Iterative Model-Driven Development of Adaptable Service-Based Applications

Leen Lambers and Hartmut Ehrig
Technical University Berlin
Franklinstrasse, 28/29 - 10587 Berlin
{leen,ehrig}@cs.tu-berlin.de

Leonardo Mariani and Mauro Pezzè[*]
University of Milano Bicocca
via Bicocca degli Arcimboldi, 8 - 20126 Milano
{mariani,pezze}@disco.unimib.it

ABSTRACT

Flexibility and interoperability make web services well suited for designing *highly-customizable reactive service-based applications*, that is interactive applications that can be rapidly adapted to new requirements and environmental conditions. This is the case, for example of personal data managers that many users tailor to their needs to meet different usage conditions and requests.

In this paper, we propose a model-based approach that provides users with the ability of rapidly developing, adapting and reconfiguring reactive service-based applications to meet new requirements and needs. Users specify their needs by describing sample executions that include interactions with web services through an intuitive interface. Interactions are stored in a visual formalism that integrates live sequence charts with graph transformation systems. Models can be visualized, modified, executed and automatically analyzed to identify inconsistencies.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications - languages, methodologies D.2.4 [Software Engineering]: Software/Program Verification - formal methods, validation D.2.6 [Software Engineering]: Programming Environments - graphical environments, interactive environments

General Terms: verification.

Keywords: visual languages, live sequence charts, graph transformations, service integration, automated analysis.

1. INTRODUCTION

People often interact with functionalities provided by modern Internet-based systems by invoking web services. For example, people order items from e-commerce systems, ask for weather forecast news, and search through documents available on the Web by invoking web services. Many useful applications can be built at the users' sites by composing available web services, for example, applications that identify the best route between locations by interacting with services that provide maps, traffic information and weather forecasts [7].

Despite the availability of engineering processes well suited for developing service-based applications [8, 1], often people integrate web services manually, by invoking web services, extracting information from responses, transforming data formats, and generating requests to other web services. This happens because common users do not have enough skills to develop their own applications, and it is not cost effective for software experts to develop applications that satisfy needs of small sets or even individual users. Consequently, people spend a lot of time interacting with web services, often repeating the same patterns of actions.

In this paper, we present a requirement-driven iterative methodology for semi-automatically developing, adapting and reconfiguring *highly customizable reactive service-based applications*. The methodology enables expert users to quickly specify and develop service-based applications, and common users to adapt and reconfigure applications to meet emerging and evolving requirements that cannot be effectively managed with standard engineering processes [8, 1].

2. ITERATIVE DEVELOPMENT OF SOA-BASED ADAPTABLE APPLICATIONS

Figure 1 illustrates the methodology proposed in this paper to iteratively develop highly customizable reactive service-based applications. The methodology is composed of a development and an adaptation cycle. In the development cycle, expert users develop the initial interactive service-based applications. In the adaptation cycle, common users adapt and reconfigure the applications to meet emerging requirements and new needs. Development and adaptation cycles are based on the same technology, but while in the development cycle, users may need to interact with the underlying models, thus needing specific skills, in the adaptation cycle, users interact through a simple GUI that does not require specific skills.

Expert users start by selecting the services to be integrated in the application. Service interfaces are augmented with GT-based specifications, which describe the external behavior of the web service [3]. These descriptions can be provided either by web service developers or expert users. GT-based specifications simulate the behavior of the web services, thus allowing users to incrementally build the application by *playing* with web services. Expert users play-in sample executions that result in interactions between users,

[*]Mauro Pezzè is also at the University of Lugano.

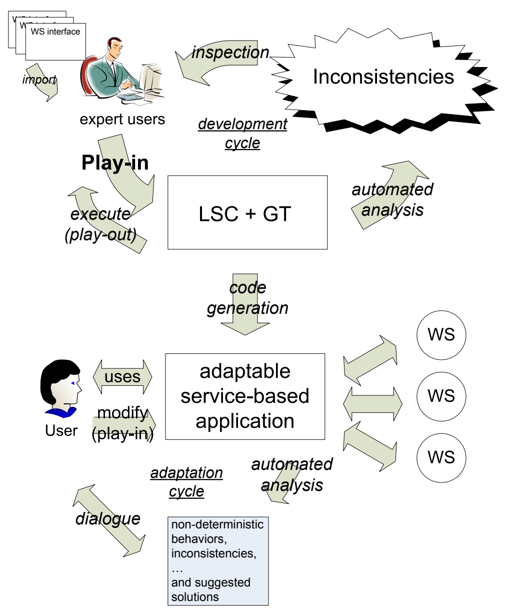

Figure 1: Development and adaptation cycles

application and web services. The technique merges the interactions that correspond to played-in executions, into an integrated model that describes the control flow by means of LSCs [2], and the data flow and data processing by means of GTs. The integrated model represents the behavior of the application.

Models are incrementally analyzed during the development cycle to reveal inconsistencies. The analysis framework identifies inconsistencies by cross checking LSC and GT models, and suggests modifications to the models to solve the identified inconsistencies. Expert users can either chose the most appropriate suggestion or manually modify the underlying models to eliminate the inconsistencies, and thus enable the execution of the specification.

Once the service interfaces are augmented with GT-based specifications and the core application has been developed, users do not need to access the underlying models, but can adapt and modify the application by incrementally adding, removing, or modifying sample behaviors through a visual interface. While during the development cycle, expert users may react to inconsistencies by either following the suggested solution, or manually modifying the underlying models, during the adaptation cycle, common users may react to inconsistencies by simply selecting from alternative suggestions proposed by the system through an intuitive interface, for example answering to questions like: "When C occurs, behavior A contradicts behavior B, do you want: (1) delete either A or B, (2) invert the call to A and B, . . . ".

Models that capture both interactions (LSC) and data transformations (GT) support not only execution, but also automatic code generation.

Our approach leverages Harel's play-in/play-out approach [4], which supports playing-in of LSCs to generate executable requirements.

3. INTEGRATED MODEL

Our integrated model represents the current state of the system by means of a disjoined set of graphs (a single graph can represent the state of either the application under development or a web service), the flow of messages exchanged between user, application, and web services by means of LSCs, and the changes on the system state induced by sent and received messages by means of GT rules. Every time a LSC is traversed, each message in the LSC triggers the application of the GT rule associated with the message. For example, the reception of the message `setRoute(Route)` specified in the LSC shown in Figure 2 triggers the execution of the corresponding GT rule shown in Figure 3.

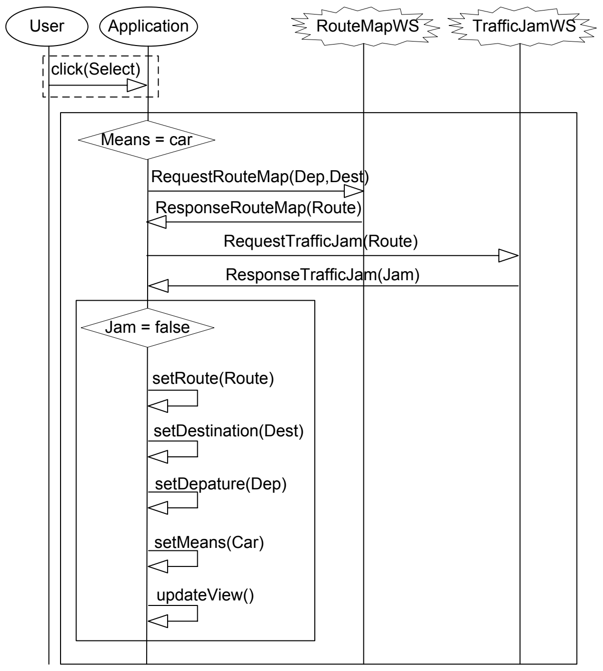

Figure 2: An example LSC that specifies the control flow of messages exchanged when users search a route connecting two places by considering traffic information. The dotted box indicates the pre-chart.

Since GT rules specify how the operations modify the system state, they can be intuitively interpreted as *contracts* that specify how LSCs can legally invoke these operations. The LSC-based specification indicates the behavior that users expect from the application under development. Discrepancies between GT- and LSC-based specifications indicate system inconsistencies, for instance, the existence of LSCs that activate illegal sequences of graph transformations. Such discrepancies can be automatically identified and reported to the users together with suggestions to remove the inconsistencies.

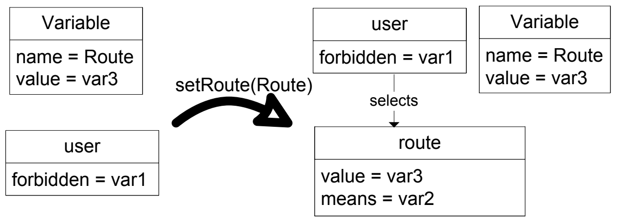

Figure 3: An example of graph transformation rule

Note that the state graph of a web service does not represent the current concrete state of the web service but the conceptual state of the web service as expected by the application under development. If operations provided by a web service are specified with GT rules, our framework can automatically trace the conceptual state of web services and automatically diagnose if a client application inconsistently uses web services. If this information is not available, our framework can automatically analyze the consistency of the behavior of a client application, but cannot check if the client application consistently uses web services.

When system messages are sent and received, GT rules are used to update the system state. The rule that is applied when a message is sent specifies the changes that need to be applied on the state of the sender, e.g., an attribute value can be increased to count the number of sent messages. The rule that is applied when a message is received depends on the nature of the message. If the message goes from the application to a web service, the rule corresponds to the one that specifies the requested operation. If the message goes from the web service to the application, the rule consists of transferring the return value generated by a web service to the application. If the message goes from the application or the user to the application, it consists of a computation that is internal to the application. The behavior instilled into the application under development is given by the message sequences generated by LSCs that do not violate constraints imposed by GTs.

4. INTEGRATED ANALYSIS

Specifications may be inconsistent. For instance, users can specify that objects of a given type must always be initialized before being used, and may define some scenarios where these objects are used without being initialized. Inconsistencies can be easily introduced into integrated specifications because users are usually concentrated on the design of single LSCs or GTs, without considering all possible interplays with other GTs, LSCs and combinations of them.

Since GTs specify the semantics of single operations and users organize these operations within LSCs, we can intuitively assume that the behavior that the user requires from the system is given by executions described by means of LSCs only. However, GTs further constrain the behavior of the system. The gap between the behavior required by users and the behavior exhibited by the system under development is given by the set of executions that are accepted by LSCs but violate some GT rules. Our analysis signals these executions to the users to warn them about system inconsistencies.

Analysis of Single LSCs. The Analysis of single LSCs identifies executions generated by LSCs that do not belong to the final system because of the constraints imposed by GTs. We can identify two classes of problems: errors and warnings.

An LSC is said to be *erroneous* if it never produces a feasible behavior. For instance, an LSC that removes all user *admin* from a system and then performs an action that requires the *admin* access rights to be completed cannot ever be successfully completed. An LSC is said to include *warnings* if it includes sequences of operations that may generate conflicts. For instance, consider an LSC that includes a message that removes an object with a given identifier and a message that modifies the same object. If the *modify* message can be sent after the *delete message*, the system may enter an inconsistent state. Both erroneous and warnings in LSCs are presented to the users, who can decide if and how to react. Since the analysis techniques are based on the identification of the possible operation sequences generated by LSCs and conflicts and dependencies between GT rules, we first present support and then the analysis techniques.

Generation of Transformation Sequences Associated to LSC. Given a LSC, we can derive a Control-Flow Graph (CFG) that represents a (super-)set of the executions generated by the LSC. The translation of an LSC to a CFG is straightforward and consists of removing conditions from the LSC and suitably mapping constructs used in LSCs to constructs of CFGs. Features that are straightforwardly mapped from LSC to CFG are cold conditions, IfThenElse constructs, subcharts, nonderministic choices and loops. The set of executions described by a CFG may be larger than the ones represented in the corresponding LSC because CFGs abstract from conditions specified in LSCs and represent only the possible control flows.

Since the set of executions specified by a CFG can be infinite, we extract a finite set of behaviors that are analyzed. In particular, we define the *unfolding* of a CFG as the set of all possible message sequences that traverse the same node at most k times. Each message in the CFG is associated with a pair of GT rules that specifies how the system state is updated when the message is exchanged. Therefore, any path from the initial state of the CFG to its final state represents a different execution that can be mapped on the corresponding sequence of GT rules that must be applied on the system state.

Conflicts and Dependencies Between Rules. GT rules cannot be always applied in any order. In some cases, the application of a rule can be necessary to apply other rules, while in other cases the execution of a rule can disable the execution of other rules. For instance, removing a planned route, disable the modification of that route. GTs can be analyzed to identify two kinds of relations between rules: conflicts and dependencies. We say that rule g_1 may disable rule g_2 iff g_1 may delete/add state entities that are required/forbidden by g_2 (*conflict*). We say that rule g_1 may cause rule g_2 iff g_1 may delete/add state elements that are forbidden/required by g_2 (*sequential dependency*) [5].

Identification of Errors. An LSC is erroneous if no state that enables its successful execution exists. Searching for erroneous LSCs is limited by a parameter k which sets the depth of the unfolding.

For each sequence of messages identified by the unfolding, we try to compute the concurrent rule [6], which is a single rule that is equivalent to the whole sequence. If it is not possible to construct the concurrent rule for any of the sequences in the unfolding, the LSC is erroneous. Note that it is necessary to compute the concurrent rule for all sequences in the unfolding only if the considered LSC is erroneous; if the LCS is not erroneous, the analysis terminates when the first concurrent rule is determined. Since erroneous LSCs seldom occur, the analysis for erroneous LCS is usually fast.

Identification of Warnings. In principle, to identify all the possible inconsistencies, we should compute the concurrent rules for all sequences. This strategy may be extremely expensive. To reduce the cost of the analysis, we exploited characteristics of GT rules to restrict the computation of the concurrent rule to *suspicious sequences* only. We eliminate from the analysis those sequences that are likely *non-suspicious*. A sequence of GT rules is *non-suspicious* if it satisfies three conditions: (1) it does not contain any conflicting operations; (2) if a rule requires satisfaction of a pre-condition, the previous rule in the sequence induces the satisfaction of the pre-condition; (3) no rule deletes nodes (this prevents inconsistent states). If a concurrent rule cannot be derived for a suspicious sequence, users receive a warning.

The analysis can be further optimized by using shift equivalence as described in [3]. Shift equivalence determines equivalent classes of GT rule sequences. Two sequences are in a same class if their application produces the same result and one can be obtained from the other by iteratively switching rules. This means that we can try to construct the concurrent rule for one sequence for each class, and avoid analysis of sequences that can be obtained by switching rules.

Analysis of Multiple LSCs. Multiple LSCs are analyzed by composing LSCs, that is considering the LSCs that can be activated by another LSC, and then analyzing the composition with techniques for analyzing single LSCs. Unfortunately, this approach may suffer from scalability problems.

In our case, we can benefit from working with reactive systems, that is systems in which executions are triggered by user interactions. This is an important information because any execution can only be triggered by user inputs. In our integrated model, user inputs correspond to the messages from the user to the application. Therefore, we can limit the analysis of multiple LSCs to composed executions that start from an LSC with a pre-chart (which is the triggering condition of an LSC) that exclusively includes user inputs, and consider the LSCs that can be recursively activated, instead of considering the composition of all LSCs. The analysis of single LSCs can be applied to multiple LSCs by extending the unfolding process to the activation of other LSCs. According to our early experience reported in the next section, this optimization makes the analysis feasible without loss of information.

Correction Mechanisms. When either a warning or error is identified, our technique automatically suggests possible corrective actions to users. Candidate solutions are identified according to the position of the rules that prevent construction of the concurrent rule. The technique automatically generates new sequences by switching and deleting rules that generate conflicts, or adding rules to remove conflicts. Automatically generated solutions are verified by first building the corresponding concurrent rule, and then verifying the whole model.

5. EARLY VALIDATION

We gained early experience with our technique, by designing and adapting a highly-customizable reactive service-based application: The Personal Mobility Manager (PMM), a reactive service-based application designed to satisfy requirements related to individual user mobility [7]. PMM is obtained as the integration of five web services which provide maps, temperature values, traffic information, weather forecast news and authentication capabilities. We specified functionalities of the target system and we evaluated the quality of the collected feedbacks, the cost-effectiveness of the analysis and the flexibility of the final result.

Since LSCs can specify several alternative scenarios, and GT-rules are re-used across different LSCs, the specification is compact and manageable. During system construction, we continuously checked the specification to find inconsistencies. *Single LSCs* are seldom inconsistent, because users can easily manage scenarios of limited size. However, we found several inconsistencies due to incremental modifications to single LSCs. On the contrary, when the size of the system grows, *multiple LSCs* are often inconsistent.

We evaluated the cost-effectiveness of the analysis technique by measuring the amount of behaviors generated with the unfolding process and the behaviors that needed to be analyzed to identify problems. Combining the identification of suspicious sequences and switch equivalence reduced the number of sequences to be analyzed due to single LSCs from 14 to 1, and the number of sequences to be analyzed due to multiple LSCs from 84 to 2. The combination of the two techniques is particularly efficient because the selection of suspicious sequences is excellent in identifying executions with no issues, while switch equivalence is excellent in identifying executions with equivalent issues. These techniques largely balanced the extra behaviors generated by unfolding.

Finally, we can say that the use of a repository with the many scenarios represented as the integration of LSCs and GTs enables the modification of the system specification in a short amount of time. Moreover, any updated configuration can be immediately verified. On the contrary, the coded version of the PMM requires extensive coding and design effort to be adapted to new requirements.

Acknowledgments. This work has been supported by MIUR under the project PRIN 2006-2007 “Mutant hardware/software components for dynamically reconfigurable distributed systems” (COMMUTA),

6. REFERENCES

[1] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology*, 15(4):360–409, 2006.

[2] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[3] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1999.

[4] D. Harel and R. Marelly. *Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[5] J. Hausmann, R. Heckel, and G. Taentzer. Detecting conflicting functional requirements in a use case driven approach: a static analysis technique based on graph transformation. In *International Conference on Software Engineering*, 2002.

[6] L. Lambers, H. Ehrig, F. Orejas, and U. Prange. Adhesive high-level replacement systems with negative application conditions. In *Workshop on Applied and Computational Category Theory*. Electronic Communications of the EASST, 2007.

[7] D. Lorenzoli, S. Mussino, M. Pezzè, D. Schilling, A. Sichel, and D. Tosi. A soa-based self-adaptive personal mobility manager. In *IEEE Conference on Service Computing*, 2006.

[8] T. N. Nguyen. Model-based version and configuration management for a web engineering lifecycle. In *15th international conference on World Wide Web*, 2006.