A Formal Framework for Developing Adaptable Service-Based Applications

Leen Lambers[1], Leonardo Mariani[2], Hartmut Ehrig[1], and Mauro Pezzè[2]

[1] Department of Software Engineering and Theoretical Informatics
Technical University Berlin
Franklinstrasse, 28/29 - 10587 Berlin
{leen,ehrig}@cs.tu-berlin.de
[2] Department of Informatics, Systems and Communication
University of Milano Bicocca
via Bicocca degli Arcimboldi, 8 - 20126 Milano
{mariani,pezze}@disco.unimib.it

Abstract. Web services are open, interoperable, easy to integrate and reuse, and are extensively used in many application domains. Research and best practices have produced excellent support for developing large-scale web-based applications implementing complex business processes.
Flexibility and interoperability of web services make them well suited also for *highly-customizable reactive service-based applications*, that is interactive applications which serve few users, and can be rapidly adapted to new requirements and environmental conditions. This is the case, for example of personal data managers tailored to the needs of few specific users who want to adapt them to different conditions and requests. Classic development approaches that require experts of web service technologies do not well support this class of applications which call for rapid individual customization and adaptation by non-expert users.
In this paper, we present the formal framework of a model-based approach that provides expert users with the ability of rapidly building, adapting and reconfiguring reactive service-based applications according to new requirements and needs. Moreover this formal approach will presumably allow adaptations and reconfigurations by non-expert users as well. The underlying technique integrates two user-friendly, visual and executable formalisms: live sequence charts, to describe control flow, and graph transformation systems, to describe data flow and processing. Main results of the paper are the specification and semantics of the integration and early analysis techniques revealing inconsistencies.

1 Introduction

Internet-based systems often expose functionality through publicly available web services. The many available services provide to end-users the interesting opportunity to satisfy their emerging needs by implementing and executing client applications that integrate these services. For instance, users can easily obtain applications managing personal mobility by integrating web services that provide maps, traffic information and weather forecasts [1].

Unfortunately, end-users often do not have enough skill to develop such client applications and software experts are usually not interested in developing systems that satisfy needs of few or even single users. Thus, notwithstanding the many available web services and the existence of well-suited engineering processes [2, 3], personal service-based applications seldom exist. In practice, users spend large amount of time by manually repeating interactions with web services.

In this paper, we present the formal foundations of a requirement-driven iterative methodology, early described in [4], 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 [2, 3]. Our technique merges interactions which should be adopted by the applications into an integrated model that describes the control flow by means of LSCs [5], and the data flow and data processing by means of GTs [6, 7]. The integrated model represents the behavior of the application and can be automatically executed and analyzed. The sample interactions can be specified by expert users through a visual and intuitive interface analogous to the Play-In approach [5]. Moreover this interface presents analysis results and automatically generates solutions to inconsistencies in the integrated model.

This paper formally describes the integration of LSCs with GTs and early proposes integrated analysis techniques of the integrated model to identify inconsistencies and errors. We validated the technique by specifying a Personal Mobility Manager (PMM), which is a reactive service-based application designed to satisfy requirements related to individual user mobility [1]. The paper is structured as follows. Section 2 provides background information about LSCs and GTs. Sections 3 and 4 present our formal integrated models and automated analysis techniques, respectively. Finally Section 5 discusses related work and summarizes the main contributions of this paper.

2 LSC and GT

In this Section, we introduce the two modeling languages, LSCs and GTs, that we integrated to specify and generate adaptable service-based applications.

Live Sequence Charts LSCs have been introduced in [5] as an extension of message sequence charts to express liveness, i.e., the existence of possible, necessary and mandatory behaviors, both globally to a collection of charts and locally to single charts. LSCs can be universal and existential. A universal chart consists of a prechart and a main chart, if an execution satisfies a pre-chart it must also satisfy the main chart. Existential charts are charts that need to be satisfied by all runs.

Example 1. An example of a universal LSC is given in Figure 1 (a). This LSC specifies the control flow of messages that are exchanged in the PMM when users identify a route connecting two places by taking traffic intensity into consideration. LSCs can contain guards that indicate conditions under which execution is suspended, e.g., the LSC in Figure 1 (a) requires *Means.name* = *car* to be fullfilled.

LSCs can have local variables. For instance, the LSC in Figure 1 (a) includes local variables created by instantiation to transmit values to web services (e.g., variable *Dep* in Figure 1 (a)) or to store return values sent (e.g., variable *Route* in Figure 1 (a)). To cope with the complex structure of the state of object-oriented programs, we associate each LSC with a local graph that represents local variables. The existence of local graphs support the use of GT rules to create/modify/read local variables.

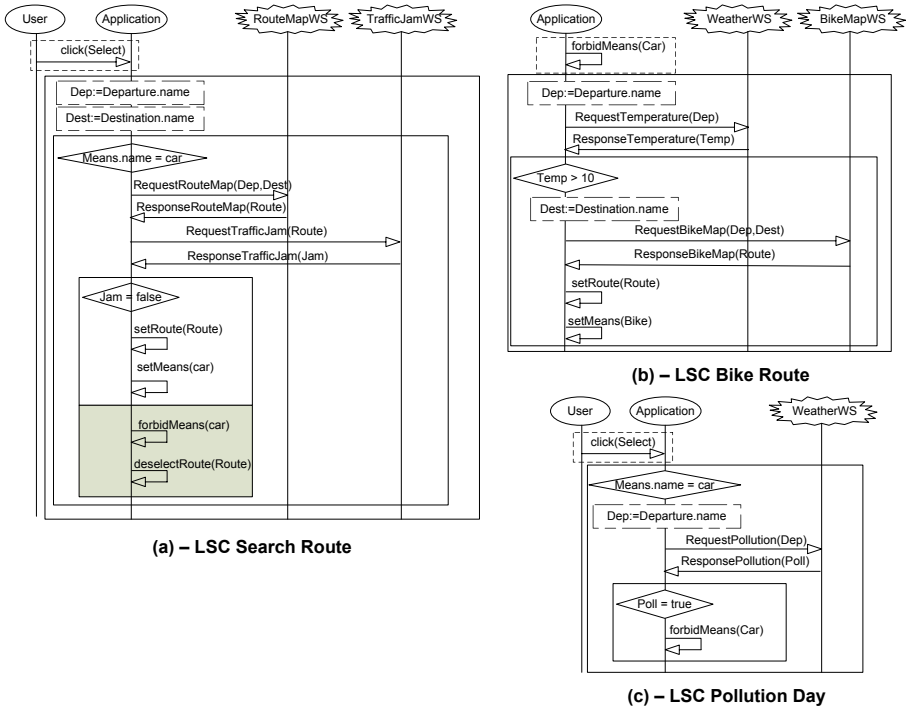

Fig. 1. Example LSCs. The dotted boxes indicate the pre-charts

Graph Transformations In our integrated model, possible state changes and computations are specified with a set of graph transformation rules. A rule consists of a left and right hand side (lhs and rhs resp.), both typed over a class diagram of the application. The lhs indicates the objects, connections and attribute values that must be present before applying a rule. The rhs indicates the result of the application of the rule. Objects and connections that are present in both the left and right hand sides are preserved by the application of the rule. Objects and connections that are present in the lhs and not in the rhs are deleted by the application of the rule. Objects and connections that occur only in the rhs are added. If attribute values differ between lhs and rhs, the application of the GT rule modifies their values. Rules can include a negative application condition (NAC) that describes which objects, connections or attribute values are forbidden be-

fore applying the rule [8]. A detailed and formal description of typed, attributed graphs and graph transformation can be found in [6, 7].

Example 2. Figure 2 shows a few examples of graph transformation rules. The rule *setRoute(Route)* creates a *Route* object and preserves the existence of the Route *Variable*; rule *setMeans(car)* modifies the value of attribute *means* and rule *forbidMeans(car)* sets the forbidden attribute of the user to *car*. The NAC of rule *setMeans(car)* specifies that it cannot be applied if the user's attribute *forbidden* is set to car.

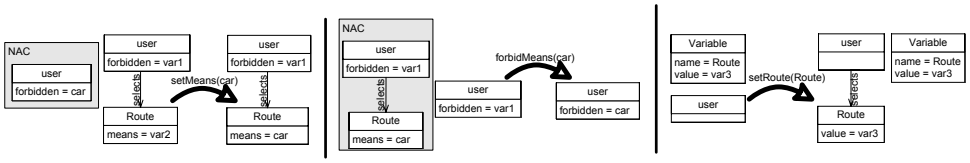

Fig. 2. Example of graph transformation rules

3 First Result: Integrating LSC and GT

We specify highly customizable reactive service-based applications by integrating LSCs and GTs: a disjoined set of graphs represents the current state of the system (a single graph can represent the state of either the application under development or a web service); LSCs describe the flow of messages exchanged between the user, the application, and web services (user inputs as the *click(Select)* message shown in the LSC in Fig. 1 (a) are modeled with messages from the user to the application); and GT rules indicate the changes on the system state induced by sent and received messages. The idea is that 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 Fig. 1 (a) triggers the execution of the corresponding GT rule shown in Fig. 2.

Definition 1 (system participants, system state, initial system state). *We represent the* participants *to a highly customizable reactive service-based application as a tuple* $P = (A, WS_1, \ldots, WS_n)$*, where* A *is the application under development and* $WS_1, \ldots, WS_n$ *are the web services accessed by* A*. The application and web services have a current state that evolves with system execution. We represent this state as a typed attributed graph and we define a mapping function* $gstate$ *to indicate the current state of a participant, e.g.,* $gstate(A)$ *is the current state of the application. The* system state *is indicated with the tuple* $gstate(P) = (gstate(A), gstate(WS_1), \ldots, gstate(WS_n))$*. States are typed over* type graphs*. We indicate the type graph associated to a participant with a mapping function TG, e.g.,* $TG(A)$ *is the type graph of the application. It is required that the state of a participant matches its type graph, e.g.,* $gstate(A)$ *is typed over* $TG(A)$*. The initial state of each participant is indicated with a mapping function*

$gstate_0$*, e.g.,* $gstate_0(A)$ *is the initial state of the application. The* initial system state *is indicated with the tuple* $gstate(P) = (gstate(A), gstate(WS_1), \ldots, gstate(WS_n))$.

Informally speaking, a typed attributed graph can be compared to a UML Object diagram and its type graph can be compared to a UML Class diagram [9].

Example 3. For the PMM $P = (Application, TrafficJamWS, RouteMapWS, WeatherWS)$. Fig. 3 shows an example state $gstate(P)$ for the PMM. The initial state $gstate_0(P)$ is obtained by leaving out the dotted nodes and edges. The web service areas indicate the visible behavior of web services (and eventually their conceptual state); the application area indicates the state of the application.

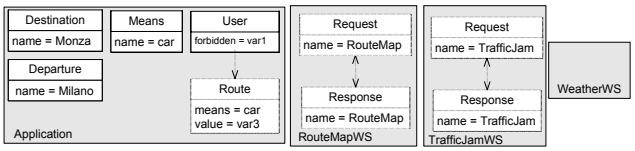

Fig. 3. An example state for the PMM system.

Note that the state graph of a web service WS_i does not represent the current concrete state of the web service but the conceptual state of the web service as expected by the application A. 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 the application A inconsistently uses web services. If this information is not available, our framework can automatically analyze the consistency of the behavior of A, but cannot check if A consistently uses web services. GT-based specifications can be obtained from UML design documents with limited additional effort [10]. Service providers that want to ease integration of their web services can consider providing this additional description.

Operational Semantics for GT Specification Now we define the GT based system specification and its associated semantics. Since GT rules specify how the system state is modified by each operation, they can be intuitively interpreted as a sort of *contract* that specifies how LSCs can legally invoke these operations. GT rules are usually specified by web service providers, while operations performed by the application are derived from user specifications (play-in).

When system messages are sent and received, GT rules are used to update the system state (at the end of this section we discuss how the matching process is guaranteed of being deterministic). 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. This rule is seldom used and often consists of the empty rule. 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. In the following paragraphs we describe how to formalize these ideas. Our formalization extends the one of Harels'play-in/play-out approach [11].

Definition 2 (system message, system alphabet). *Formally, a* system message M_s *is described by a tuple* $M_s = \langle Src, Dst, (r_1, r_2), sync \rangle$*, where* Src *and* Dst *is any between* $\{U, A, WS_1, \ldots, WS_n\}$ *(*U *indicates the user of the system),* (r_1, r_2) *is a pair of rules such that* r_1 *(resp.* r_2*) is typed over TG*(Src) *(resp. TG*(Dst)*) and* $sync$ *is a Boolean variable describing if the message is synchronous (*$sync = true$*) or asynchronous (*$sync = false$*). The set of all possible system messages, namely the* system alphabet*, is indicated with* Σ*.*

Given the set of messages that can be used to specify the behavior of a highly-customizable reactive service-based system, we can derive the list of message sequences that can be legally executed. This set is clearly infinite and our technique does not compute it in practice; however, its formal definition is useful to automatically identify inconsistent uses of existing operations.

Definition 3 (GT system specification and semantics). *A* GT system specification *is a tuple* (P, Σ) *where* P *is a set of participants and* Σ *a system alphabet for* P*. Given a GT system specification GTSPEC* $= (P, \Sigma)$*,* $Sem(GTSPEC)$ *consists of a set of transition systems that describe the sequence of transformation rules that can be executed by each participant in* P*. In particular,* $Sem(GTSPEC) = (Sem(A), Sem(WS_1), \ldots, Sem(WS_n))$ *such that* $Sem(X) = (\mathcal{G}_X, R(X), \Rightarrow, gstate_0(X))$ *is a state transition system with* X *any of* $A, WS_1, \ldots, WS_n$*,* $\mathcal{G}_X$ *possible state values for* X*,* $R(X)$ *the GT rules occuring in the system alphabet that are typed over TG*(X)*,* $gstate_0(X)$ *the initial state of* X *and* $\Rightarrow \subseteq \mathcal{G}_X \times R(X) \times \mathcal{G}_X$ *with* $(G, r, G') \in \Rightarrow$ *if and only if the rule* $r \in R(X)$ *can be applied to G such that* $G \overset{r}{\Rightarrow} G'$*.*

Example 4. The rules depicted in Fig. 2 belong to $R(Application)$ because they only change the state of the application. The system reaches the state shown in Fig. 3 from the initial state by applying as last rules *setRoute(Route)* and *setMeans(car)* from Fig. 2. This state is reached after running completely through the LSC in Fig. 1 (a) following the if-branch.

Operational Semantics of the Integrated Model Hereafter we present the semantics associated with the LSC-based specification, which is automatically obtained from scenarios provided (played-in) by users. This semantics indicates the behavior that users expect from the application under development. Differences between GT- and LSC-based semantics are used to automatically discover important inconsistencies, e.g., the existence of LSCs that activate illegal sequences of graph transformations. Moreover, we formally define the semantics of the integrated model, which specifies the behaviors that are instilled in the application under development.

Scenarios played-in by users describe a control flow of messages exchanged between participants of a highly customizable reactive service-based application. These

scenarios are formally specified as LSCs. Each message included in LSCs is associated with a sender, a receiver, a system message and a location that defines at which time the message should be sent and received [11].

Definition 4 (LSC message). *A LSC message M_l included in a LSC L is a triple $M_l = (I_{src}, I_{dst}, M_s)$, where I_{src} and I_{dst} represent the source and destination lifelines [11] with I_{src} and I_{dst} participants of the system P, and M_s a system message.*

Example 5. The LSC message *setMeans(car)* shown in Fig. 1 (a) has the Application lifeline as both source and destination; its system message has an empty source rule and a destination rule that corresponds to rule *setMeans(car)* shown in Fig. 2.

LSCs describe the behavior that users expect from the application under development. We formally define the behaviors generated by a *LSCSPEC* as in [11].

Definition 5 (LSC specification and semantics). *The complete set of LSCs that are specified by users form a* LSC system specification *LSCSPEC* $= (P, \Sigma, \mathcal{S_U})$ *with $\mathcal{S_U}$ a set of universal LSCs. The* operational semantics for *LSCSPEC is defined as a state transition system Sem(LSCSPEC) $= \langle \mathcal{V}, V_0, \Delta \rangle$ where $\mathcal{V}$ is the set of possible states of Sem(LSCSPEC), V_0 is the initial configuration and $\Delta \subseteq \mathcal{V} \times (\mathcal{E} \cup \bigcup_{L \in LSCSPEC} E_L) \times \mathcal{V}$ the set of allowed transitions with E_L the set of events for each LSC L. A state $V \in \mathcal{V}$ is defined as $V = \langle \mathcal{RL}, Violating \rangle$ where $\mathcal{RL}$ is the set of currently running LSCs and $Violating$ indicates by True or False whether the state is a violating one. The initial configuration is $V_0 = \langle \phi, False \rangle$.*

If we remove behaviors that are specified in LSCs and do not satisfy the constraints imposed by GT rules, we obtain the set of behaviors that are instilled into a highly-customizable reactive service-based system. In the following therefore we define how both specification techniques interplay on the syntactical and semantical levels. We therefore define an integrated system specification and its semantics by runs allowed by the specification. Informally, a run includes a sequence of observable events occurring when running the system.

Definition 6 (integrated system specification). *A* LSC L is compatible with a GT system specification $GTSPEC = (P, \Sigma)$ *if for each message $M_l = (I_{src}, I_{dst}, M_s)$ in L such that $M_s = (Src, Dst, (r_1, r_2), sync)$ it holds that $Src = I_{src}$ and $Dst = I_{dst}$. An* integrated system specification *LGTSPEC* $= (GTSPEC, LSCSPEC)$ *consists of a GT system specification GTSPEC* $= (P, \Sigma)$ *and a LSC specification LSCSPEC* $= (P, \Sigma, \mathcal{L})$ *such that every LSC $L \in \mathcal{L}$ is compatible with GTSPEC.*

The play-in GUI in which the expert users instill sample interactions allows creation of compatible specifications only.

Definition 7 (run consistent with LSC specification). *Given a LSC system specification $(S, \Sigma, \mathcal{L})$, a* run r *represents a system execution and is defined as a sequence of visible events $e_1 \dots e_m$, with $e_j \in M_L \times \{Send, Recv\}$, $L \in \mathcal{L}$ and M_L a LSC message. A* run r is consistent with *LSCSPEC if, starting from V_0, the rules of Δ can be applied iteratively to the events in r, and to the hidden events generated between them, without reaching a violating transition.*

Example 6. An example run for the LSC shown in Fig. 1 (a) is *(Click(Select),Send)(Click(Select),Rcv)...(setMeans(car),Send)(setMeans(car),Recv)*. This would be the begining and end of a consistent run with *LSCSPEC* for the case that the user has chosen to take his car and there is no traffic jam.

A run can be mapped to the corresponding set of system-level events. Thus, we can preserve the sequence of events removing information about lifelines, which are useful at the specification level, but not necessary at this point. Given a GT system specification *GTSPEC*, a *system run* is consistent with *GTSPEC* if the sequence of operations executed by each system participant is accepted by its GT-spec. Note that in the following definition we use the notation $part$ (resp. gt) to indicate a system participant (resp. rule) associated with the event.

Definition 8 (system run consistent with GT specification). *In particular, given a run* $r = e_1 \dots e_m$*, with* $e_j = ((I^j_{Src}, I^j_{Dst}, M^j_s), x^j)$*, its associated* system run *is* $r_S = e_{1,S} \dots e_{m,S}$ *with* $e_{j,S} = (M^j_s, x^j) \in \Sigma \times \{Send, Recv\}$*. Given GTSPEC* $= (P, \Sigma)$*, a system* run $r_S = e_{1,S}, e_{2,S}, \dots, e_{m,S}$ is consistent with *GTSPEC if for each participant* X *in* P*, there exists both a sequence of GT rules* $s_X = s_0 s_1 \dots s_n \in Sem(X)$ *with* $s_i \in R(X)$ *and an isomorphic mapping* map_X *assigning to each event* $e_{i,S}$ *in* r_S *such that* $part(e_{i,S}) = X$ *a unique rule* s_j *in the transition sequence* s_X*, i.e.,* $map_X : \{e_{i,S} | i \in \{1, \dots, m\}, part(e_{i,S}) = X\} \to \{1, \dots, n\}$*. Moreover, the mapping* map_X *must satisfy the following properties:* compatibility with gt*, i.e., if* $map_X(e_{i,S}) = j \Rightarrow gt(e_{i,S}) = s_j$*, and* preserve ordering of events*, i.e., if* $i \leq j$ *and* $part(e_{i,S}) = part(e_{j,S}) = X \Rightarrow map_X(e_{i,S}) \leq map_X(e_{j,S})$.

Example 7. Ax example consistent run, corresponding to the case of a user taking a car when a traffic jam exists, for the LSC shown in Fig. 1 (a) is *(Click(Select),Send) (Click(Select),Rcv)...(deselectRoute(Route),Send)(deselectRoute(Route),Recv)*. The corresponding system run would be inconsistent with *GTSPEC* since rule *deselectRoute(Route)* cannot be applied. The reason is that *Route* has never been selected, thus it is impossible to deselect it. This is formally expressed by the sequential dependence of rule *deselectRoute(Route)* from *selectRoute(Route)*. These kinds of inconsistencies are automatically detected by analysis techniques presented in the next section.

Definition 9 (semantics of integrated specification). *Given an integrated system specification LGT* $= (GTSPEC, LSCSPEC)$*, a* run r is consistent with *LGT if its associated system run* r_S *is consistent with GTSPEC and* r *is consistent with LSCSPEC. Given a system specification LGT* $= (GTSPEC, LSCSPEC)$*, then* $Sem(LGT)$ *consists of all LGT consistent runs.*

Play-Out We now describe how an integrated specification can be executed. The process of executing traces is called *play-out*. A trace executed by an integrated specification *LGT* is not necessarily a *LGT* consistent run. In particular, given a system event e, the event is played-out executing the following steps (we call this sequence of steps $gtstep(e)$, in contrast with the original definition in [11] which was called $step(e)$).

gtstep(e)

1. Apply $\Delta(e)$[3].
2. Change the system state according to the system event e:
   (a) if ($e = \langle M_s, Sent\rangle$ and $M_s = \langle Src, Dst, (r_1, r_2), false\rangle$ apply the rule r_1 to $gstate(Src)$
   (b) else if ($e = \langle M_s, Recv\rangle$ and $M_s = \langle Src, Dst, (r_1, r_2), false\rangle$) apply the rule r_2 to Dst
   (c) else if $M_s = \langle Src, Dst, (r_1, r_2), true\rangle$ apply r_1 to $gstate(Src)$ and r_2 to $gstate(Dst)$ in parallel

When playing out a certain system event it is possible that a transition system $Sem(\textit{LSCSPEC})$ reaches a Violating state as defined in [11] or that one of the transition systems in $Sem(\textit{GTSPEC})$ cannot be propagated. This happens when we execute an inconsistent LGT run/trace. Finding out and/or foreseeing inconsistent runs is the subject of section 4, which presents analysis techniques for this problem.

Matching of rules The process of selecting the instances where the rule must be applied is known as matching. Formally this match is expressed by a suitable morphism of the lhs of a rule into the state graph G. In general, we can have three main possibilities: (1) no matching, (2) single matching and (3) multiple matchings. No matching corresponds to the impossibility of finding any matching (and thus the rule cannot be applied to the current state). Single matching represents the existence of a unique match (and thus the rule can be applied in a unique way). Multiple matching corresponds to many possible matchings between the rule and the current state (and thus the rule can be applied in several ways). For instance, if the current state of an application includes multiple active users, the rule *setRoute(Route)* shown in Figure 2 can be applied to any of the users.

In our approach we assume deterministic matching. Thus the system specification should forbid multiple matching. To guarantee that this property is fulfilled, we restrict the possible structure of the disjoint graphs in S to type graphs with cardinalities that satisfy the following properties: (1) only one copy of any type of object can be instantiated, i.e. the cardinality of each type is minimal zero and maximal 1; (2) if multiple instances of a certain type are allowed, we require that all rules that affect this object type either use an identifier to select the exact instance or select all instances; (3) cardinality of each edge is between 0 and 1.

4 Integrated Analysis

System specifications instilled by expert users in the Play-In GUI or specification reconfigurations demanded by users may be inconsistent. For instance, in the *GTSPEC* an object trip must always be initialized before it can be populated with routes, and simultaneously, in the *LSCSPEC*, a scenario is present where routes are added to a trip

[3] $\Delta(e)$ indicates that the event is initially processed as in the original Harel et al.'s approach

without initializing it. Such kind of inconsistencies can be easily introduced into an integrated specification because during system design or reconfiguration the expert user (or user even less) is usually concentrated on a single LSC or GT, without considering all possible interplays with other GTs, LSCs and combinations of them.

Due to Play-In of the system behavior single operations are organized within LSCs and therefore we we can intuitively assume that the required behavior from the system is given by all consistent runs in $Sem(\textit{LSCSPEC})$. However, these single operations are represented by *GTSPEC* and further constraint the behavior of the system, since it is not always possible to execute all single operations in any order. The gap between the behavior intended by expert and end-users and the behavior exhibited by the system under development is given by the set *Suppressed Runs* of runs consistent with all LSCs but not consistent with the integrated system specification due to the effect of single operations. Such a suppressed run is presented in Example 7. Our analysis points out runs that are in *SR* in order to make sure that in the end expert users (or users) are aware of runs that are automatically suppressed. If this suppression is not desired it can be repaired and the reconfigured behavior can be instilled into the specification.

Definition 10 (suppressed runs).
$\textit{SR} = \{r | r \; consistent \; with \; \textit{LSCSPEC}\} \setminus \{r | r_S \; consistent \; with \; \textit{GTSPEC}\}$

In the following, we present the analysis techniques that identify suppressed runs due to the interplay of GT operations with single LSCs and multiple LSCs. Problems due to *LSCSPEC* and *GTSPEC* only are addressed with analysis techniques specific for the single techniques [11, 7].

Second Result: Consistency Analysis of Single LSCs Analysis of single LSCs consists of the identification of suppressed runs due to the interplay of GT operations with single LSCs. The analysis techniques for revealing suppressed runs are based on both the identification of possible runs generated by LSCs and conflicts and dependencies between GT rules. Therefore at first we present these supporting techniques and thereafter the final analysis techniques.

Generation of Runs Associated to LSC Given an LSC l, we can derive a Control-Flow Graph (CFG), denoted with $cfg(l)$, that represents a (super-)set of the runs generated by l. Each edge in the CFG is labeled with the name of a message, thus a sample run can be obtained by following the path from the starting node to the ending node. The translation of an LSC to a CFG is straightforward and consists of removing conditions from the LSC and suitably mapping constructs used in LSC to constructs of CFG. Features that are straightforwardly mapped from LSC to CFG are conditions, IfThenElse constructs, subcharts and loops. It is subject of future work to consider in the analysis also hot and cold temperatures in the LSCs as introduced in [5]. Thus for now we don't distinguish between provisional (cold) and mandatory (hot) behavior. Since the set of runs specified by a CFG can be infinite because of the loops, we extract a finite set of runs to analyze by only considering runs that traverse a node in a CFG a tunable finite number of times. Note that runs described by a CFG may not always be runs generated by the corresponding LSC because CFGs abstract from conditions specified in LSCs.

This may generate false alarms that can be removed though by checking in the original LSC if the problematic run really occurs.

Definition 11 (LSC and GT unfolding). *Given an LSC l, we define $U_k(cfg(l))$ as the* LSC unfolding *consisting of all possible runs generated by l that traverse a same node in $cfg(l)$ at most k times. Given $U_k(cfg(l)) = \{ex_1, \ldots, ex_n\}$, where $ex_i = \langle M_{S,1}, \ldots, M_{S,n_i}\rangle$ is a sequence of system messages representing a path in the CFG, each ex_i can be mapped to the corresponding sequence of GT rules $seq_t(ex_i)$ by replacing each message $M_{S,j} = \langle Src_j, Dst_j, (r_{1,j}, r_{2,j}), sync_j\rangle$ in the original sequence with: $r_{1,j}, r_{2,j}$, if $sync = false$ or $r_{1,j} + r_{2,j}$, if $sync = true$[4]. Given an LSC l, we denote the set of all sequences of GT rules obtained from l by the* GT unfolding *$seq_t^k(l) = \{seq_t(ex) | ex \in U_k(l)\}$.*

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, if you remove a planned route, you are not allowed to modify that route anymore. A specification $GTSPEC$ 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*) [12].

Identification of Suppressed Runs In order to identify all suppressed runs in a single LSC l we should check which runs belonging to the LSC unfolding $U_k(cfg(l))$ correspond to sequences of GT rules in the GT unfolding $seq_t^k(l)$ which are inconsistent, i.e., that cannot be applied to the system state. In [13] criteria are formulated which ensure the applicability of GT rule sequences. If these criteria are fulfilled, the rule sequence is applicable. We call these sequences *non-suspicious*. Thus we only have to analyze rule sequences which do not fulfill the above-mentioned criteria and we call them *suspicious sequences*. The set of suppressed runs in an LSC is then a subset of the set of runs corresponding to suspicious sequences and thus we narrowed down the search analysis.

Definition 12. *[(non-)suspicious sequences]*
A sequence $ex_t = \langle r_1, \ldots, r_i, \ldots, r_j, \ldots, r_n\rangle \in seq_t^k(l)$ is defined to be non-suspicious *if it satisfies four criteria: (1) (initialization) the first rule is applicable on the initial system state; (2) $\nexists r_i$ that can eliminate nodes; (3) (no impeding predecessors) ex_t does not contain a rule r_i that may disable rule r_j and $i < j$; (4) (enabling predecessor) if a rule r_i is not applicable on the initial system state $\exists r_j$ which causes r_i and $j < i$. A sequence is defined to be* suspicious *if at least one of these criteria is not satisfied.*

The third rule guarantees the absence of conflicting operations in the sequence. The fourth rule guarantees that prerequisites for executing an operation are satisfied. The second rule guarantees that no unwanted dangling edges are created by the application

[4] the operator $+$ indicates parallel execution of two rules.

of a rule; informally speaking this is similar to guaranteeing that no undesired null references are generated in the state of an object-oriented system.

If a rule sequence is suspicious it has to be further investigated in order to find out if the corresponding run is really a suppressed one. This can be done by trying to construct a *concurrent rule* for the sequence. The concurrent rule of a rule sequence $\langle r_1, \ldots, r_n \rangle$ for rules with NACs is constructed as described in [14] (unique matching guarantees uniqueness of the concurrent rule). If the construction is valid, the run is consistent; if it is not the run will be a suppressed one. These identified suppressed runs can now be presented to the expert user (or user) who can decide (with means of an automatic correction mechanism as explained in the next paragraph) if it should be suppressed or it should be part of the intended behavior and should be repaired.

Note that the search for suppressed runs can be further optimized by using shift equivalence as described in [7] and explained also in [13]. 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 it is sufficient to analyze one sequence for each class, and avoid analysis of sequences that can be obtained by switching rules.

We can identify now two classes of problematic LSCs: erroneous LSCs and LSCs containing warnings. An LSC is said to be *erroneous* if it never produces any feasible behavior. For instance, an LSC that removes the user $admin$ from a system and then performs an action that requires the rights of an $admin$ to be completed cannot ever be successfully completed. An LSC includes *warnings* if it includes suppressed runs that should be investigated further, but others that are consistent. For instance, consider an LSC including a message that removes a route with a given identifier from a planned trip and a message that modifies the same route. If the second message can be exchanged after the first, this can be the reason for a suppressed run or warning.

Definition 13 (erroneous LSC, LSC with warnings). *An LSC l is erroneous if it only contains suppressed runs. This means that each suspicious sequence in the GT unfolding $seq_t^k(l)$ corresponds to a suppressed run in the LSC unfolding $U_k(cfg(l))$. A LSC l contains a* warning(s) *if a suspicious sequence(s) in $seq_t^k(l)$ corresponds to a suppressed run in $U_k(cfg(l))$, but at least one consistent run in $U_k(cfg(l))$ exists.*

Example 8. Our analysis technique signaled an inconsistency in the LSC in Fig. 1 (a) in the else branch to the condition *Jam=false*. There are two messages in this branch, one to forbid the use of the car and one to remove the selected route. However the route is selected only in the if branch, thus there is no route to remove. This run is suspicious because condition (2) of Def. 12 does not hold for the message *deselectRoute(Route)*. Since a concurrent rule cannot be determined, this run is inconsistent, thus a warning is generated.

Third Result: Consistency Analysis of Multiple LSCs Multiple LSCs can be analyzed by composing LSCs, i.e., 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 largely benefit from working with reactive systems, i.e., systems in which executions are triggered by user interactions. This is important information because any execution can only be triggered by user inputs. In our integrated model, user inputs coincide with messages going from the user to the application. Therefore, we can limit the analysis of multiple LSCs to composed executions that can be obtained by starting from an LSC with a pre-chart that exclusively includes user inputs and considers the LSCs that it can recursively activate, instead of considering the composition of all LSCs. The analysis of single LSCs can be directly applied to the analysis of multiple LSCs by extending the unfolding process to the activation of other LSCs. According to the experience reported at the end of this section, this optimization makes the analysis feasible without loss of information.

Example 9. In case of pollution, the LSC shown in Fig. 1 (c) activates the LSC in Fig. 1 (b), thus bike is selected if the temperature is high enough. If there is also no traffic jam, the LSC in Fig. 1 (a) selects car as transportation mean. However, bike has been already selected and the run includes an inconsistency. The inconsistency is due to the application of two rules with conflicts, $forbidMeans(Car)$ and $setMeans(Car)$. In fact, a concurrent rule expressing the application of both runs cannot be built.

Correction Mechanisms When either a warning or error is identified, our technique can automatically suggest possible corrective actions which can be presented in a suitable way also to end users. Candidate solutions are generated depending from the reason of the inconsistency. For example, if a sequence of GT rules $ex_t = \langle r_1, \ldots, r_n \rangle \in seq_t^k(lsc)$ includes rules r_i and r_j leading to the non-satisfaction of criterion 3 in Def.12 our technique automatically generates a new sequence ex'_t obtained by applying any of the following strategies: (1) switch r_i and r_j, (2) delete r_i or r_j and (3) add a message r, which overrides the effect of r_i, between r_i and r_j. If necessary, expert users can play-in tailored solutions to address conflicts between system messages. If now the new sequence ex'_t corresponds to a consistent run, our technique proposes the solution to the user. If the user accepts the solution, our system applies the corrective action applied on the sequence to the faulty LSC. Afterwards, we compute the unfolding of both the modified and the updated LSC and we present the set of behaviors that have been removed and added as a consequence of the modification to the user. Finally, to detect possible side effects of the change, the new version of the integrated model is analyzed to find eventual problems due to the interplay of multiple LSCs. If no problems are detected, the change is definitive.

Example 10. Our technique can automatically correct the problem presented in the Example 8 by deleting the message *deselectRoute(Route)*. A correction mechanism for the Example described in Example 9 could be to delete the rule $forbidMeans(Car)$ from LSC (c) in Fig. 1. Moreover the following tailored solution proposed by the expert user for the conflict described in Example 9 could be accepted by the end user as a possible reconfiguration. The rule $forbidMeans(car)$ is weakened to a rule $warning(pollution)$ which warns the user about pollution day instead of forbidding him to use his car.

Early Validation We used an integrated model to describe the PMM system [1] and we analyzed the model for consistency. Validation highlighted effectiveness of both switch equivalence and selection of suspicious sequences. In fact, single LSCs generated 8 sequences to be analyzed, which have been reduced to 1 by selection of suspicious sequences. This suspicious sequence is generated by the LSC in Figure 1 (a) and resulted to be inconsistent as described in Example 8 and corrected as described in Example 10. Thus, we verified again consistency of single LSCs and we analyzed multiple LCSs. Since we do not have complete tool support yet, we limited the analysis of multiple LSCs to pairs of LSCs. When analyzing multiple LSCs, the three LSCs in Figure 1 generated 194 sequences to be analyzed. Switch equivalence reduced the sequences from 194 to 5. Selection of suspicious sequences further reduced to 2 the sequences to be analyzed. These sequences resulted to be inconsistent. They are generated by the interplay of LSC (a) and LSC (b) as explained in Example 9. The use of both selection of suspicious sequences and switch equivalence demonstrated the scalability of the analysis technique.

5 Related Work and Conclusions

Other work suitably integrated multiple formalisms to describe aspects related to both data and interactions, even if not specifically designed to describe interactions in service-based applications. Harel et al. integrated LSCs with statecharts [15] in order to synthesize the behavior of a set of LSCs. Whittle et al. integrated sequence charts and OCL to synthesize into statecharts the behavior of component-based systems [16]. Statecharts describe state changes of objects, but, in contrast with graph transformations, they cannot specify how the object structure, i.e., the object graph, of an application changes. Therefore it is not possible to exploit the available analysis techniques for graph transformation to reveal inconsistencies. Finally, several work integrated UML-based dynamic models, e.g., activity and sequence diagrams with GTs in different application domains. FuJaBa integrates story diagrams (a kind of activity diagrams) with graph transformations to obtain a result close to the one presented in this paper [17]. Mehner et al. integrated activity diagrams and GTs to describe aspect-oriented software [18]. The former technique does not support an analysis similar to the one described in our approach. The latter supports consistency checks, but does not formally describe the operational semantics of the integrated specification, thus it does not support automated code generation. Moreover, activity and story diagrams do not allow to specify several features inherent to LSCs, such as cold/hot features and prechart/mainchart structure.

In this paper, we presented a formal framework for a model-based approach for the development of highly-customizable reactive service-based applications. The technique is based on the integration of the visual languages LSCs and GTs. Main results of this paper are the specification and semantics of the integrated model together with early analysis techniques revealing inconsistencies. Based on this formal framework it is possible to iteratively develop distributed service-based applications by incrementally modifying the specification, collecting feedbacks from the analysis of the specification and adapting the desired behavior as soon as individual user requirements change. The technique has been experienced with a service-based system for the management of

personal mobility information. The formal framework as presented in the paper has been worked out for this service-based application. The empirical experience provided confidence over the capability to manage realistic applications.

In the future, we aim to push forward the complete development of prototype tools enabling a comfortable play-in of the system behavior or system reconfigurations together with a suitable play-out (visualization + code generation) of the system behavior together with analysis results and corresponding correction mechanisms.

References

1. Lorenzoli, D., Mussino, S., Pezzè, M., Schilling, D., Sichel, A., Tosi, D.: A soa-based self-adaptive personal mobility manager. In: IEEE Conference on Service Computing. (2006)
2. Nguyen, T.N.: Model-based version and configuration management for a web engineering lifecycle. In: 15th international conference on World Wide Web. (2006)
3. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process modeling in web applications. ACM Transactions on Software Engineering and Methodology **15**(4) (2006) 360–409
4. Lambers, L., Ehrig, H., Mariani, L., Pezzè, M.: Iterative model-driven development of adaptable service-based applications. In: proceedings of the International Conference on Automated Software Engineering. (2007)
5. Damm, W., Harel, D.: Lscs: Breathing life into message sequence charts. Formal Methods in System Design **19**(1) (2001) 45–80
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer (2006)
7. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1. World Scientific (1999)
8. Annegret Habel, R.H., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
9. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An integrated semantics for uml class, object and state diagrams based on graph transformation. In: Third International Conference on Integrated Formal Methods. LNCS, Springer (2002)
10. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: a software engineering perspective. In: International Conference on Graph Transformation. Volume 2505 of LNCS., Springer (2002)
11. Harel, D., Marelly, R.: Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
12. Hausmann, J., Heckel, R., Taentzer, G.: 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)
13. Lambers, L., Ehirg, H., Taentzer, G.: How to ensure or eliminate the applicability of a sequence of rules. In: Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques, EASST (2008) submitted.
14. Lambers, L., Ehrig, H., Orejas, F., Prange, U.: Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions, ENTCS (2008) to appear.
15. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling. (2005) 309–324
16. Whittle, J., Schumann, J.: Generating statechart designs from scenarios (2000)
17. Fujaba Developer Team: Fujaba home page (2005) http://www.fujaba.de.
18. Mehner, K., Monga, M., Taentzer, G.: Interaction analysis in aspect-oriented models. In: proceedings of the IEEE International Requirements Engineering Conference. (2006)