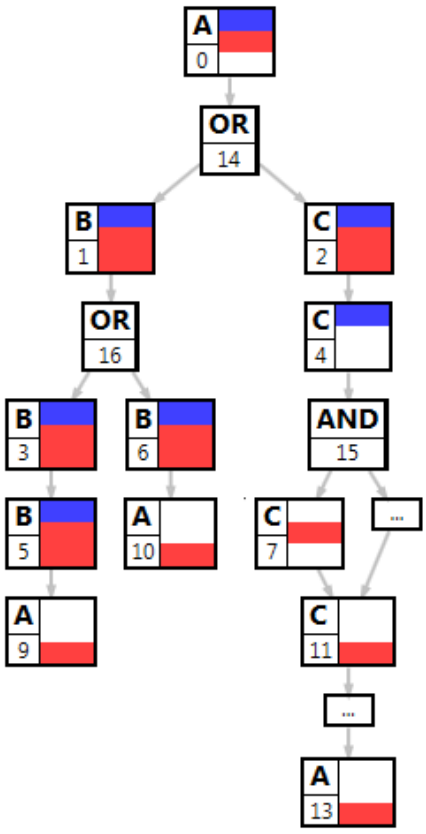


Figure 33. The scenario graph after filtering out scenario D

When there are a lot of scenarios in the system, it may become annoying to first deselect a lot of checkboxes and select them again to see the initial scenario graph. The latter part of the problem (i.e. returning back to the initial graph) can be solved by presenting a simple resetting mechanism, so that when the user presses a button to clear all currently applied filters. However, this resetting mechanism apparently does not help the user to add a range of scenarios to the filter. To design a proper solution for the problem of selecting multiple scenarios at once, we first list a number of requirements for this solution:

- The user should have an option to select a range of scenarios, which is quicker than using checkboxes in the table.
- The user should be able to indicate the parameter upon which the scenario filtering is based: either occurrence, or support, or confidence.
- The user should be able to indicate a number of scenarios he wants to keep at the scenario graph. This number can be either a relative one (a percentage of the total number of scenarios) or an absolute one (an exact amount of scenarios).
- The user should have an option to return back to the initial scenario graph (i.e. when no filters are applied to it), which is quicker than using checkboxes in the table.

Our implementation of the idea for selecting several scenarios at once is shown in *Figure 34*. By using this approach, the user can for example easily select top 50% scenarios that occur most often. To accomplish it, the user should enter "50" in the textbox, select percents in the first dropdown list, select descending order in the second dropdown list, and select O in the third dropdown list. Such a choice indicates that the user wants to select only 50% of all scenarios from the list, which is sorted in a descendant order by column that represents scenarios occurrences number. After selecting all the needed values, the user should click Select button. Then, to return to the initial scenario graph, the user may enter the value 100 in the textbox, make sure that percent is chosen in the first dropdown menu, and click Select button again. Using such a mechanism, it is possible to filter scenarios basing not only upon their occurrences count, but also upon the values of support and confidence parameters. Also, by changing the value of the first dropdown list to "psc.", it is possible to indicate an absolute number of scenarios to be filtered instead of a relative one.

Although the scenario selection mechanism meets all the requirements that have been formulated for it, it has one drawback: using this mechanism, it is possible to select only top scenarios from the table. So, if the table is sorted by Occurrence column, we can select either scenarios that occur most often (in case the table is sorted in descending order) or scenarios having the rarest occurrence number (in case the table is sorted in ascending order). Thus, using our implementation of the scenario selection mechanism, it is not possible to add to the filter those scenarios that occur at least n times and not more than m times. However, we do not need an opportunity to select a middle range of scenarios in our use cases. That is why improving the scenario selection mechanism has been left for the future work.

Figure 34. Scenario selection mechanism

The filtering mechanism described above just replaces nodes that correspond to scenarios that have been filtered out with dot nodes. So, if the user filters out scenarios C and D, the scenario graph now looks as shown in *Figure 35*.

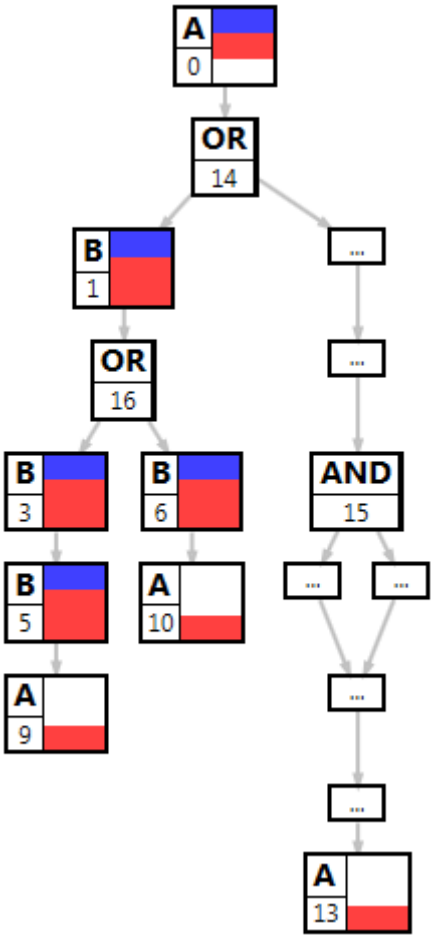


Figure 35. The scenario graph after filtering out scenarios C and D

By inspecting *Figure 35*, we can see that there are two cases when we should change a scenario graph. The first one deals with the situation when several dot nodes form a sequence. Although it is important for the user to see that in case something is filtered out, it still happens in the system, multiple dot nodes following one another do not add any valuable information. That is why we merge them into one node. The second case concerns AND nodes. It might happen that two or more child nodes of an AND node were replaced with dot nodes after applying a filter. In this case, we merge such nodes. If all the child nodes of an AND node became dot nodes, we replace this AND node and all its child nodes with a single dot node because such an AND node

is now meaningless. Making these changes to the scenario graph from *Figure 35* leads to the result depicted in *Figure 36*.

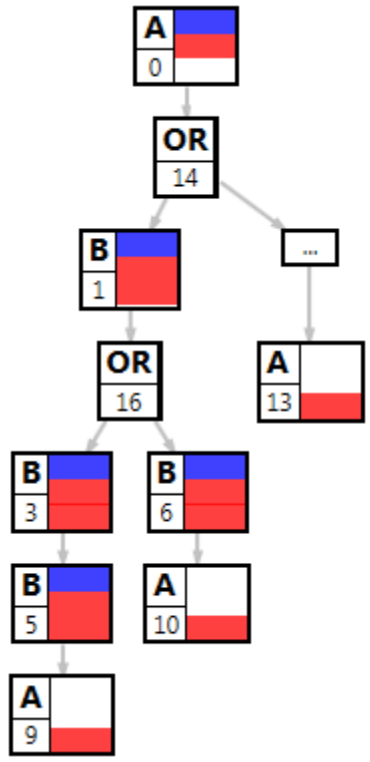


Figure 36. The scenario graph after removing redundant dot and AND nodes

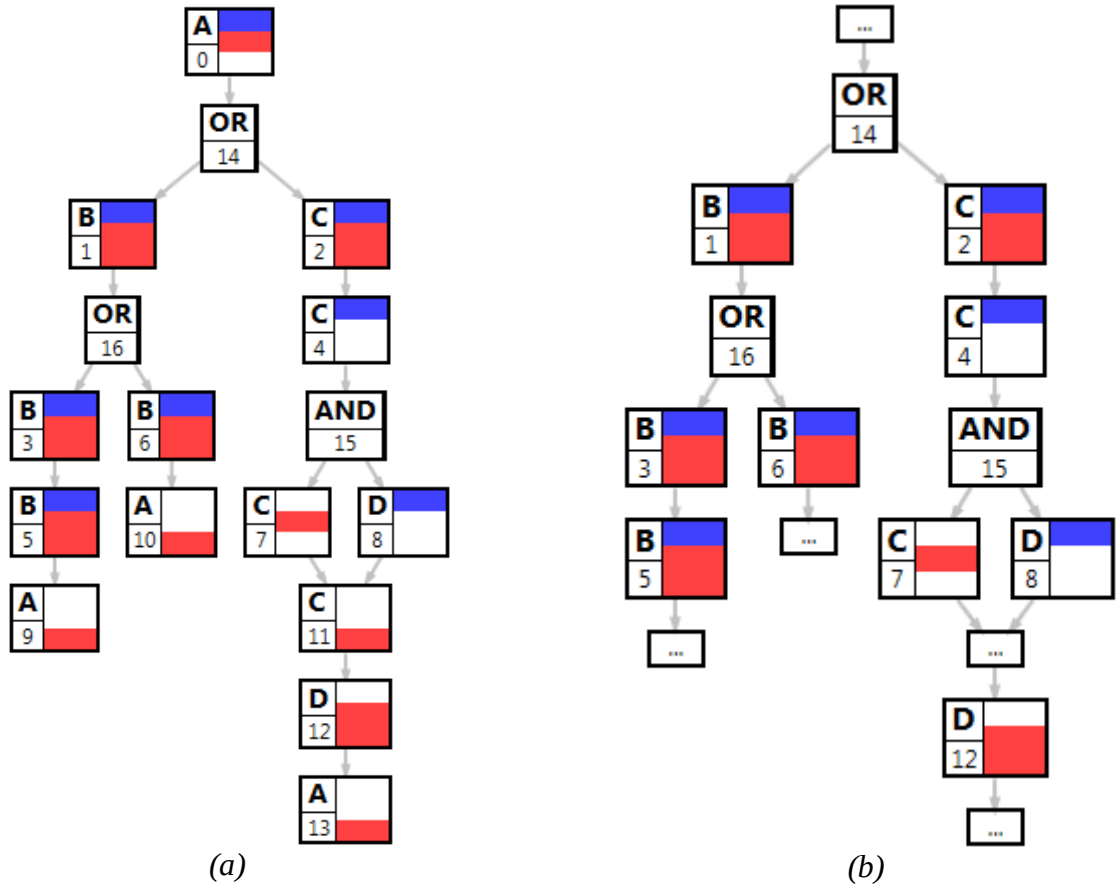
By comparing the obtained scenario graph (*Figure 36*) with the initial one (*Figure 31*) we can conclude that: (a) it is now clear, which scenarios occur most often (those are scenarios A and B); (b) it is easier to concentrate at these two scenarios because the scenario graph is now smaller than the initial one since several nodes that represent scenarios of rare occurrence are filtered out.

6.2. Object-based filtering

Besides scenarios, filtering may be also applied to objects. Unlike the scenario-based filtering, filtering objects may be done in two different ways. The first one concerns the detailed view. The size of an LSC displayed in the detailed view may be large, so the diagram cannot be observed and analyzed without using quite a lot of scrolling (both vertical and horizontal). Thus, the user may want to hide some objects in the detailed view to concentrate on particular objects and to reduce the size of the diagram. Since such filtering affects only currently expanded LSC, we call it local object-based filtering. The other object-based filtering mode is similar to scenario-based filtering in terms that it affects the scenario graph: we collapse those nodes whose corresponding LSCs do not have events that are not hidden after applying the filter. Such filtering mode is called global object-based *filtering* because it affects not only currently expanded LCS, but the whole scenario graph.

Global object-based filtering

The idea behind global object-based filtering is collapsing those nodes in the scenario graph that do not have visible events left after applying the filter. *Figure 37* shows the comparison between the initial scenario graph (*Figure 37a*) and the scenario graph obtained after obj_A is added to the global filter (*Figure 37b*).



(a) *(b)*

Figure 37. The initial scenario graph and the scenario graph after filtering out object obj_A

To understand why we got such a result, let us take a closer look at nodes that were filtered out. There are five such nodes, but they represent only two scenarios: two fragments of scenario A and a fragment of scenario C. LSCs for those fragments are shown in *Figure 38*. If we filter out obj_A, all the events that use this object should be also filtered out. Those are: event_1, event_2, event_3 and event_9. We can see that excluding these events from LSCs depicted *Figure 38* leads to empty LSCs since there are no events in those LSCs that do not use obj_A. For this reason, we replace these nodes with dot ones.

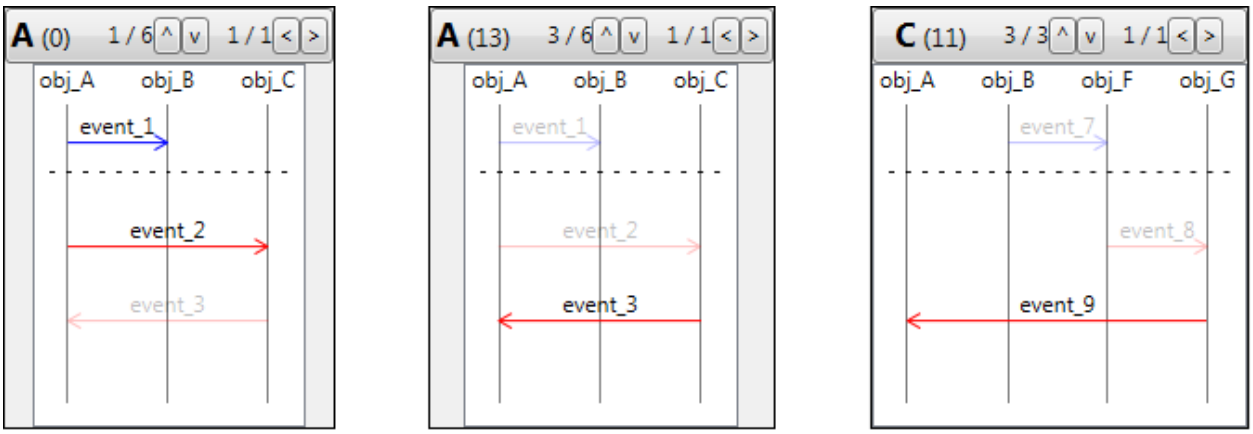
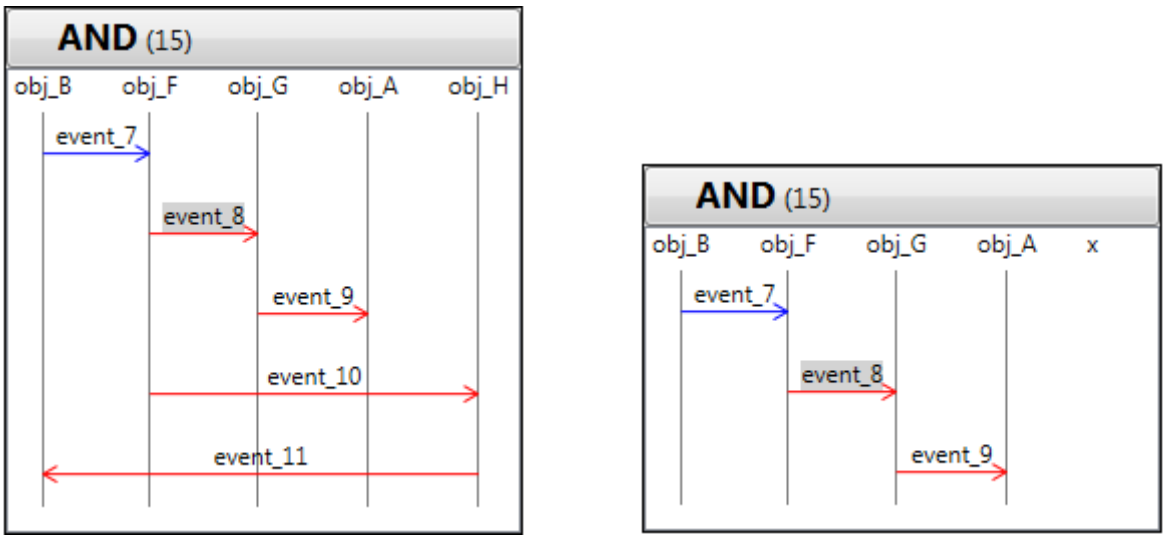


Figure 38. LSCs for nodes 0, 13, 11

Local object-based filtering

Some LSCs (especially LSCs for AND nodes) contain a large number of objects, so it is difficult to analyze them as a whole because their width exceeds one screen. For this reason the user might want to concentrate at particular objects at one time so that he can see lifelines of all the chosen objects at once. To have this feature, we allow the user to hide the selected objects inside the detailed view that will lead to reducing the size of the LSC. In this use case, only currently expanded LSC is affected, i.e. hiding an object in one scenario does not lead to hiding the same object in other scenarios.

We illustrate objects hiding at the following example. *Figure 39a* shows the LSC for node 15, which is an AND node where scenarios C and D overlap by event_8. This event is sent by obj_F and received by obj_G, so let us assume that the user wants to filter out all objects that do not directly communicate with either obj_F or obj_G. Only one object satisfies this requirement, which is object obj_H. After adding obj_H to the local filter we get the result depicted in *Figure 39b*. When obj_H is filtered out, all the events that use this object are also removed from the LSC. That is why we do not see events event_10 and event_11 anymore. At this, objects that are filtered out are represented by a cross sign.



(a) *(b)*

Figure 39. The LSC before including obj_H into the local object-based filter and after that

When the user has studied the behavior of objects that are not hidden, he might want to bring back some objects to continue examining the LSC. However, it is difficult to remember which cross sign corresponds to the object the user wants to return. To solve this problem, we add tooltips to hidden objects as shown in *Figure 40*.

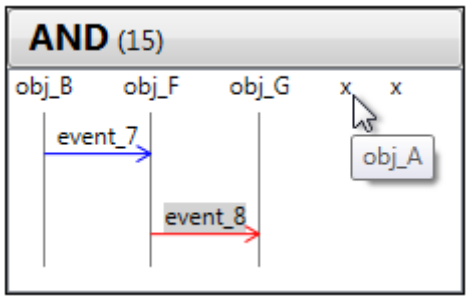


Figure 40. The tooltip for hidden object obj_A

Hiding and revealing an object can be done either from the context menu in the detailed view or by using a table similar to one used in scenario-based filtering. Such a table has as many rows as the number of objects presented in the logs.

The user might want to choose a set of objects, so in the scenario graph only those scenarios will be shown that uses one or more selected objects. To accomplish this task, we introduce a table similar to the one used for scenario-based filtering, which shows the statistics of objects usage and allows the user to filter them (see *Figure 41*).

Name	S	R	T	GF	LF
obj_A	2	3	5	☑	☑
obj_B	3	3	6	☑	☑
obj_C	1	1	2	☑	☑
obj_D	2	1	3	☑	☑
obj_E	0	1	1	☑	☑
obj_F	5	3	8	☑	☑
obj_G	2	3	5	☑	☑
obj_H	2	2	4	☑	☑

Figure 41. The table showing the information about objects usage and their presence in filters

- S – shows how many times the corresponding object is used as a sender;
- R – shows how many times the corresponding object is used as a receiver;
- T – shows how many times the corresponding object is used in total;
- GF – indicates if the corresponding object is used in the global filter;
- LF – indicates if the corresponding object is used in a local filter.

The purpose of adding column T is straightforward: using this column, the user can tell objects, which are used a lot from objects, which are rarely used. Adding columns S and R to the table allows to better understand the role of an object in the system. For instance, if some object is used a lot as a receiver, it might represent a utility or a helper class.

To help the user to add (remove) several scenarios at once to (from) the filter, we implement scenario selection mechanism, which is similar to one that was used for scenario-based filtering. It is shown in *Figure 42*.

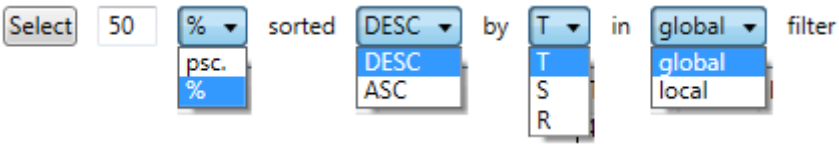


Figure 42. Filtering mechanism for object-based filtering

7. Generalization

After applying all the transformation discussed in the previous chapters to the initial execution tree, the resulting scenario graph is also a tree, which is not good for analysis purposes for several reasons. First, once two branches are created in a tree, they will not flow together regardless of the fact that their endings are the same (that is a common pattern for many systems). At this, if several nodes represent the same scenario or scenario fragment and have equal possible continuations, it is redundant for the user to see them all in the scenario graph and such nodes can be folded, therefore decreasing the size of the scenario graph even further. Second, there can be places in a system where a continuation should be chosen (typically by the user) out of several options. And after the selected scenario is executed, the system returns in the state where the same choice is possible again. So, loops are found in the system and it can be quite difficult to detect them by looking at the tree. To help the user in analyzing the system and to solve the mentioned problems, we decided to apply generalization to the scenario graph.

Consider the execution tree and the corresponding scenario graph shown in *Figure 43*. There are two notable things about this scenario graph: (1) every branch ends with the same fragment of scenario A; (2) scenarios B and C are wrapped by scenario A in such a way that a fragment of scenario A occurs first, following by occurrence of either scenario B or C. After that, it is possible either to execute the occurred scenario once again or to terminate with a segment of scenario A.

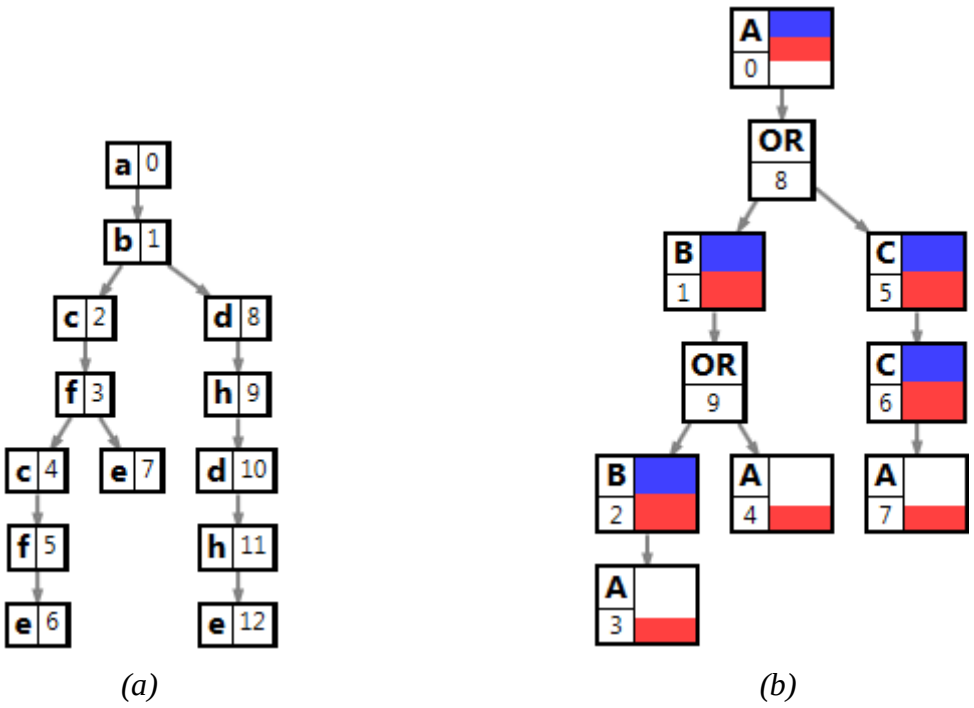


Figure 43. The execution tree and the corresponding scenario graph

To avoid problems with generalization a scenario graph that contains AND/OR nodes, we use not the final version of such a graph, but rather a version, which we get right before inserting AND/OR nodes (see *Figure 44*). In this version, we see which events have been incorporated in which node and which scenarios correspond to these events. For instance, node 2 corresponds to events c and f, which are an occurrence of scenario B. Since at this step one node may

correspond to multiple scenarios (in case when several scenarios overlap on some events), there are no progress trackers attached to nodes yet.

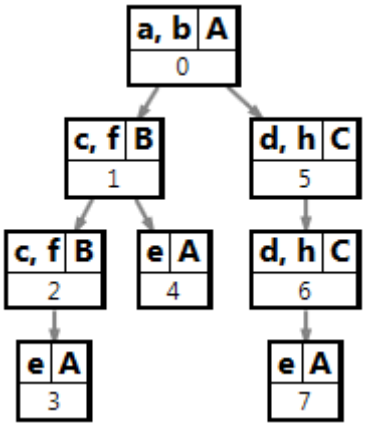
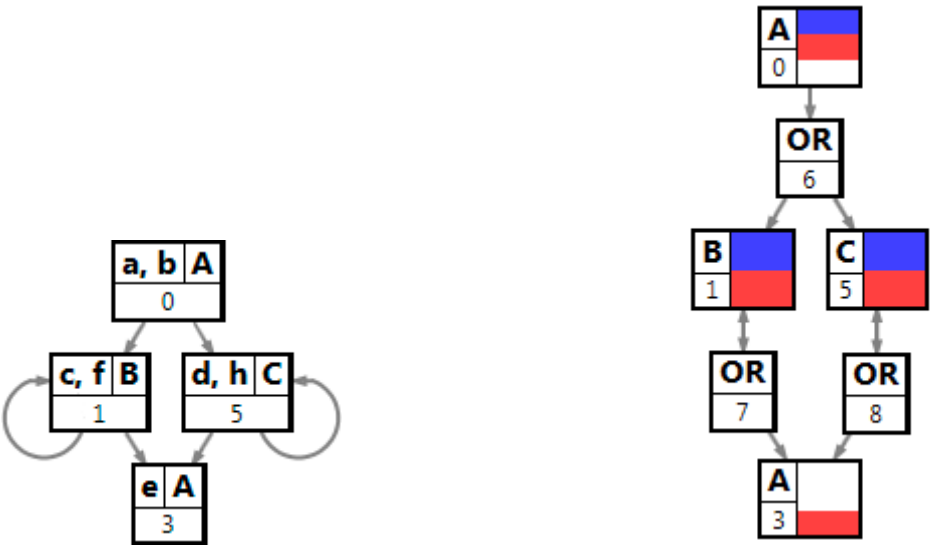


Figure 44. The raw version of the scenario graph

The first idea concerning scenario graph generalization was the following one: for every scenario or scenario fragment leave in the scenario graph only one node that corresponds to that scenario or scenario fragment, thereby folding all its occurrences into one node. Applying such a transformation to the example results in the scenario graph depicted in *Figure 45*.



(a) raw version *(b) final version*

Figure 45. Generalized scenario graph

By comparing this result to the scenario graph depicted in *Figure 43b*, we can conclude that it is now easier to see that every trace in the system ends with a segment of scenario A and that scenarios B and C are wrapped by scenario A. However, after applying generalization we added some behavior that could not be found in the initial system and we cannot retrieve some information any longer. It leads to the following problems.

In the initial graph scenarios B and C occurred two times, while in the transformed version we see just a loop without knowing how many times scenarios, which are a part of this loop may occur. For those systems that work in compliance with such a logic that does not constrain the

number of scenario occurrences, it is good to have a generalization described above because logs basing upon which we construct scenario graphs are almost always incomplete. So, for such systems having directly connected nodes that correspond to the same scenario is redundant and they can be replaced with a loop. However, for other systems the number of some scenario occurrences may be of significant importance. For instance, in the system, which scenario graph is shown in *Figure 43b*, the amount of scenarios B and C occurrences may be limited to 2 by a business rule. Loops that are seen in *Figure 45* do not reflect this information and therefore can be misleading.

We can see that in the generalized scenario graph it is possible to execute the second fragment of scenario A and thereby terminate right after executing scenario C once at node 5, although it is not an option in the initial graph where scenario C occurs twice before termination becomes possible. On the one hand, such behavior may appear due to the incompleteness of the logs, i.e. in the system there is an option to terminate after executing only one instance of scenario C and the reason why we do not see it in the scenario graph deals only with the fact that there is no corresponding trace in the logs because they are incomplete. In this case, the generalization is a good one in that sense that it adds the behavior that can be found in the system. On the other hand, the fact that scenario C occurs twice before the termination becomes possible may arise because of actual system specification. In this case, incorporating two instances of scenario C in one node with a self-loop may mislead the user.

Due to the mentioned problems, we decided to change the generalizing procedure and make it more flexible, so the user can decide himself how far he would like to generalize the scenario graph. We base our new generalization procedure upon equal possible behavior at nodes, i.e. we fold two nodes if they have the same possible continuations. To search for such nodes, we first divide all the nodes of the scenario graph into groups in such a way that nodes belong to one group if they are located at the same distance from a leaf, which is the necessary condition for folding nodes because only such nodes may have the same successors. Then, within each group we look for two nodes that have the same possible continuations and fold them. This procedure is repeated until there are no nodes that satisfy the folding conditions.

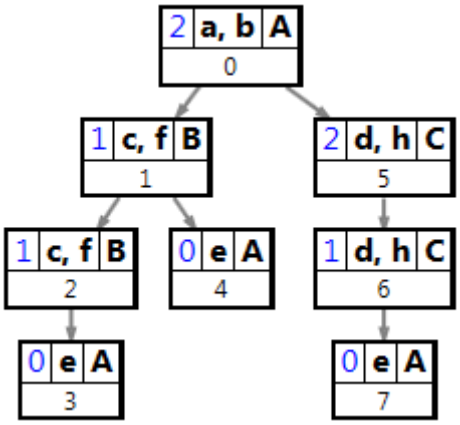
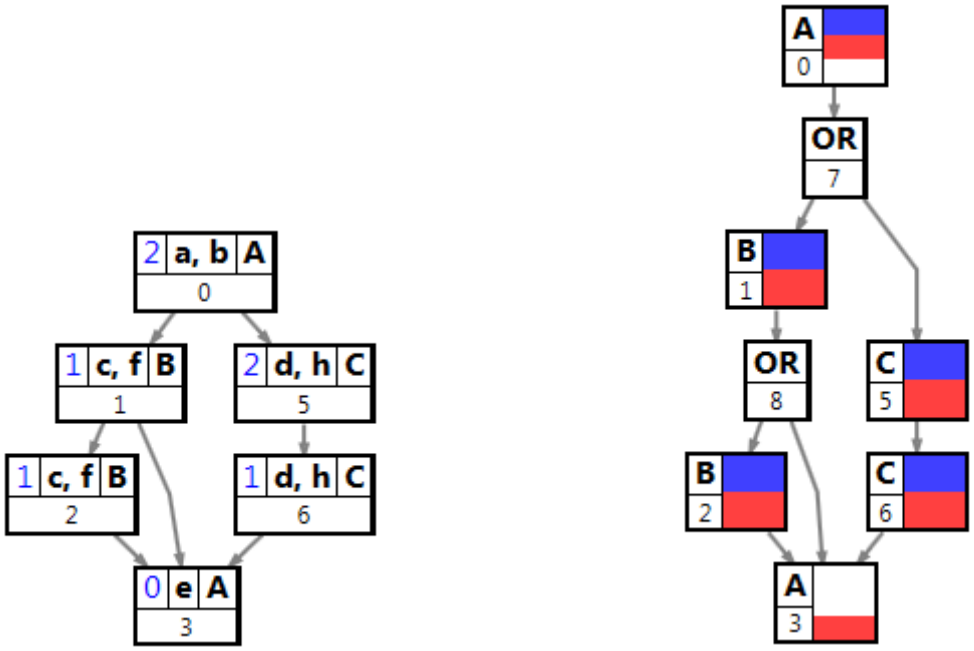


Figure 46. The raw version of the scenario graph with minimal distances to a leaf

Figure 46 shows the example scenario graph with computed minimal distance from a leaf for every node. All the nodes are divided into groups basing upon this number, so there are three groups: nodes 3, 4, 7 (they are leaves and therefore their minimal distance to a leaf is 0), nodes 1, 2, 6 (minimal distance to a leaf is 1), nodes 0, 5 (minimal distance to a leaf is 2). Now, every such a group is checked for existing nodes that have equal successors. This procedure starts with the first group, which contains only leaves. In this example, all leaves are events e. Leaves do not have any continuations by definition, so all the leaves can be folded. The result is depicted in *Figure 47*. We can now notice more easily that every trace in the system starts with one fragment of scenario A and ends with the other fragment of this scenario. We can achieve the same using our first generalization idea, but unlike that generalization shown in *Figure 45*, we now did not lose the details that concern execution of scenarios B and C.

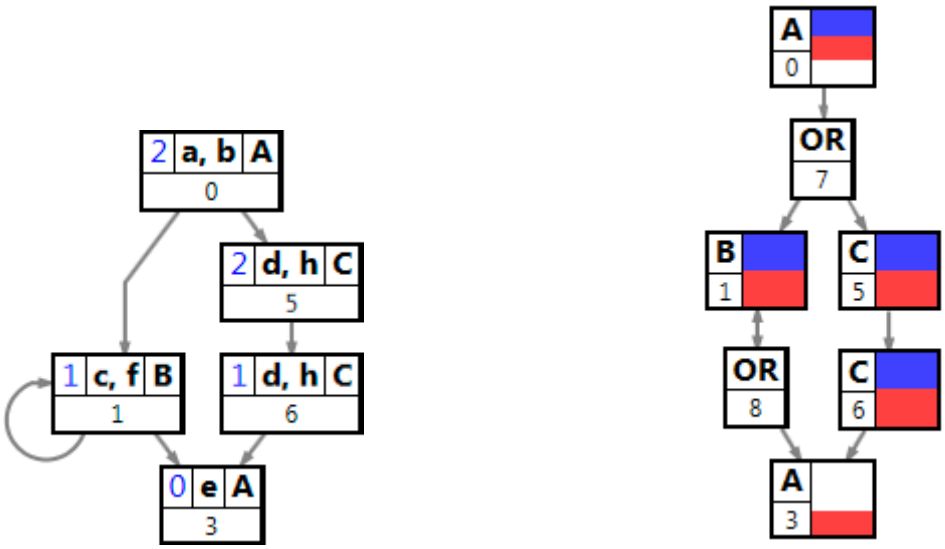


(a) raw version *(b) final version*

Figure 47. The scenario graph with generalization rate = 1

To generalize the graph further, we consider the scenario graph obtained after performing the previous generalization step and the next group of nodes where minimal distance to a leaf equals 1. Those nodes are: 1, 2, 6. Among them there is only one node that corresponds to the set of events {d; h}, so this node will not be folded with any node. Two other nodes are more interesting in terms of generalization: they both correspond to the same set of events: {c; f}. There is also an option to terminate right after executing the scenario that corresponds to each of these nodes. However, after node 1 there is also an option to continue with node 2. So, excluding the edge that directly connects nodes 1 and 2, these nodes satisfy our folding conditions. Taking into account that at any of these nodes there are options either to repeat the same scenario or to make a choice between equal continuations, we may conclude that multiple sequential occurrences of scenario B appear not because of some business rule, but due to the nature of logs, which contain all the travelled traces. Therefore, such nodes can be replaced with a self-loop (see *Figure 48*). It may seem that nodes 5 and 6 are of the same kind as nodes 1 and 2, but although they both correspond to scenario C, there is no option to terminate after node 5, while

there is such an option after node 6. So, these nodes do not satisfy the folding conditions and we do not fold them.



(a) raw version *(b) final version*

Figure 48. The scenario graph with generalization rate = 2

The next step is to search for nodes that can be folded within next group, which contains nodes whose minimal distance to a leaf equals 2. There are only two of such nodes (nodes 0 and 5) and they correspond to different events. Therefore, these nodes cannot be folded and at this generalization step no folding occurred.

If the information from the logs is incomplete, we are unlikely to get the fully generalized scenario graph (such as depicted in *Figure 45*). However, such a generalization is still valuable since it allows the user to immediately see all the possible connections between scenarios and the possible paths to particular scenarios. For this reason, if our generalization procedure fails to produce a generalized scenario graph that contains only one instance of each scenario and scenario fragment, we compute such a version and add the result to the set of generalized versions of the scenario graph.

The user can control the generalization rate for the scenario graph by using a slider, which allows switching between items that belong to the set of generalization. The most left position of this slider indicates "non-generalized" state (i.e. the initial scenario graph), while the most right position indicates "fully generalized" state (i.e. the generalization, which contains only one instance of every scenario and scenario fragment). Between these two edge cases there are a number of states that correspond to generalizations obtained by sequentially applying generalization procedure to the scenario graph to fold nodes that are located at the same distance from a leaf. So, the total number of possible generalizations equals $n + 2$, where n is the farthermost distance from a leaf a node in the graph has; and 2 stands for the initial and fully generalized versions of the scenario graph. The total number of generalized versions can be smaller than the maximal one in case when at some sequential iterations of there are no nodes that satisfy folding conditions or when the fully generalized version can be obtained by using iterative generalization procedure.

Using generalization provided several benefits. First, applying it allows getting graphs that are smaller than initial ones because equal parts are folded. Second, the output of the generalization procedure is a graph, while its input is a tree, which is a good thing since graphs may contains loops and trees cannot. Finally, the introduced configurable generalization helps not to lose the important system behavior by allowing the user to choose how far he would like to generalize the scenario graph basing upon the minimal distance to a leaf.

8. Implementation and evaluation

In this chapter, we first describe details concerning the tool, which is the implementation of our approach explained in the previous chapters. Then, we make this tool evaluation using the scenarios mined from logs of the application that implements FTP protocol.

8.1. Implementation

The suggested approach has been incorporated into the tool whose screenshot is shown in *Figure 49*.

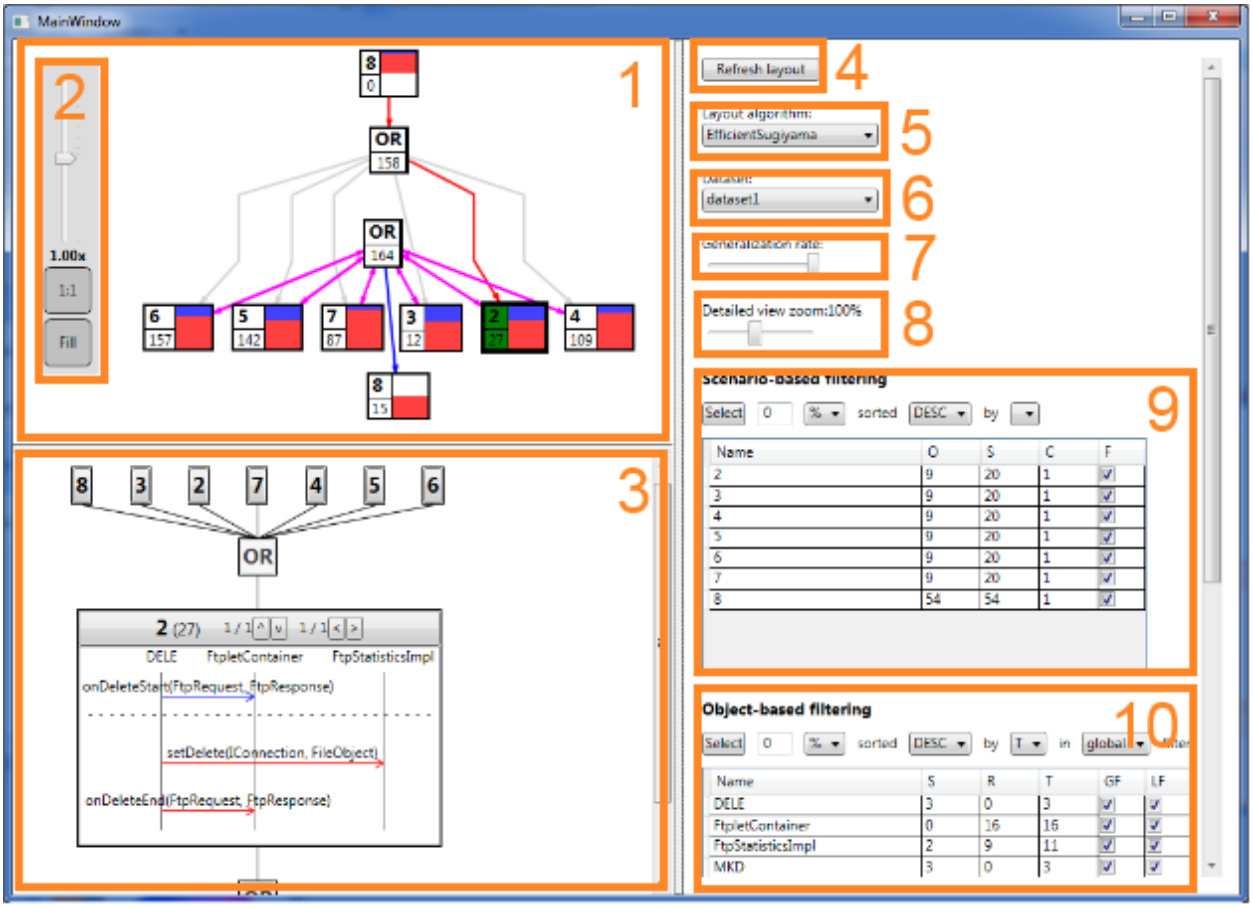


Figure 49. Screenshot of the tool with enumerated areas of interest

The whole window is divided into three main parts. Between them there are splitters that can be dragged to change the amount of space each part takes.

At the left top corner there is an area for the scenario graph (1). The user can zoom in and zoom out it by using the control (2). This control has a slider to quickly change the zooming factor and two buttons for edge positions: “1:1” button sets the zooming factor to 1, and “Fill” button sets such a zoom factor that will allow putting the whole scenario graph in the area (1).

Under the scenario graph area there is a detailed view area (3) where an LSC that corresponds to the currently selected node is shown along with some context information added.

At the right side there are controls that allow manipulating two left views. Pressing “Refresh” button (4) results in updating layout, which is useful for example after making changes to a scenario name. “Layout algorithm” dropdown menu is used to select a layout algorithm according to which nodes of the scenario graph are positioned. “Dataset” dropdown menu allows loading the data into the tool. Every dataset should be located in a separate folder, which is placed at the following path: [PATH_TO_THE_TOOL]/data/. Then, it will be found automatically when the tool is launched. “Generalization rate” slider (7) controls the generalization level of the scenario graph. Its most left position corresponds to the “non-generalized” state, while the most right position represents the “fully generalized” state. It may be difficult to analyze large LSCs because they cannot be seen as a whole without using scrolling. This problem is partly solved by “Detailed view zoom” slider (8) that is used to zoom out the whole detailed view. Scenario-based filtering can be applied using the corresponding table (9) next to which scenario selection mechanism is located. The similar table and selection mechanism is used for object-based filtering (10).

8.2. Evaluation

This subsection describes how our approach can be used to get insights of the architecture of the real application. This is accomplished at the example of CrossFTP software. Scenarios that are mined on basis of logs are used as input.

The dataset contains scenarios that have been mined using the logs for CrossFTP software, which implements FTP protocol allowing authorized users to execute different FTP commands, such as downloading, uploading, creating, deleting, and renaming files and categories. Within this subchapter, we will compare our approach with the current state of art, i.e. with the approach, which is currently used to visualize the mined specification. The output of this approach is an HTML file, which consists of images for all mined LSCs. All these images are grouped by the pre-charts of corresponding scenarios. The whole content of the HTML file can be found in Appendix A.

We start the analysis with uploading the dataset into the tool and looking at the scenario graph, which is shown in *Figure 50*.

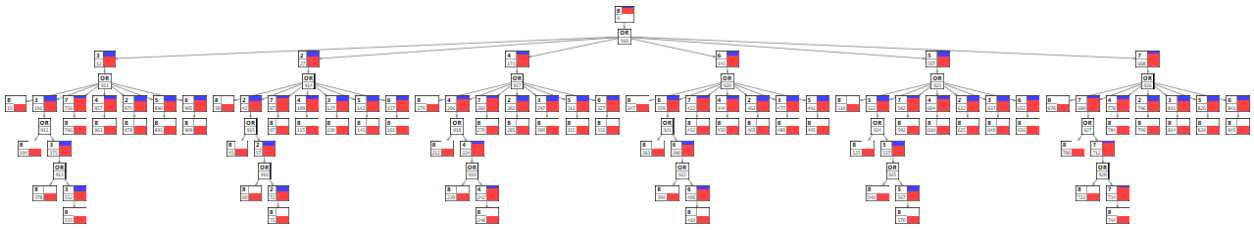


Figure 50. The scenario graph for the CrossFTP dataset with support = 20

The entire graph is too big to be readable at a page, but we still make the following observation about it basing upon its topography: there are six clusters in this scenario graph which are very similar to each other. In fact, in topological sense they are the same. However, they were not folded into one, so there should be differences, which we can reveal by zooming in two of such segments (see *Figure 51*).

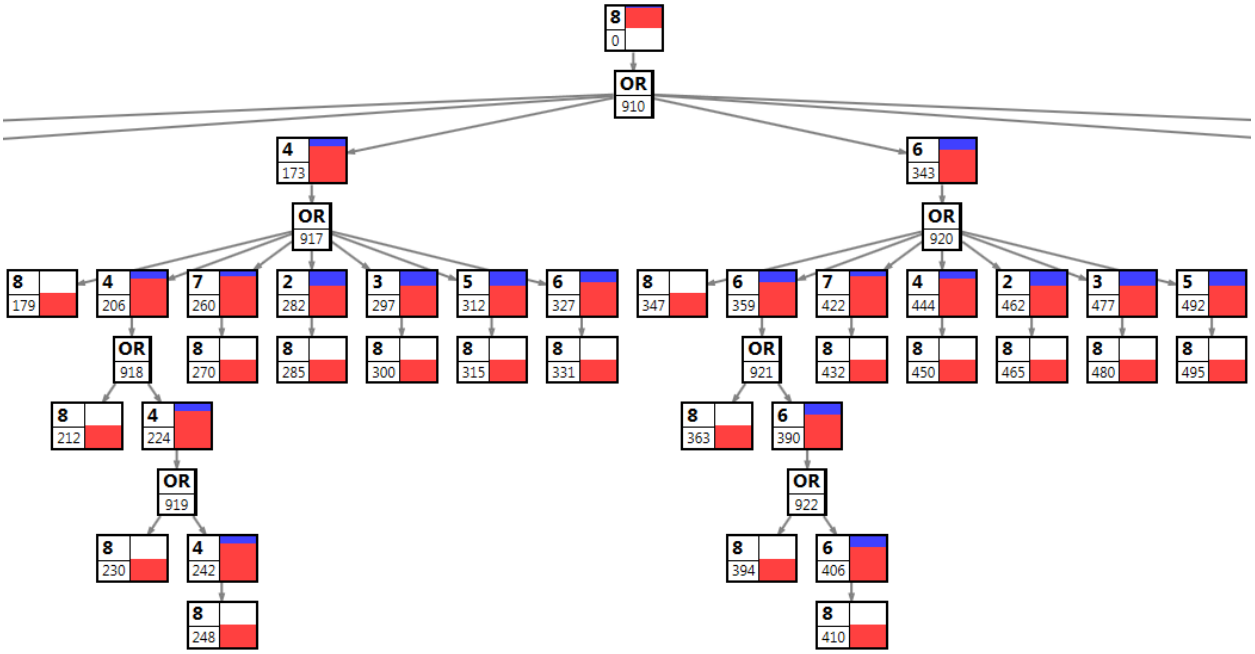


Figure 51. The zoomed part of the scenario graph

Now, we can see that after starting with scenario 8, the user have a choice between other scenarios (e.g. scenarios 4 and 6 in the figure). After the chosen scenario is finished, the user again makes a choice. However, at this step scenario 8 can be chosen, thereby terminating. Scenario 8 is an interesting one because every trace starts with this scenario and finishes with it. At this, this scenario occurs only once per trace and it is fragmented. Every trace begins with the first fragment and ends with the other one. So, scenario 8 is a wrapper for all other activities in CrossFTP. Let us get a closer look at both fragments, which are depicted in *Figure 52*.

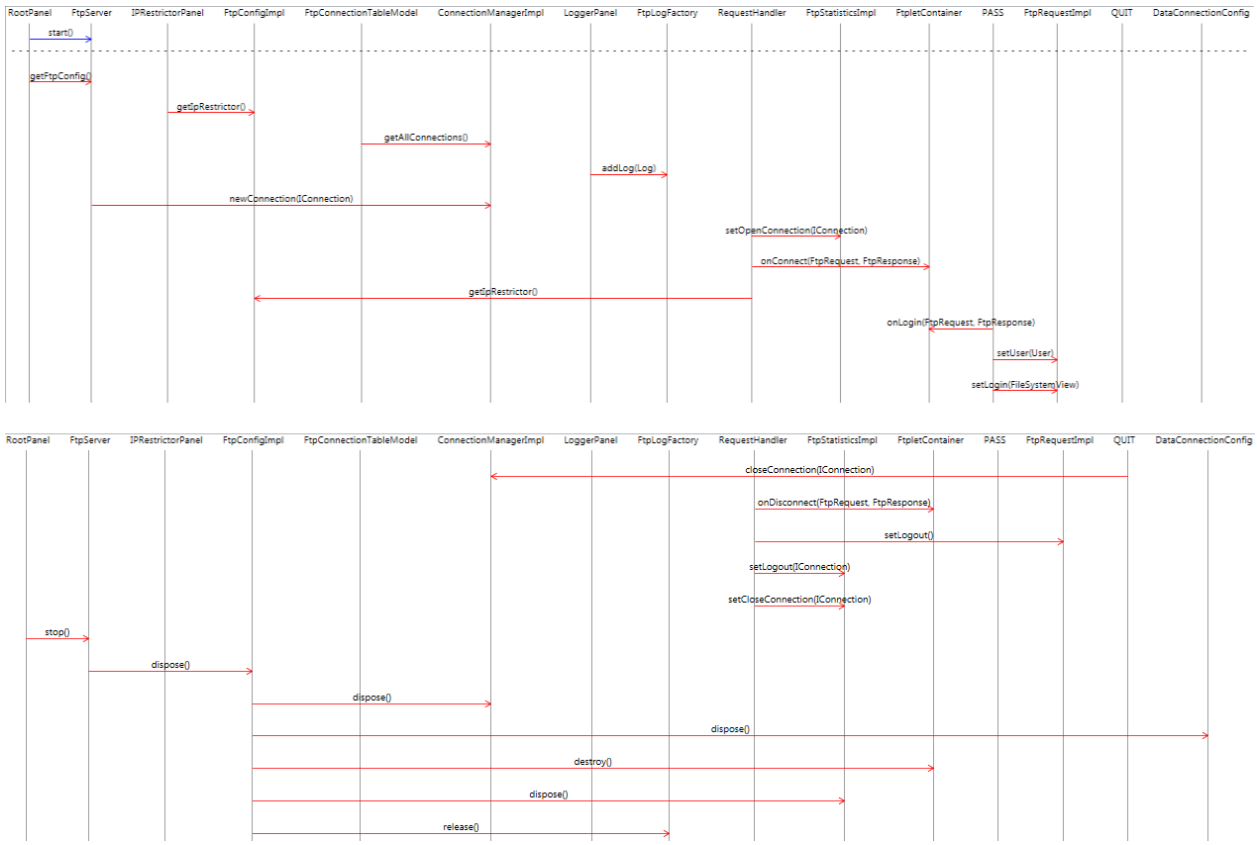


Figure 52. Two fragments of scenario 8

The first fragment of scenario 8 starts with initializing operation *start()*, then it continues with getting the IP restrictor and opening connections. This fragment ends with authorizing the user to execute commands. On the contrary, the second fragment disconnects the user from the server, closes all the opened connections and disposes objects that are not needed anymore. Basing upon the described behavior, we can conclude that scenario 8 includes operations for setting up connections, handling user authorization and terminating previously opened connections. From this point of view, it is logical that this scenario is divided into two segments that create a wrapper around all other behavior in the system: no actions are possible before the user has successfully logged in or after s/he has logged out. Thus, we assign "Login/Logout" name to scenario 8. We make sure that the second fragment of scenario 8 is always a leaf of the scenario graph (the scenario graph is now a tree because no generalization have been applied to it so far, and mapping scenarios on an execution tree results in obtaining another tree) by generalizing it to level 1, i.e. such a level when leaves are folded when they correspond to one scenario or scenario fragment. After such a generalization is applied, we get the scenario graph shown in *Figure 53*.

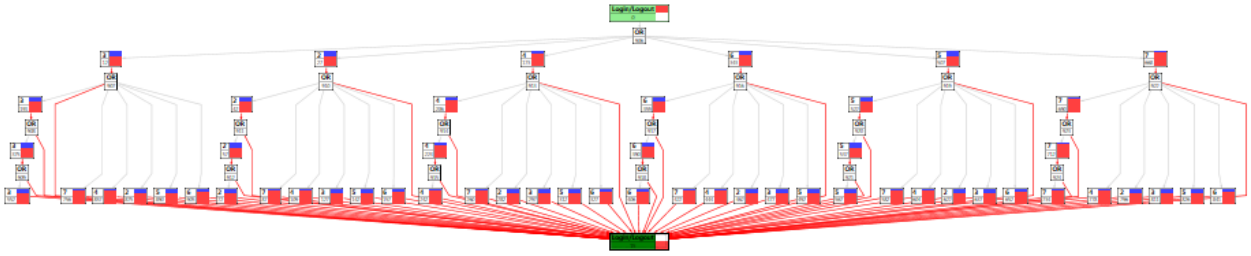


Figure 53. The scenario graph with the first generalization level applied and the logout fragment of scenario 8 selected

When the first generalization level has been applied, we can see that now only one leaf has left in the scenario graph. After selecting this leaf, all the edges incoming to it are highlighted with red. We can see that every node (except two nodes representing two fragments of scenario 8) has an edge leading to the leaf that means that after the user has logged in, at least one other scenario occurs and after that there is a choice between some other scenarios and logging out. Let us now look at other scenarios that occur in the system. Figure shows the LSC for scenario 3.

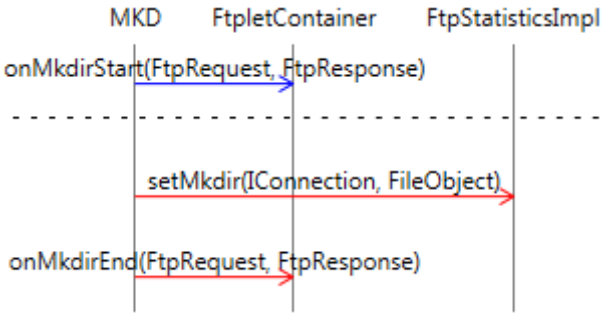


Figure 54. The LSC for scenario 3

This scenario starts with executing MKD command, which results in calling *onMkdirStart* method. This method has two arguments: a request that should be processed and response for this request. Most likely, the response variable that is sent to *FtpStatisticsImpl* object is empty when *onMkdirStart* method is invoked, and it is filled up by *FtpletContainer* object, which should construct an ftplet and thereby create a directory. After that, *MKD* object invokes *setMkdir* method, so that *FtpStatisticsImpl* object updates the statistics about how many directories have been created (probably, within the current session; that is why an object that is sent implements interface *IConnection*). Eventually, *MKD* object checks if the directory has been successfully created by calling *onMkdirEnd* function. So, we can conclude that scenario 3 corresponds to MKD FTP command, which creates a directory. Other scenarios (i.e. scenarios 2, 4, 5, 6, 7) represent other FTP commands (DELE, RETR, RMD, RNFR/RNTO, STOR respectively). Their LSCs can be found in Appendix A. Now, when we know which actions every scenario describes, we can rename all of them in the scenario graph. The result of doing it is depicted in *Figure 55* where only a part of the graph is shown for readability purposes.

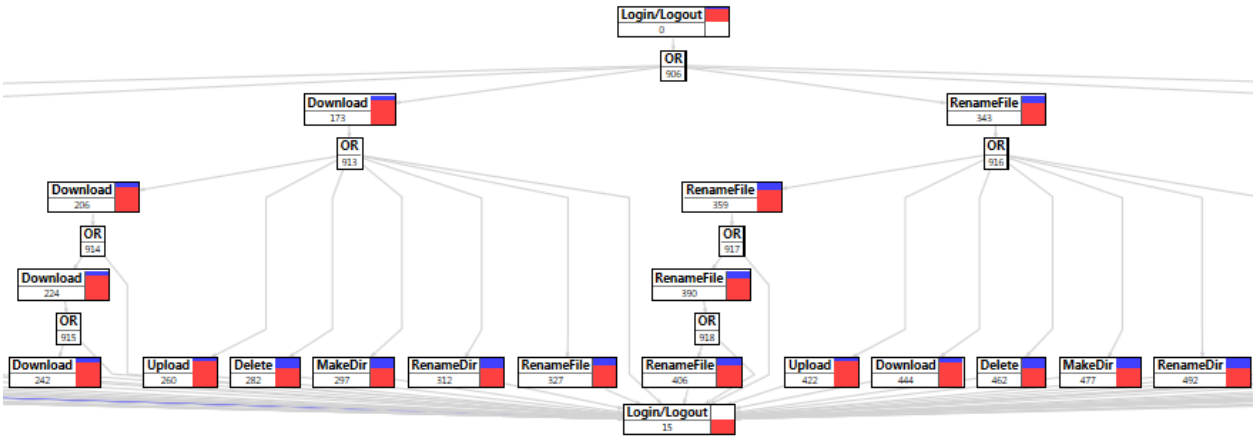


Figure 55. A part of the scenario graph after renaming scenarios

By analyzing *Figure 55* we can notice that there exits the following pattern: after the occurrences of the first segment of Login/Logout scenario and Download scenario, the user executes either Download scenario several more times or another scenario that corresponds to some other FTP command and then logs out. The similar pattern can be seen at the example of RenameFile scenario or any other scenario that represents an FTP command. It is obvious that FTP commands are independent from each other in the sense that executing one command does not forbid the execution of other command. For example, after the user executed two Download scenarios, s/he should be able to execute other commands such as RenameFile or Delete. However, that is not the case according to the scenario graph because this graph has been constructed using logs that are incomplete. To solve this problem, we fully generalize the scenario graph obtaining *Figure 56*, so that it contains only one instance of every scenario or scenario fragment.

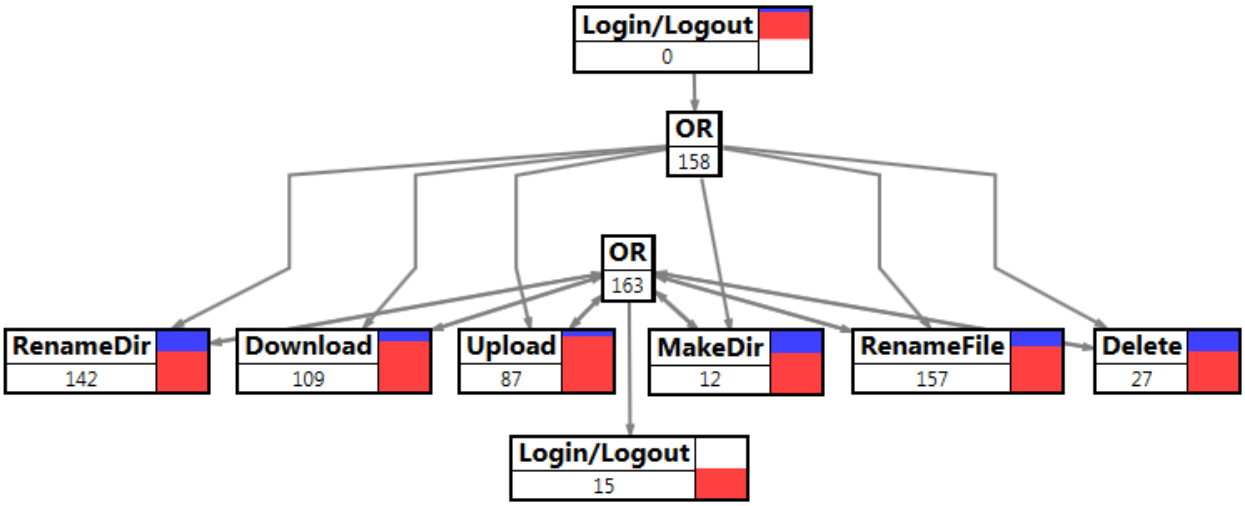


Figure 56. Fully generalized scenario graph

Figure 56 shows that after the user logged in, one of scenarios representing FTP commands occur. Afterwards, any of these scenarios can occur once again or the user can log out thereby terminating the execution of CrossFTP.

There are 6 scenarios that correspond to FTP commands and every of them occur 9 times in the log according to *Figure 57*. As shown above, after every such a scenario occurrence, there is an option to log out, so the number of occurrences of Login/Logout scenario is the sum of all other scenarios occurrences and equals 54. So, there is no option in the logs for the user to log out right after logging in.

Name	O	S	C	F
Delete	9	20	1	☑
MakeDir	9	20	1	☑
Download	9	20	1	☑
RenameDir	9	20	1	☑
RenameFile	9	20	1	☑
Upload	9	20	1	☑
Login/Logout	54	54	1	☑

Figure 57. Statistics of scenarios usage

Figure 58 shows the table with figures about objects usage for this dataset. It is notable that objects that correspond to FTP commands (they are named with capitalized letters only) are never used as senders, so it is unclear who initializes an execution of a command. Probably, it starts whenever a command object is created, e.g. in its constructor.

Name	S	R	T	GF	LF
DELE	3	0	3	☑	☑
FtpletContainer	0	16	16	☑	☑
FtpStatisticsImpl	2	9	11	☑	☑
MKD	3	0	3	☑	☑
RETR	6	0	6	☑	☑
FtpRequestImpl	0	8	8	☑	☑
RequestHandler	8	3	11	☑	☑
RMD	3	0	3	☑	☑
RNFR	1	0	1	☑	☑
RNTO	3	0	3	☑	☑
STOR	7	1	8	☑	☑
NativeFileObject	0	1	1	☑	☑
StatisticsPanel	0	2	2	☑	☑
RootPanel	3	0	3	☑	☑
FtpServer	2	3	5	☑	☑
IPRestrictorPanel	1	0	1	☑	☑
FtpConfigImpl	5	3	8	☑	☑
FtpConnectionTableModel	1	0	1	☑	☑
ConnectionManagerImpl	0	4	4	☑	☑
LoggerPanel	1	0	1	☑	☑
FtpLogFactory	0	2	2	☑	☑
PASS	3	0	3	☑	☑
QUIT	1	0	1	☑	☑
DataConnectionConfig	0	1	1	☑	☑

Figure 58. Statistics of objects usage

The table above shows quite a lot of objects, and we already succeeded to indentify how some of them are used, as well as their purpose within the system. However, the understanding of other objects is not full yet. To fill this gap, we need to dive deeper into the system logs and analyze the mined specification with a lower value of the support parameter. *Figure 59* shows the scenario graph after loading the dataset with the support value 10.

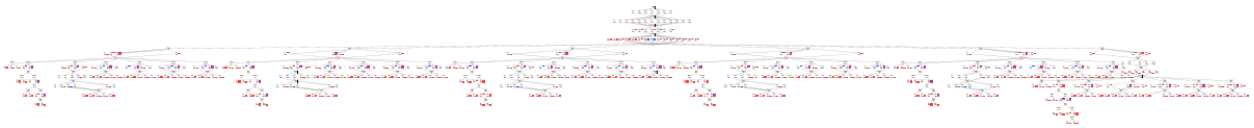


Figure 59. The scenario graph for crossftp dataset with support = 10

The scenario graph rendered with support value 10 is larger than the one depicted in *Figure 50* where support parameter equals 20. The scenario graph now contains 635 nodes while the one shown in *Figure 50* contains only 124 nodes. Such an increase in graph size deals not only with the fact that mining with a lower value of the support parameter resulted in getting a bigger number of scenarios, but also with the fact that we now have overlapping scenarios. Since we already gained some knowledge of CrossFTP workflow, we know that for this dataset generalization may help to significantly decrease the size of the scenario graph. The result of applying it is shown in *Figure 60*.

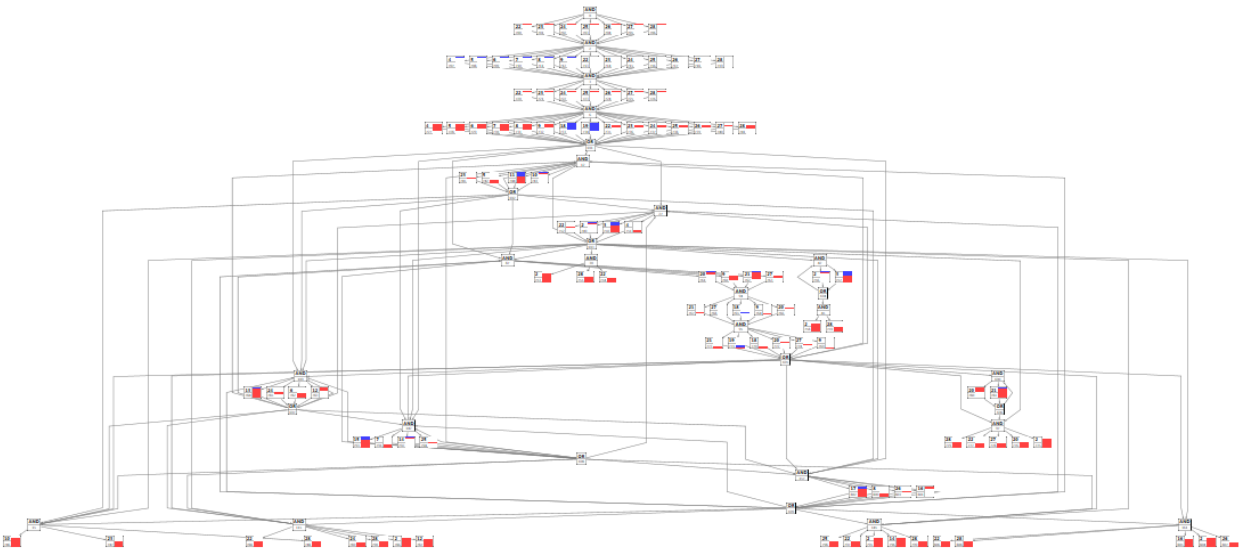


Figure 60. Generalized scenario graph with support = 10

Obviously, the scenario graph contains a lot less nodes than the one before applying generalization. However, it is still difficult to analyze without going into details. To understand the structure of the scenario graph from *Figure 60* we zoom in the diagram.

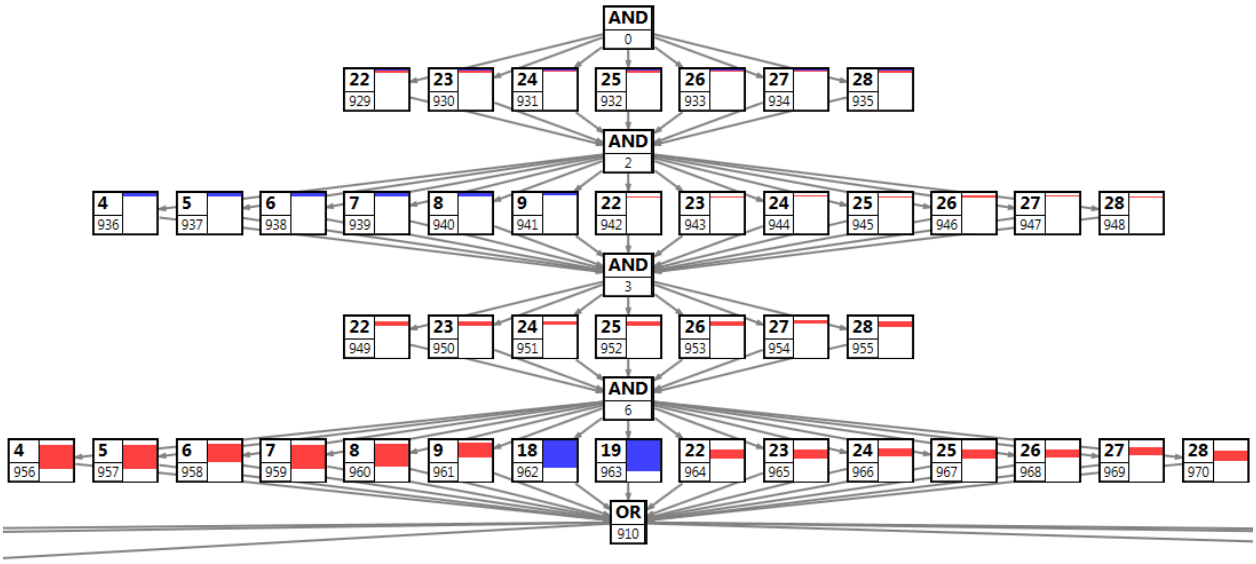


Figure 61. The upper part of the scenario graph

Figure 61 shows the part of the scenario graph before the first OR node. By looking at the set of events that form the scenarios from this part, we can see that they are those events that corresponded to the login part of Login/Logout scenarios in the scenario graph with support value 20. When we lowered this value, scenarios that describe such chain as "log in => execute FTP command => log out" have appeared and since they all overlap on login part, we observe branching behavior in the upper part of the scenario graph. A branching behavior is also found when we go down along the execution flow.

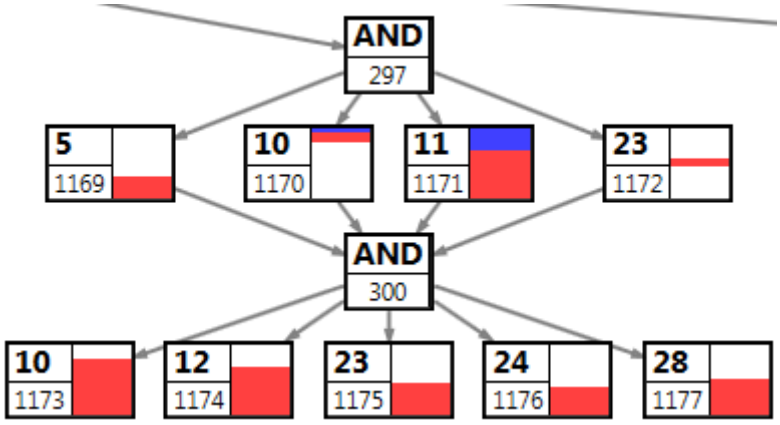


Figure 62. The part of the scenario graph showing two places in the system where scenarios overlap

Figure 62 shows the bottom part of the scenario graph, where AND node 300 is a leaf. This node indicates overlapping of scenarios 10, 12, 23, 24, 28. A part of the merged LSC for this node is depicted in *Figure 63*. Events by which scenarios overlap have grey background. These are the events for closing connections and disposing different objects, i.e. for those actions that indicate a logout of the user. The reason why there are several scenarios that overlap on the logout section at the end of the execution flow is similar to the one for having overlapping at the upper part of the scenario graph (*Figure 61*); actually, here we can see ending fragments of those scenarios.

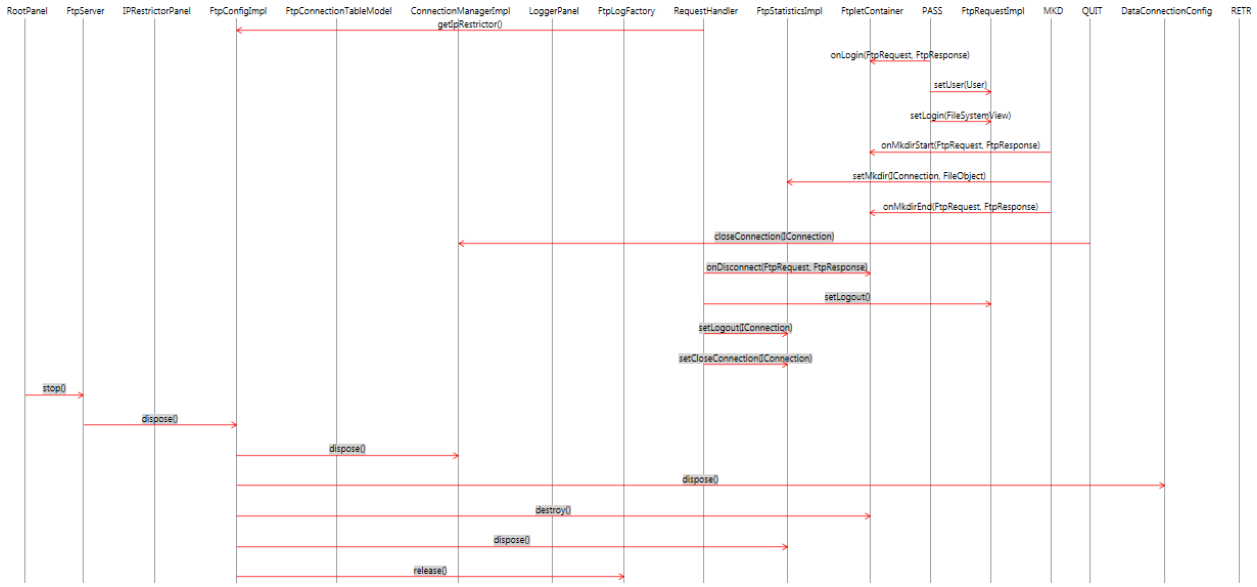


Figure 63. Merged LSC for node 300

Now, when we understood why there are many overlapping scenarios obtained after lowering the support threshold, we take a closer look at those overlapping to get some insights of the system architecture. We start with FTP command MKD, which is used to create a directory. Node 297 in *Figure 62* contains events that form this command. 4 scenarios overlap by these events: 5, 10, 11, 23. A merged LSC for node 297 is quite big, so we first filter out all the objects that do not communicate with the objects, which are used by overlapping events (see *Figure 64*). We can notice that several objects send messages to *FtpletContainer* object and according to the methods signatures those messages indicate that some request should be sent. All such methods pass not only a variable that contains a request, but also *FtpResponse* variable that is most likely used for obtaining a response. The class is named *FtpletContainer*, which should imply that this is some sort of a wrapper for individual ftplets. However, we cannot see the corresponding objects at LSCs, so it is probably not a wrapper but an implementation of an interface that defines methods for receiving and responding to FTP requests. Messages where *FtpletContainer* is a receiver are sent either by objects corresponding to FTP commands or by *RequestHandler* object. Judging by its name, this class receives a request and delegates it to a method, which can handle this request. From this point of view, it is strange that *RequestHandler* does not communicate with command objects since it could delegate some requests to them. When some request started executing, the corresponding message is sent to *FtpStatisticsImpl* object, which purpose is to store statistics information. Besides set methods, which update statistics, there should also be get methods to retrieve it.

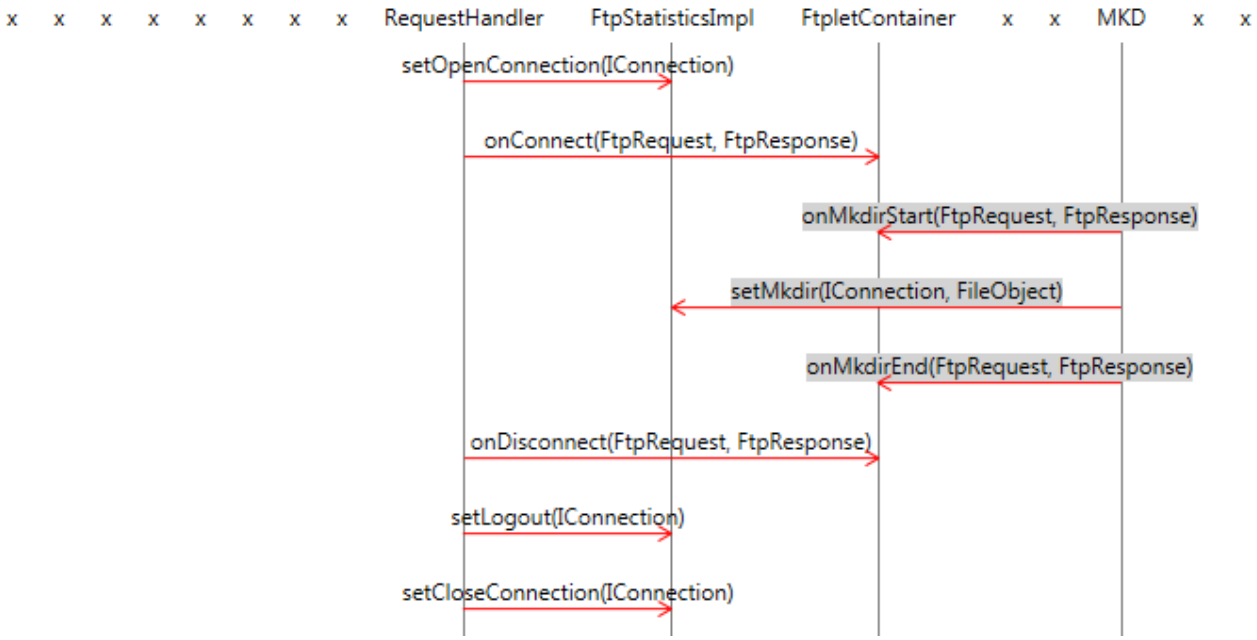


Figure 64. A merged LSC for node 297 with some objects filtered out

To find out, which object initializes the application execution and how it later communicates with other objects, we can choose a scenario that contains the root node (see *Figure 65*). Thereby, we can see that execution starts with event *start()*, which is sent by *RootPanel* object. Besides starting a server, this object is also responsible for stopping a server and loading configuration files.

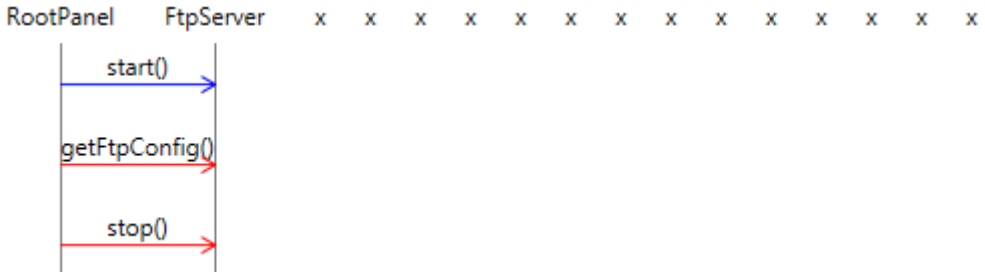


Figure 65. All events that are connected to RootPanel object

We also tested our tool at another dataset, which is larger and more complicated than CrossFTP. This second dataset represents Columba software, and the scenario graph obtained after loading the dataset to the tool is shown in *Figure 66*.

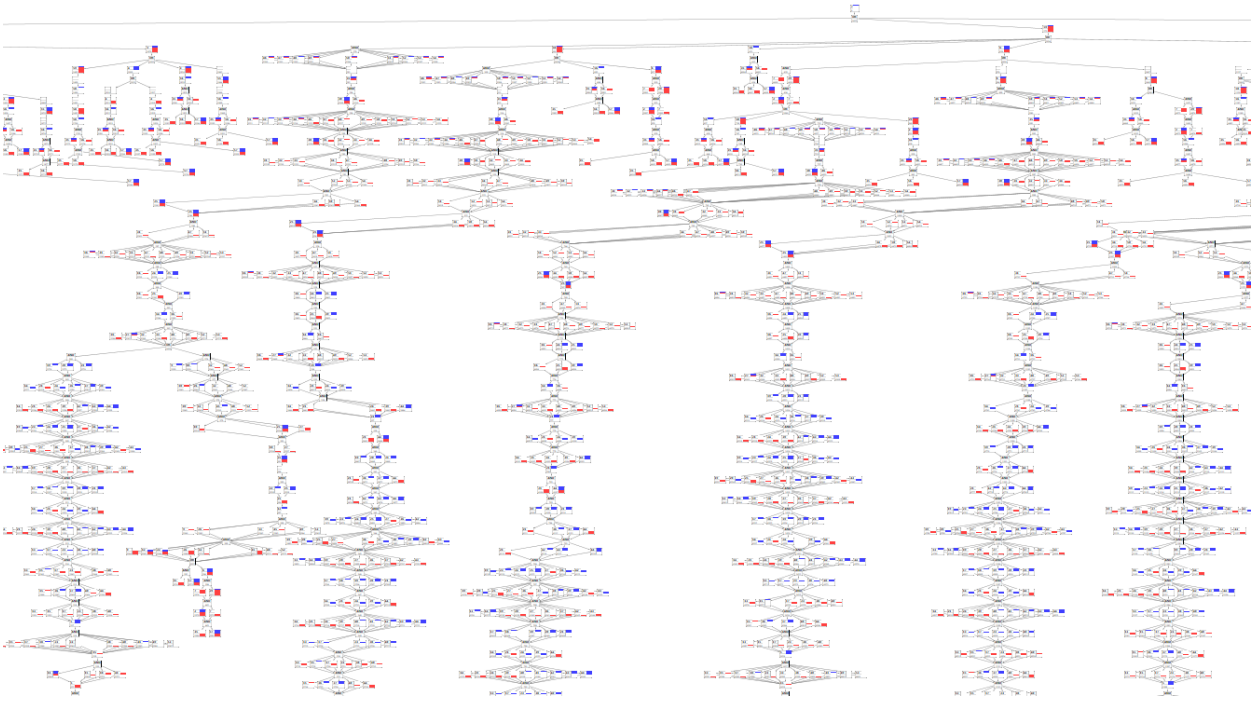


Figure 66. A part of the scenario graph for Columba dataset with support = 20

By observing the scenario graph, we can say that Columba is an email client with support of calendar. It is interesting to look at the first scenario fragment that occur when application start (see *Figure 67*). The first event is a message that is sent by *CalendarStoreFactory* to *LocalCalendarStore*. All panels initialization events happen only after this *getComponentInfoList()* event.

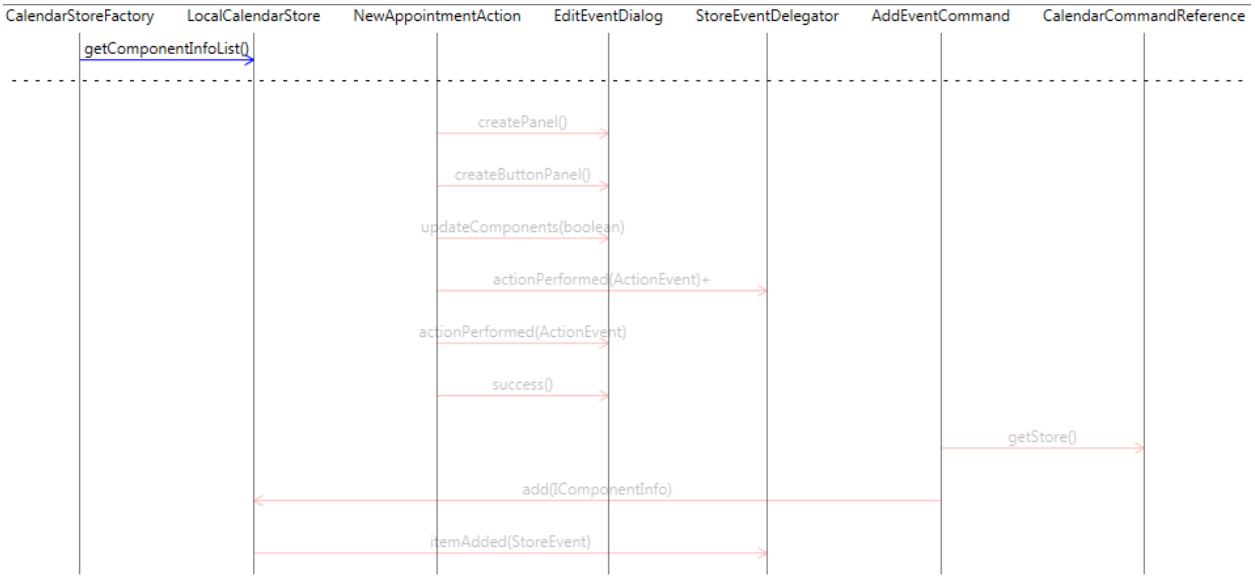


Figure 67. The beginning of Columba execution

Scenarios with the most number of occurrences correspond to leaf nodes and deal with saving the data before exiting the application.

9. Conclusion and future work

In this thesis, a visualization technique for exploring and analyzing system specification has been introduced. Such a specification is represented by an execution tree and a number of LSCs, which are mined from system logs.

For creating visualization we use the mined scenarios together with the execution tree, which represents all the traces that the log contains. It allows us to track the execution flow of the system and therefore to reveal user stories, i.e. to understand how the system is used. Then, we project scenarios onto the execution tree and build the scenario graph, which is smaller and more analyzable for the user because a scenario becomes a first class citizen instead of an event.

Although the scenario graph helps to see inter-dependencies between scenarios, it abstracts from events. In order to capture them, we introduce a separate view, which is an LSC extended with context information.

Our technique supports scenario-based and object-based filtering, which allows effectively investigating the system behavior. Configurable generalization of the scenario graph is used to eliminate the problem of incompleteness of logs without losing important information about system behavior.

The suggested visualization approach has been implemented into the tool and evaluated using the dataset generated for CrossFTP software. The results of this evaluation showed that our approach is more effective in analyzing in particular dependencies between scenarios from the system specification and the execution flow of the system than the one, which is currently used.

We consider several directions for future work. First, generalization procedure can be improved, so that it can discover not only self-loops, but also more complex ones. Second, scenario selection mechanisms that are used to choose ranges of scenarios while filtering can be reworked, so more conditions can be expressed with them.

Taking into consideration the above, it can be concluded that the formulated objectives have been reached and the project has been finished successfully.

Bibliography

[1] T. Standish. An essay on software reuse. IEEE Trans. on Software Engineering, 5(10):494–497, 1984.

[2] Harel, D., Thiagarajan, P.: Message sequence charts. In: UML for Real: Design of Embedded Real-time Systems (2003)

[3] Genest, B., Muscholl, A., Peled, D.: Message Sequence Charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 537–558. Springer, Heidelberg (2004)

[4] ITU-TS recommendation Z.120, 1996.

[5] Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods Syst. Des. 19(1), 45–80 (2001). In: Preliminary version appeared in Proceedings of 3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)

[6] Brill, M., Damm, W., Klose, J., Westphal, B., Wittke, H.: Live Sequence Charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147, pp. 374–399. Springer, Heidelberg (2004)

[7] D. Harel and I. Segall, "Visualizing Inter-Dependencies between Scenarios," in Proc. of the 4th ACM symposium on Software visualization, ser. SoftVis, 2008, pp. 145-153.

[8] D. Fahland, D. Lo, and S. Maoz. Mining branching-time scenarios. In IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013.

[9] G. Sibay, S. Uchitel, and V. A. Braberman. Existential live sequence charts revisited. In ICSE, pages 41–50, 2008.

[10] Maoz, S., Harel, D.: On tracing reactive systems. Softw. Syst. Model. 10(4), 447–468 (2011)

[11] M. Gordon and D. Harel, "Semantic Navigation Strategies for Scenario-Based Programming", IEEE Symposium on Visual Languages and Human-Centric Computing 2010.

[12] N. Eitan , M. Gordon , D. Harel , Assaf Marron , Gera Weiss, On Visualization and Comprehension of Scenario-Based Programs, Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, p.189-192, June 22-24, 2011

Appendix A

This appendix contains the output of the tool, which is currently used to present the results of the miner.

General

executed on: 2015/02/06 04:52:37
input file: ./crossftp.xes.gz
min. support threshold: 10
confidence branching scenarios: 2.0
confidence linear scenarios: 1.0

Input

number of nodes: 921
number of events: 53
avg. out degree: 0.998914223669924
max. out degree: 7.0
depth: 63
width: 51

max. support/confidence to find at least one alternative with k branches

k	1	2	3	4	5	6	7
support	9	9	9	9	9	9	1
confidence	1.0	1.0	1.0	1.0	1.0	1.0	0.45

max. support/confidence to find all alternatives with k branches

k	1	2	3	4	5	6	7
support	9	6	1	1	1	1	1
confidence	1.0	0.7	0.45	0.45	0.45	0.45	0.45

Discovered Scenarios (general statistics)

tree coverage (branch), main chart events: 0.0
tree coverage (linear), main chart events: 0.9072494669509595

tree coverage (branch), all chart events: 0.0
tree coverage (linear), all chart events: 1.0
alphabet coverage (branch): 0.0
alphabet coverage (linear): 1.0

supported words: 1653400
time to find supported words: 203713ms
time to find branching scenarios: 327957ms
time to find linear scenarios : 896085ms

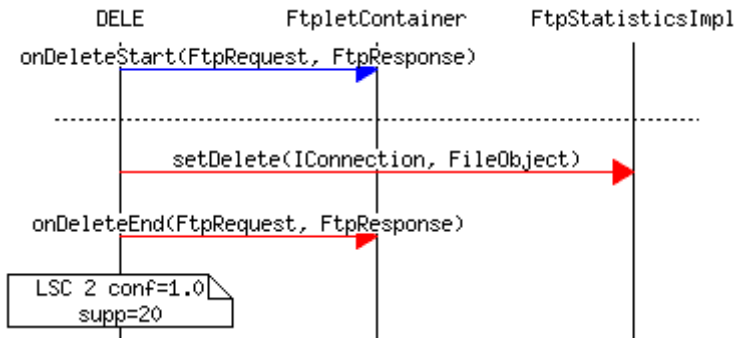
total number of scenarios found by branching miner: 0
total number of scenarios found by linear miner: 4054915

strictly branching scenarios (ignoring subsumed scenarios)
found **0 strictly branching** scenarios
length pre-chart (min/avg/max): 0.0/0.0/0.0
length main-chart (min/avg/max): 0.0/0.0/0.0
#components (min/avg/max): 0.0/0.0/0.0

linear and branching scenarios (ignoring subsumed scenarios)
found **9 linear** and branching scenarios
length pre-chart (min/avg/max): 1.0/2.3333333333333335/7.0
length main-chart (min/avg/max): 1.0/5.444444444444445/23.0
#components (min/avg/max): 3.0/6.222222222222222/15.0

onDeleteStart(FtpRequest, FtpResponse)

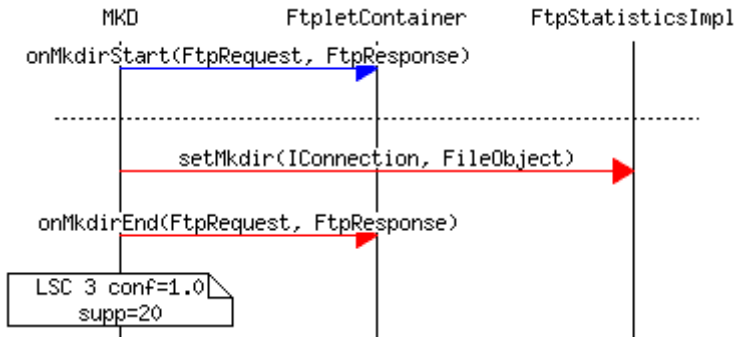
1 linear and branching LSC



scenario #67287903
scenario: 27 --- 28 29
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

onMkdirStart(FtpRequest, FtpResponse)

1 linear and branching LSC

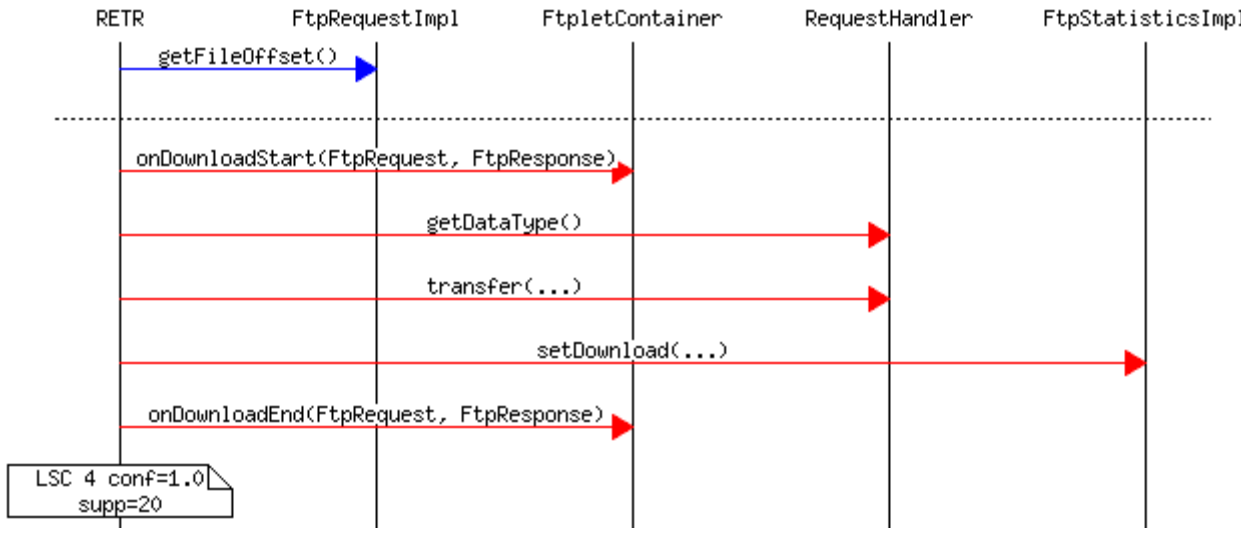


scenario #67287600
scenario: 12 --- 13 14
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

---

getFileOffset()

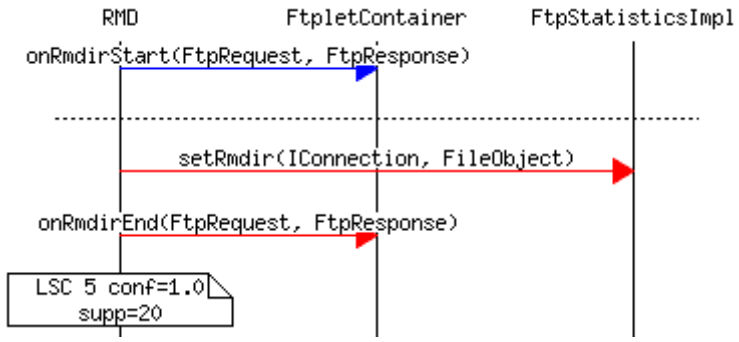
1 linear and branching LSC



scenario #67175830
scenario: 40 --- 41 42 43 44 45
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

onRmdirStart(FtpRequest, FtpResponse)

1 linear and branching LSC

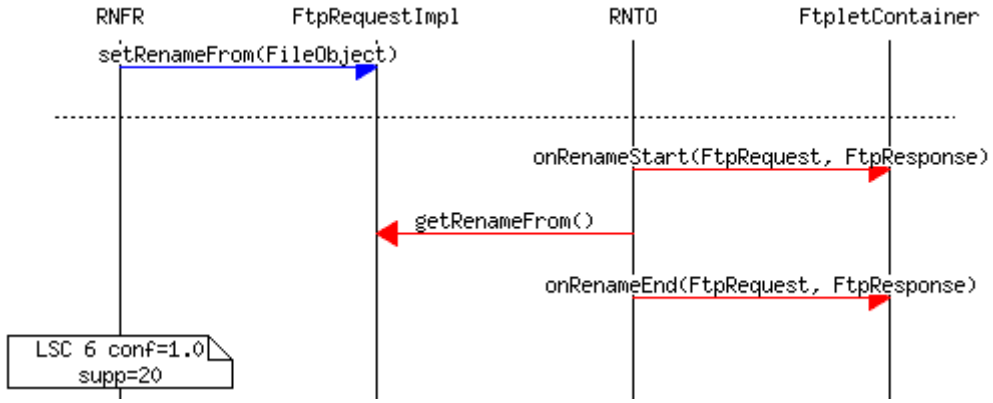


scenario #67289000
scenario: 46 --- 47 48
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

---

setRenameFrom(FileObject)

1 linear and branching LSC

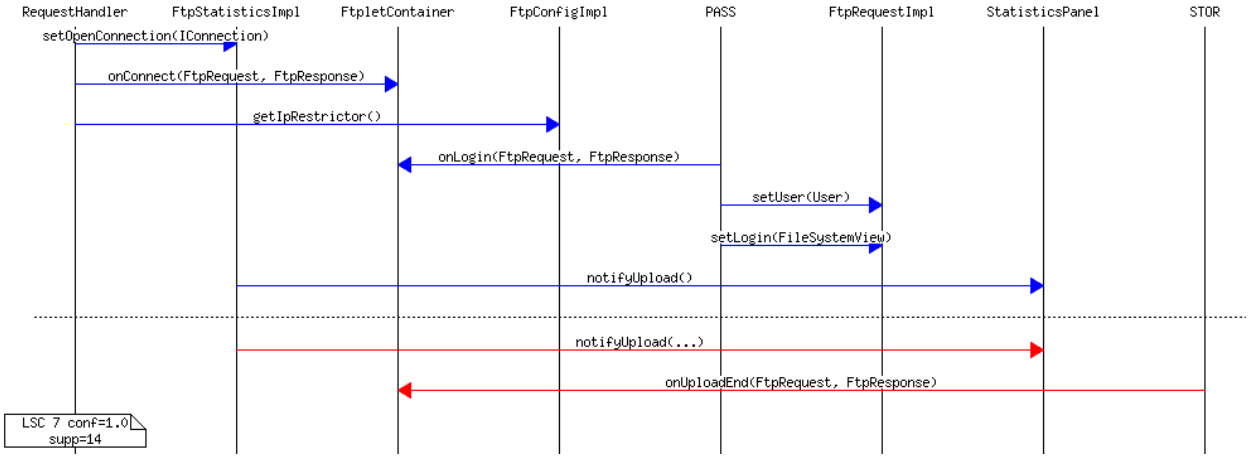


scenario #67281623
scenario: 49 --- 50 51 52
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

---

setOpenConnection(IConnection) onConnect(FtpRequest, FtpResponse) getIpRestrictor() onLogin(FtpRequest, FtpResponse) setUser(User) setLogin(FileSystemView) notifyUpload()

1 linear and branching LSC



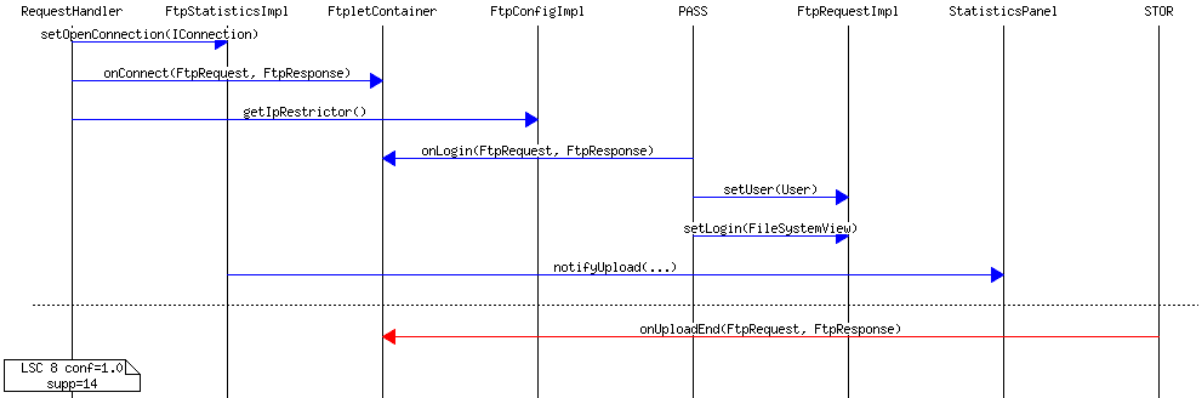
scenario #65904483

scenario: 6 7 8 9 10 11 37 --- 38 39
support: 14
confidence (branch) 1.0
confidence (linear) 1.0

---

setOpenConnection(IConnection) onConnect(FtpRequest, FtpResponse) getIpRestrictor() onLogin(FtpRequest, FtpResponse) setUser(User) setLogin(FileSystemView) notifyUpload(...)

1 linear and branching LSC



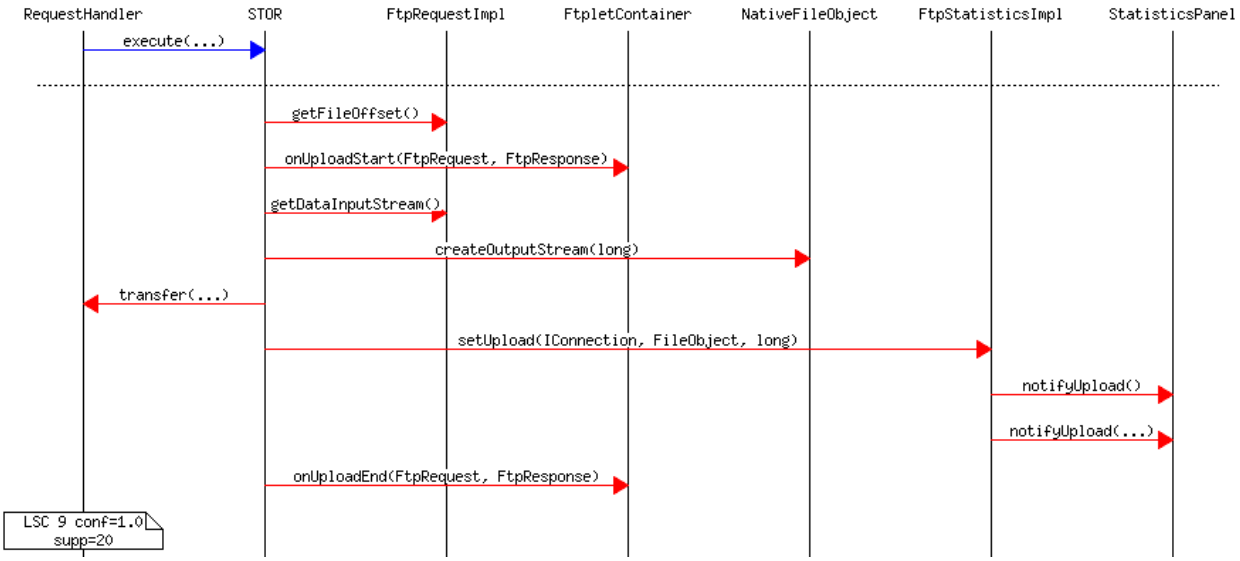
scenario #66517014

scenario: 6 7 8 9 10 11 38 --- 39
support: 14
confidence (branch) 1.0
confidence (linear) 1.0

---

execute(...)

1 linear and branching LSC

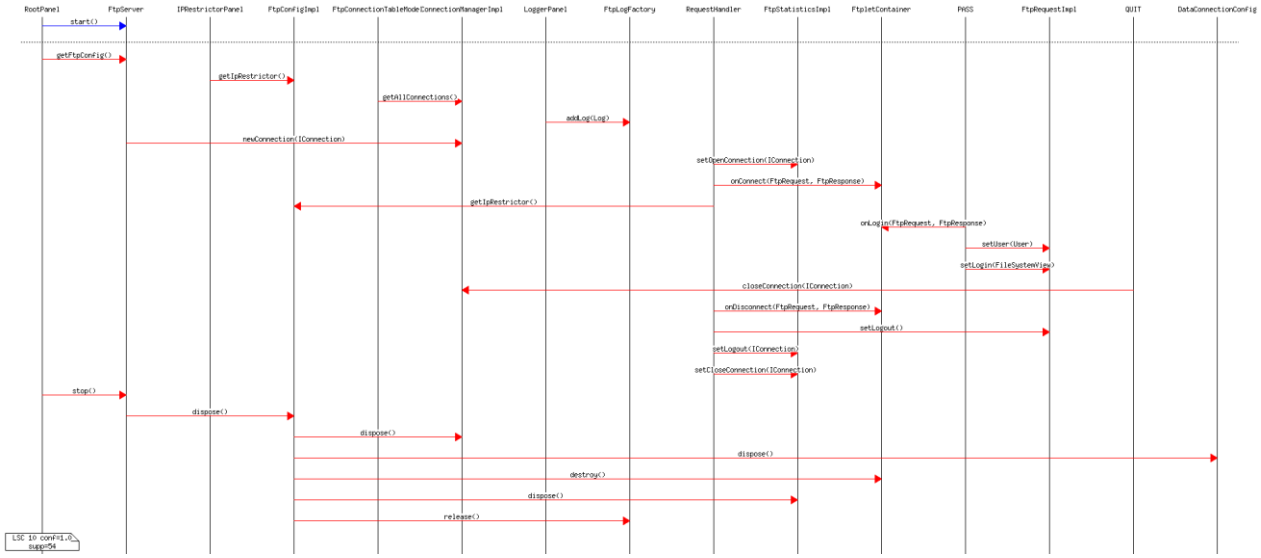


scenario #65490444
scenario: 30 --- 31 32 33 34 35 36 37 38 39
support: 20
confidence (branch) 1.0
confidence (linear) 1.0

---

start()

1 linear and branching LSC



scenario #36412073
scenario: 0 --- 1 2 3 4 5 6 7 8 9 10 11 15 16 17 18 19 20 21 22 23 24 25 26
support: 54
confidence (branch) 1.0
confidence (linear) 1.0