Profile-Based Approach to Support Comprehension of Software Behavior

Johannes Koskinen, Markus Kettunen and Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{johannes.koskinen, markus.kettunen, tarja.systa}@tut.fi

Abstract

When reusing an existing software component, the software developer needs to understand the functionality and possible extension points of the component, as well as constraints and rules to be taken into account when using it. In addition to structural rules, the software component may imply interaction rules that must be followed in application development.

In this paper we discuss behavioral profiles, given in UML, used to capture and illustrate architecturally significant behavioral rules. The behavioral rules may capture interaction rules to be obeyed when reusing an existing software component. They can also be used to support runtime analysis of existing systems: with a proper tool support, the validity of extracted interaction models can be automatically checked against the behavioral rules given in the profiles. Moreover, the profiles can be used to prune the size of the interaction trace to include only the information relevant from the point of view of the behavioral rules. In this paper we discuss such tool support and demonstrate the usefulness of the approach by applying behavioral profiles to define and illustrate behavioral rules relevant for applications using the Graphical Editing Framework (GEF). Moreover, we analyze a sample GEF application by validating its run-time behaviour with respect to the defined behavioral rules.

1. Introduction

Software architecture is traditionally understood as a structural rather than behavioral concept. However, besides the structural relationships among components there can be architecturally significant collaborative behavior. For example, general architectural solutions like architectural or design patterns [3] are often characterized by certain interaction patterns. Similarly, product-line architectures [2, 4] may imply behavioral rules that must be followed in the applications built on top of the product-line platforms. Further, consider e.g. object-oriented application frameworks, which are reusable, “semi–complete” applications that can be specialized to produce custom applications [10]. The application frameworks, especially white-box frameworks, which rely on inheritance and dynamic binding to achieve extensibility, typically require that the application-specific code follows more or less strict forms, not only in terms of the static structure but also in terms of the interactions of the objects. Due to conditional statements and dynamic binding, it is not possible, in general, to infer statically whether the required interaction forms are followed or not.

To specialize an application framework into an application, the software engineer typically wants to know how to use the framework, what are the services it provides and how they work, and what are the constraints, rules, and implications of its specialization. A documentation that lists different extension points and interfaces that can be called is insufficient; support for understanding which methods to use and in which order to achieve a desired functionality using existing services would be desirable. As with any software products, from the user’s point of view, a user’s manual is often the most interesting piece of documentation and easiest to understand and use. As in user’s manuals, example use cases and scenarios can be used for that purpose.

To understand the provided services and functionality of an application framework (or any reusable component), we need to understand what the services are, how to use them, and how they work. To achieve this goal, the software engineer needs to comprehend how and what parts of the software are related to certain kind of run-time behavior. Traditionally, the focus of architecture recovery and reverse engineering has been on structural models that can be analyzed for finding answers to questions starting with What. When considering the problem from the maintainer’s, re-engineer’s or reuser’s point of view, we should also pay attention on constructing models that give answers to questions starting with How. This way of thinking raises new
questions and problems concerning information extraction as well as information visualization and documentation.

Various tools and methods have been provided to support run-time analysis of software systems [5, 9, 12, 16, 17, 19]. These tools often extract event traces (including e.g. operation calls and information on creation and destruction of objects) and use different kinds of scenario notations for visualizing the event traces. One of the main challenges in these approaches is to find illustrative visualization method that both scales up for the huge amount of information extracted and supports the user to focus on essential pieces of information she is interested in.

In this paper, we discuss behavioral profiles [13], that can be used to express architecturally significant behavioral rules. We present behavioral profiles as UML profiles. The behavioral rules are given in terms of the roles different participants play within a collaboration. This approach leads to example-like descriptions that are matched against actual system models, to recognize instances of the rules. The advantage of this approach is a more intuitive and easy-to-use notation for defining behavioral profiles compared e.g. to the use of OCL (Object Constraint Language) for expressing the rules.

A behavioral profile can be used to support software designers in creating behavioral models that conform to some predefined rules or ensuring that an application behaves correctly with respect to the rules given in the profiles. With proper tool support, behavioral models can be automatically checked against behavioral profiles. A behavioral model can be created by a designer during the development process, or it may be generated on the basis of an execution trace obtained from a running system, using dynamic reverse engineering techniques. The essence of the approach is to extract role behavior relevant from the point of view of the behavioral rules stated in behavior profiles, not in generating event traces. In fact, we can use existing techniques and tools to extract the event trace. Including only the essential role behavior from the event trace and filtering out the rest also helps us to deal with the huge amount of sequential information typically included in the traces.

Behavioral profiles have initially been introduced in [13]. In this paper, we focus on the tool environment, usage of behavioral profiles, and the process flow. We apply behavioral profiles for documenting relevant constraints and rules of a large application framework, namely, Graphical Editing Framework (GEF) [7], thus supporting the software engineer in understanding how the framework is to be correctly specialized. We further extract execution traces of an application built as the specialization of a framework, filter out the scenarios relevant from the point of view of rules defined in the profiles, transform them into sequence diagrams and then check the sequence diagrams against the behavioral profiles given for the framework.

2. Behavioral Profiles

Behavioral profiles are a technique to express rules for different types of the run-time behavior of software systems. In addition to specifying simple interaction patterns (i.e. scenarios), the behavioral profiles can express alternatives and repetitions inside the patterns. Also a part of the interaction can be grouped to larger units that are handled as a single call. As often it is easier to specify disallowed execution paths, the pattern can also present an illegal order of an interaction.

The notation for behavioral profiles consists of a set of class and sequence diagrams (Figure 1). The sequence diagrams are used to describe interactions between roles, while the class diagrams specify the roles used in a profile. The roles can be specialized using class inheritance. The specialized role can be in the place of the base role, thus allowing e.g. role categorization and reducing the number of role bindings.

![Figure 1. Notation for behavioral profiles](image)

The roles specified in behavioral profiles are bound to corresponding elements in a model using stereotypes. As the classes and the operations are bound with played roles, sequence diagrams attached to the classes describe the role behavior, and thus follow the rules specified in the profile. Unbound participants and operation calls can be mixed with the bound ones. An example of such role binding is shown in Figure 2. It states that ElementFactory plays the role ElementFactoryRole, and the profile roles makeCreatureModelRole and makeCreatureViewRole are played in the model by the operations makeCreatureModel and makeCreatureView.
Figure 2. Binding roles

The profile can be applied to classes or class instances. In the latter case, the instances playing particular role are fixed when they are referred to for the first time. Any new combination of the participating instances will cause a new instance of the profile to be created.

In addition to specified roles, the special participant Any can be used in the place of any participant (with or without role binding). Similarly, any operation call can be expressed with the special call Any.

For more details considering the notation, the reader is referred to [13].

3. An Overview of the Processing Environment

The environment for the behavioral profiles consists of a UML CASE tool used for design model and profile visualization, components for processing the conformance checks of the models against the profiles, and CASE tool specific components to import and export UML models. In addition, a user interface for navigating between profiles and models is provided. This process is shown in Figure 3.

3.1. Input

The input of the process consists of profile definitions, role descriptions and one or more execution traces. As profile definitions are regular UML models, they can be handled and imported like any UML model using the importers provided by the tool environment. The definitions are specialized for the used domain (and thus for the specific model) by adding stereotypes to specify played roles. This detailed class diagram could be generated from the result of a reverse engineering activity. Depending on the system, the role stereotypes can be added manually, semi-

automatically or automatically based on the domain specific rules and naming conventions (e.g., every class name with postfix Server means that the class plays role ServerRole).

The execution traces to be checked can be generated by a tracing tool or using instrumented code. Of course, the tracing tool will vary depending on the used system, programming language and platform. Because trace files are usually huge with thousands of messages, visualizing them using a CASE tool might be impossible. So, instead of first loading traces to the CASE tool and then importing them with the proper tool importer for processing, the traces are imported directly. Thus, large traces can be used without any pre-filtering. The importer used for trace files depends on the format of the log files. Trace logs are often only regular text files (or XML) and logged operation calls are referred only with the operation signature. Thus, more detailed class diagrams with the stereotype information needed by the profile tools are used to generate a single model that follows the UML metamodel. The combined model along with profiles are transferred to the next phase.

3.2. Processing

The Profile Engine takes profile, model, and message exchange information as inputs and generates output to be exported. The engine consists of a set of interacting components. In the first stage, the profile and role information is pre-processed to be suitable for profile search. The profiles are used to create a state machine presentation that is used for matching profile sequences with the given sequence diagrams. Synthesizing state machines from sequence diagrams has been widely studied (for example [14, 20, 22, 23]).

When a profile is applied to the instances of the role mapped classes, a copy of the state machine is created for
each set of the profile role bindings. Every state machine instance contains information on bound objects. When a method call is made, state machine instances having bindings to the method caller and the callee are triggered. If no suitable state machine instance is found, a new state machine instance is created, and the caller and the callee objects are bound to the corresponding roles.

The method call is regarded as violating the profile, if any of the triggered state machine instances is moved to an error state. When a violation occurs, an error information block is generated for the violation and added to the violating message. Finally, the model with the profile violation diagrams and found profile instances are generated and passed to Result Exporter.

3.3. Output

The Profile Engine produces two kinds of output: profile violation information and interactions for found profile instances. The violations show the execution flow fragments breaking rules specified by the profiles (like lacking a mandatory method call). The profile instances represent execution flow patterns that have been matched with the interaction patterns defined in profiles. The profile instances are useful, for instance, to filter huge trace information based on pattern and role specifications. As the filtering is done using roles, the filtered output depends on actual implementation and role mappings. Thus, the filtering is more efficient compared to common trace filters that are based on message names or specified participants.

After the outputs are exported to the CASE tool, the user interface for error browsing and profile instance navigation (called Profile Browser) is started. Profile Browser reads information stored by the Profile Engine and shows the profile instances and violations found, categorizing them using several selectable criteria. Error information, along with the operation call caused by the violation and corresponding profile information, is shown on one page (Violations). The instance information with roles and participants is shown on another page (Instances). Using the interface, the user can easily navigate to (possible erroneous) interaction fragments, class or operation specifications and profile diagrams.

In addition to just showing the results in a CASE tool, the filtered output can be exported to an XML (or XMI) file to be processed with separate tools.

3.4. Current Implementation

For the tool support we have used an existing UML processing platform xUMLi (executable UML interface) [1]. It provides a model processing API following the UML metamodel, an OCL interpreter, importers and exporters for CASE tools, and scripting mechanism. The processing and user interface components (like Profile Engine) are implemented as xUMLi components.
4. Case Study: Applying Behavioral Profiles

In this section we demonstrate the usage of the behavioral profiles with an application framework. As an example we use a UML state machine editor that was implemented as an assignment in a course on software architectures at Tampere University of Technology. The implementation of the editor consists of a state machine editor framework that was built on top of the Graphical Editing Framework (GEF) [7], and the editor itself that is a specialization of the state machine framework.

GEF offers an infrastructure for developing graphical representations and editors for existing models by providing the methods and practices for how the model is displayed graphically and how the modifications to it are handled. GEF is available as an Eclipse plugin.

The architecture of the GEF framework is based on MVC (Model-View-Controller). The controller in GEF is called an EditPart. EditParts are the link between the model and the view (Figure 4). GEF does not give any restrictions for the implementation of the model or to communication methods between EditParts and model objects.

![Figure 4. GEF MVC model](image)

The state machine editor framework is a simple framework for different kinds of state machine editors. It has default implementation for the model, controller and view parts. New state machine editors can be implemented by specializing the framework’s default implementation.

4.1. Behavioral Profiles

For the behavioral profiles, 12 class roles and 36 operation roles were used (partly shown in Figure 5). These roles were found from the GEF’s base architecture and from the design patterns [8], like Observer, Command and Abstract factory, which are used in the GEF framework. Most of the roles were bound to more than one class. For example, 21 classes are bound to ModelRole that represents the model part of the MVC.

Like the roles, most of the profiles were found from the GEF’s architecture and used design patterns. The rest were found from the GEF’s manual. The profiles used for the framework can be divided into two categories:

1. A redefinition of a framework operation must call specific operations.
2. The profile is an interaction pattern that illustrates the behavior of the framework and communication between different parts of the framework.

The profiles belonging to category (1) state that a specific operation call is followed by the defined set of operation calls from the same role. As these operation calls can be made in any order, they are placed inside a par interaction fragment (parallel execution). The profiles belonging to category (2) illustrate interaction between different roles in a form of operation calls. Loop and option fragments are used to define that the specific operation can be called more than once or that certain operation calls are optional. These profiles are usually considerably larger, including roles for different parts of the application.

Most of the interaction patterns concentrate on illustrating how the communication between EditParts and model objects should be done, and thus how the modifications are made to the model and how the changes are made to views based on these modifications. In this paper we will focus on two profiles, which will cover that kind of interaction. Totally five profiles were applied to the state machine editor, two of them belonging to the category (1) and the rest of them to the category (2).

In GEF, all modifications to the model is made by using commands. Using of commands also provides means to implement undo and redo functionality. As one of the main ideas of the MVC is to keep the model independent of other parts (controller and view), Observer design pattern is used for this purpose.

Commands are created by EditPolicies which are provided by the EditParts. The EditPolicies define what kinds of operations are possible for the model(s) that EditPart controls. When some modifications to the model are needed, the EditPart is queried for the Command that can be used to complete the task. Next, the EditPart checks the EditPolicies it owns to find a suitable Command for the task. If appropriate EditPolicy is found, the Command is constructed by the EditPolicy and returned. A behavioral profile for querying and creating Commands is shown in Figure 6.

After the Command is found it can be executed. First the tool is told that the Command can be now executed. The tool gives the Command to the command stack that executes it. The Command modifies the model, which notifies the changes to the listeners. A behavioral profile for the Command execution is illustrated in Figure 7.
Figure 5. Role definitions

Figure 6. Profile for retrieving the Command in GEF
4.2. Models

To apply the profiles for the state machine editor, a UML model for the framework and the specialization is needed. The model was generated using a reverse engineering tool for Java. The used example implementation has about 1200 classes.

The roles (i.e. stereotypes) for the classes were added semiautomatically. The roles for the base classes were handled by adding manually required stereotypes using the CASE tool. These roles were then copied automatically to the specialized classes using the scripting capabilities provided by the tool. Thus, most of the roles were added manually only once. Of course, the model could be automatically stereotyped based on separately made role lists (like spreadsheet files).

For the execution trace, a simple state machine was built using the editor. The editor provides a state machine simulation possibility, which was also used during tracing process. The actual trace file was generated using the Eclipse Test & Performance Tools Platform (TPTP) [6]. The tracing tool generates XML files that were imported and combined with the existing UML class model.

4.3. Results

In this case study about 20 percentages of the stereotyping was defined manually and the rest were automatically generated by copying the stereotypes from the base classes. The profiles and the trace with role information were processed using the Profile Engine. The result was exported then back to the CASE tool and analyzed using the Profile Browser (in Figure 8). The Profile Engine found at least one instance for each defined behavioral profile. Totally, there were 32 found instances, but no violations. For the example profiles, there were 11 instances for both of them.

The trace file contained 6378 objects, 430 class definitions, 1412 methods and 47033 method calls. This trace was further filtered to only contain information relevant from the point of view of the behavioral profiles. As a result, from the trace file’s 47033 method calls 46751, about 99 percentages, were filtered out.
4.4. Discussion

When analyzing the generated trace file without the behavioral profiles, it became clear the used interaction patterns would be almost impossible to find from the trace manually. Filtering the trace using only participants (or classes) as a filtering criterion would not reduce unnecessary operation calls enough, because the number of the classes participating in the interaction pattern was still too high. Filtering the trace further using the called operation name is difficult if the naming convention does not follow the roles that the operations are playing.

By using the behavioral profiles the trace was filtered out of all messages that did not belong to any instance of any of the profiles. Therefore only the method calls that we were interested of were left. This should make the runtime behavior of the implementation easier to understand and should help to understand how different design patterns are used in the implementation and how they behave at runtime.

No violations were found from the used example implementation. One reason for this could be that the used example implementation and profiles were made by the same person. In an ideal case, the designer of the framework also designs the profiles that illustrate how the specialization of the framework should behave at runtime and the implementer only checks the implementation against the profiles.

One of the used profiles defines the essential parts of the EditParts life cycle. This kind of profile can be used for example to check that all necessary reserved resources are in some point also released. By using the profile, possible memory leaks can be found automatically.

The profiles used in this case study could also be used to help the checking of the assignments of e.g. the software architectures course by the course personnel or the student themselves.

5. Related Work

Various techniques and tools supporting run-time analysis of software have been developed. One of their major challenges is to manage with a huge amount of run-time information generated when running the subject system. In what follows, we related our work to techniques and tools supporting run-time analysis of software systems. We especially focus on techniques supporting abstraction and selective information generation.

To support the analysis of the run-time behavior of a subject software system, various kinds of scenario notations are typically used for visualizing the monitored event trace [5, 9, 12, 16, 17, 19]. The extracted event traces, which may contain e.g. operation calls, are typically large. Therefore, recurring behavioral patterns, such as a certain sequence of operation calls, are often identified and used to structure and abstract the scenarios to make them more comprehensible. The pattern identification method chosen depends on the way the event trace has been produced and on the analysis task in question. For instance, the event trace may contain pattern instances that include additional communication to that described in the pattern, due to e.g. more detailed instrumentation method used. Behavioral pattern recognition allowing interleaved messages is supported e.g. by ISVis tool[9]. In this paper we have discussed behavioral profiles that can be used to model the communication patterns of interest. Tool support for indentifying them from the execution trace helps the engineer to understand the communication during the example runs. Correspondingly, behavioral profiles can be used to check that the assumed communication patterns and protocols have been followed accordingly.

In addition to using behavioral profiles to model behavioral patterns that in turn can be used for abstracting the execution traces to be monitored, the behavioral profiles can be used to slice the execution trace to capture only parts of the run-time behavior of interest. The individual program comprehension tasks are often well focused and deal with a certain part of the subject system only. Selective instrumentation [19] of the subject system is another technique often used to reduce the size of the event trace to be generated and to focus on run-time analysis of selected parts only. The size of event trace can also be kept smaller by allowing turning the event monitoring on and off during the execution of the subject system.

In our approach, behavioral profiles are used to capture high-level architecture-related rules that are assumed or required from the subject system. This approach is thus related to top-down reverse engineering techniques in which a high-level model is first composed (typically manually) that is then matched with the software artifacts extracted from
the subject system. In [15] Murphy et al. propose a reflection model technique that follows such a top-down reverse engineering approach, focusing on structural aspects of the software. In [21] Walker et al. introduce a tool AVID for architecture-level visualization of behavior in a top-down reverse engineering fashion. AVID uses a specific graph-based visualization technique. In AVID, the software engineer composes a high-level model of the system and defines a mapping from it to the extracted run-time information. The tool uses the mapping to produce a high-level visualization of the behavior. In our approach, the mapping from profile rules to the event trace is based on stereotypes. In fact, the roles of the events, which are defined by stereotypes, are of interest rather than the specific events (e.g. method calls). Even though stereotyping is in principle manual, in certain types of applications, like one discussed in this paper, a large degree of the stereotypes can be generated automatically.

6. Discussion

Software architecture defines not only structural but also behavioral patterns. We need therefore techniques to specify behavioral rules and to enforce them during the detailed design and implementation of the system. The UML profile mechanism can be used to define customized modeling languages for a particular architecture, allowing only models that conform to the architectural rules [18]. We have applied UML profiles to define behavioral rules associated with a given architecture. The constructed behavioral profiles help the engineer to understand those behavioral rules and constraints relevant for the subject software system or component. Moreover, with proper tool support, such behavioral profiles can be used to guarantee that the created behavioral models (given as sequence diagrams) are consistent with the architecture, and to test an implementation against behavioral architectural rules. The latter requires reverse engineering capabilities for producing behavioral models from a running system. We demonstrated the usefulness of behavioral profiles in the validation of the behavior of a framework-based application against the behavioral rules implied by the framework. While we used framework specialization as an example, in principle the same technique can be applied for specifying and enforcing any collaboration patterns.

In addition to support software reuse, behavioral profiles are applicable to support reverse engineering and program comprehension of existing systems. For some reverse engineering tasks, especially when studying communication-based systems, static analysis is not sufficient. Instead, for identifying communication patterns, analysis of the run-time behavior is needed. In such cases, behavioral profiles can be used to model the communication patterns of interest. Tool support for identifying them from the execution trace would help the engineer to understand the communication during the example runs. Correspondingly, the behavioral profiles could be used for checking that the assumed communication patterns and protocols have been followed accordingly.

Abstraction and filtering rules used when reverse engineering the run-time behavior are often based on looking for specific object types and events from the event trace. Behavioral profiles can provide a more flexible and powerful means to define such rules; instead of focusing on specific events or messages, the rules could concern the roles the messages and objects play at a certain point of execution. That can be enabled by using stereotypes to define the different role types. This implies that a certain message or object, for instance, can play different roles, depending on the context it is used in. In addition, a certain role could be played by different messages (or objects). When analyzing the run-time behavior of a subject software system, it is indeed relevant to understand the context in which a certain method has been called, i.e., the role that method call plays.

An obvious extension of our approach is to support on-line run-time monitoring of a software system, rather than off-line checking of forward or reverse engineered interaction diagrams. The former type of monitoring becomes particularly interesting in a situation where unanticipated components take part in a collaboration during the use of the system, like in Web Services. If the on-line monitoring reports a deviation from a predefined behavioral pattern, specified as a behavioral profile, various recovery actions can be considered, like dynamically replacing a misbehaving component with another one. In the case of message-based communication, monitoring code can be generated from behavioral profiles and run by the message dispatcher, which is aware of the roles components play in a collaboration. We are currently investigating this kind of facility in the context of Web Services. In the current implementation of the run-time monitoring approach, proposed in [11], aspect-oriented techniques are advocated for monitoring.

Acknowledgments

This work has been financially supported by Nokia.

References
