Object Composition in Scenario-Based Programming [⋆]

Yoram Atir, David Harel, Asaf Kleinbort, and Shahar Maoz

The Weizmann Institute of Science, Rehovot, Israel
{yoram.atir,dharel,asaf.kleinbort,shahar.maoz}@weizmann.ac.il

Abstract. We investigate the classical notion of object composition in the framework of scenario-based specification and programming. We concentrate on live sequence charts (LSC), which extend the classical partial order semantics of sequence diagrams with universal/existential and must/may modalities. In order to tackle object composition, we extend the language with appropriate syntax and semantics that allow the specification and interpretation of scenario hierarchies – trees of scenarios – based on the object composition hierarchy in the underlying model. We then describe and implement a composition algorithm for scenario hierarchies, and discuss a trace-based semantics and operational semantics (play-out) for the extension. The extension has been fully implemented, and the ideas are demonstrated using a small example application.

1 Introduction

Building upon the preliminary (unpublished) work in [3], we integrate object composition with scenario-based specification and programming. Object composition, that is, the 'part-of' hierarchical relation, is a fundamental concept in object oriented analysis and design [5]. We consider strong composition, for which part-objects are intrinsically associated with their whole, and do not exist independently. Scenarios, depicted using variants of sequence diagrams, are popular means for specifying the inter-object behavior of reactive systems (see, e.g, [8, 12, 18, 20]), are included in the UML standard [19], and are supported by many modeling tools. To specify scenarios we use a UML2 compliant variant of *live sequence charts* (LSC) [7, 10], a visual formalism that extends classical message sequence charts (MSC) [13], mainly by making a distinction between possible and mandatory behavior. The LSC language has an executable (operational) semantics (play-out) [11], and thus may be used not only for requirements and specification but also as a programming language.

We define an appropriate extension of the syntax and semantics of LSC, which allows the specification and interpretation of a scenarios hierarchy that

[⋆] This research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science and by a Grant from the G.I.F., the German-Israeli Foundation for Scientific Research and Development.

is based on the object composition hierarchies in the model. Combining object composition with scenario-based specifications supports information hiding together with a scalable and decentralized design, where high level scenarios are refined by specifying the inner behavior of their participating objects modulo the scenario-based context.

The main mechanism we introduce is *LSC-trees*; hierarchies of modal scenarios induced by the system model's object composition hierarchy. LSC-trees are created by allowing lifelines to be decomposed into *part-scenarios*. Semantically, part-scenarios are indeed *parts*; i.e., not only do they specify the interaction between part-objects, but in addition their lifespan (as scenario instances) and scope (in terms of binding and unification) are restricted by the parent scenario. We define syntactic rules and (operational) semantics for LSC-trees and present a composition algorithm that checks the consistency of a given LSC-tree and outputs a semantically equivalent (implicit) annotated *flat-LSC*. The semantics of LSC-trees and the composition algorithm handle the classical partial order semantics of sequence diagrams and the must/may modalities of LSC.

An implementation of LSC play-out using aspects was presented in [16] and has been implemented in the S2A compiler [9]. In way of implementing the ideas of the present paper, we have implemented object composition in S2A by supporting the compilation of LSC-trees. The implementation is compliant with the UML2 standard notion of part-decomposition ([19], pp. 496–499) and the *modal* profile as defined in [10].

Combining object composition with scenario-based specifications has been studied before (see, e.g., [12, 15]). The main contributions of our work are these: we explicitly describe and implement a composition algorithm; we focus on the operational semantics and the execution of the composed scenarios; and, finally, we not only handle the classical partial order semantics of sequence diagrams but consider the part-decomposition extension in the context of the more expressive must/may (hot/cold) modal semantics of LSC. We consider the extended semantics along with the concepts presented in Sec. 4 and Sec. 5, as the main contributions of this paper.

The paper is organized as follows. In Sec. 2 we briefly discuss LSC and object composition. Sec. 3 presents syntax and semantics for the integration of the two, and defines *LSC-trees*. Sec. 4 describes the lifeline composition algorithm and discusses its complexity. Some more advanced issues are discussed in Sec. 5. In Sec. 6, we illustrate our work using a simple example. Sec. 7 discusses related work and Sec. 8 concludes. Additional technical details, proof sketches, an optimized version of the basic composition algorithm, and an extended example appear in [4].

2 Preliminaries

2.1 Live Sequence Charts and Play-out

Live Sequence Charts We use a UML2 compliant variant of live sequence charts (LSC) [7, 10, 11], a visual formalism for scenario-based inter-object specifications which extends the partial order semantics of classical message sequence charts (MSC) [13] with universal and existential modalities. LSC is defined as a proper UML profile that extends UML2 Interactions [19] with a `<<modal>>` stereotype consisting of two attributes: *mode* and *execution mode*. Each element in an LSC, e.g., a message, a constraint, has a *mode* attribute which can be either *hot* (universal) or *cold* (existential), and an *execution mode*, which can be either *monitor* or *execute*. Thus, LSC allows not only to specify traces that "may happen", "must happen", or "should never happen", but also to divide the responsibility for execution between the environment, the participating objects, and the coordination mechanism. Notice that this LSC variant is a proper extension of the original LSC language. For example the notion of prechart is generalized, since cold fragments inside universal interactions serve prechart-like purposes: a cold fragment does not have to be satisfied in all runs but if and when it is satisfied it necessitates the satisfaction of its subsequent hot fragment; and this is true in all runs. LSC notation extends the classical sequence chart notation as follows: hot (resp. cold) elements are colored red (resp. blue), execution (resp. monitoring) elements use solid (resp. dashed) lines.

Play-out An operational semantics for LSC, termed *play-out*, was presented in [11]. Each event in a chart includes a number of locations and *covers* (visually and logically) one or more lifelines. The covered lifelines are those that participate in the execution of the event or need to synchronize on it. A *minimal event* in a chart is an event, which no other event precedes it in the partial order of event induced by the chart. Minimal events are important in our execution mechanism: whenever an event e occurs, a new copy of each chart that features e as a minimal event is instantiated and start being monitored/executed. Each active LSC, instantiated following the occurrence of a *minimal event*, has a *cut*, which is a mapping from each lifeline to one of its locations. Roughly, the execution mechanism reacts to events that are statically referenced in one or more of the LSCs; for each LSC instance the mechanism checks whether the event is *enabled* with regard to the current cut; if it is, it advances the cut accordingly; if it is *violating* and the current cut is cold (a cut is cold if all its elements are cold and is hot otherwise), it discards this LSC instance; if it is violating and the current cut is hot, an exception is thrown; if the event does not appear in the LSC, it is ignored (an LSC does not restrict the occurrence of events not explicitly appearing in it). Conditions (UML2 state-invariants), are evaluated as soon as they are enabled in a cut; if a condition evaluates to true, the cut advances accordingly; if it evaluates to false and the current cut is cold, the LSC instance is discarded; if it evaluates to false and the current cut it hot, an appropriate exception is thrown. If the cut of an LSC instance reaches maximal locations on all lifelines, the instance is discarded. Once all the cuts have been updated, the execution mechanism chooses an event to execute from among the execution-enabled methods that are not violating any chart, if any exist.

Play-out requires careful event unification and dynamic binding mechanism. Roughly, two methods are unifiable if their senders (receivers) are concrete instance-level (or are already bound) and equal, or are symbolic class-level of the same class and at least one is still unbound. When methods with arguments are considered, an additional condition requires that corresponding arguments have equal concrete values, or that at least one of them is free.[1]

The LSC `MVCSetState` (Fig. 1), specifies an interaction between 3 objects: `view, controller` and `model` of types: `IView,IController`, and `IModel` respectively. The three lifelines in the chart are interface-level, i.e, each of them can represent any instance that implements the corresponding interfaces. This LSC specifies part of a variant of the behavior of the classic *Model-View-Controller* design pattern (MVC): *Whenever the view informs the controller that the user has input, and the input is not null, then a series of actions must eventually occur: the controller should eventually set the model's state* [2]; *the model should eventually inform the view that the state has changed, and then the view should eventually update itself according to the new state. Finally, the controller should order the view to start listening for new input.*

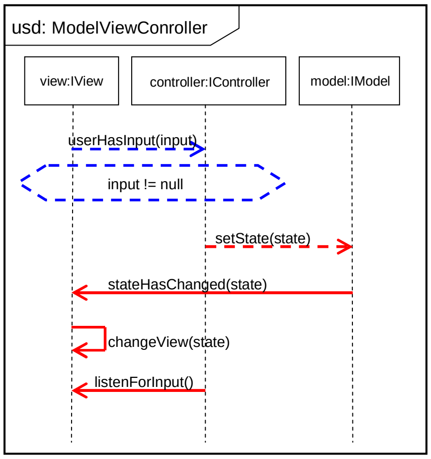


Fig. 1. The LSC `MVCSetState`

Scenario aspects and S2A An implementation of play-out using aspects was suggested in [16], and has been implemented in the S2A compiler [9]. Each LSC is translated into a *scenario aspect*, which simulates a small automaton whose states correspond to the LSC cuts. S2A exploits the inherent similarity between the scenario-based approach and the aspect-oriented approach to software specification: in both cases part of the system's behavior is specified in a way that explicitly crosses the boundaries between objects. The compilation scheme takes

[1] Full definitions of the play-out algorithm for LSCs, including unification, can be found in [11].

[2] This method is marked as 'monitored' so this LSC is not responsible for executing it.

advantage of the similar unification semantics of play-out and AspectJ pointcuts. The work described in the present paper was implemented in S2A, see Sec. 4.

2.2 Object composition

Object composition, that is, a 'part-of' hierarchical relation, is a fundamental concept in object oriented analysis and design [5]. We consider composition to imply strong ownership, that is, part-objects are intrinsically associated with their whole, and do not exist independently (in the UML standard this is termed *composite aggregation* in contrast to other types of aggregation). A part instance is included in at most one composite ('whole') at a time. The composite object (the 'whole') has the responsibility for the existence and storage of its part objects. If a composite is deleted, all of its parts are deleted with it [6, 19].

We assume that a system is equipped with a directed acyclic graph (DAG) of objects (representing the *part-of* binary relation). The DAG has a transitive deletion characteristics; deleting an object results in the deletion of the subgraph below it (i.e, deleting its parts). The graph can be symbolic, with objects being replaced by classes. Notice that since a part object can be included in at most one composite, the object graph is actually a forest (at the class level, we allow classes to be part-of more than one class – the single owner restriction applies only for instances – hence the use of a DAG rather than a tree).

We use the directed acyclic graph of objects (or classes) in order to answer relation queries about objects (at run-time) or classes (statically, during LSC compilation). We say that a class (object) A is a *part-of* a class (object) B iff there is a direct edge from B to A in the DAG. B *has-a* A iff A is *part-of* B. We say that A is *recursively part-of* B if there is a directed path from B to A. Thus, 'recursively part-of' is a strict partial order (transitive, antisymmetric, and irreflexive) between objects (classes).

The package `PhoneBook` in Figure 4 displays a simple part-of graph. For example, in the figure, the class `InputPane` is part-of the class `PhoneBookView`.

3 Object Composition in Scenario-Based Programming

We are now ready to present our integration of object composition and scenario-based programming. We use UML2 terminology.

3.1 The basics

Syntactically, we use *PartDecomposition* as the main mechanism to introduce object composition into a scenario-based specification. In every scenario, each lifeline may be decomposed into a new set of lifelines, which collectively form a new scenario. Thus, a forest-like hierarchical structure is created.

More formally, Let L be an LSC with a set of lifelines $I = \{I_1, \ldots, I_n\}$. Each lifeline I_i has a property I_i*.decomposedAs*, which can either be null (and then we call I a *flat* lifeline) or hold a reference to another LSC L_{I_i}, which specifies the inner behavior of I_i's parts modulo the scenario described in L. We call L a *parent-LSC* and L_{I_i} a *part-LSC*. Lifelines of part-LSCs may be further decomposed and an LSC may have several non-flat lifelines, giving rise to the depth and width of the hierarchical structure. An LSC is *flat* if all its lifelines are flat. The *part-of* relation defined between objects in the model is naturally extended to scenarios. An LSC L_1 is *part-of* an LSC L_2 iff L_2 contains a lifeline I s.t. $I.decomposedAs = L_1$. We call the resulting hierarchical structure an *LSC-tree*. In the following we adopt tree terminology and use *parent*-LSC, LSC-*node*, LSC-*leaf* (necessarily flat), and LSC-*root*.

We require two basic syntactic constraints on the part-of relation between LSCs, as follows (we refine these rules in Sec. 5):

R1 Lifelines of a part-LSC may only represent classes (objects) that are part-of the class (object) represented by the decomposed lifeline.

R2 Given a lifeline I decomposed into part-LSC L, all events covering I must appear in L and induce the same partial order.

A specification that violates the above rules is considered inconsistent.

Fig. 2 shows an example LSC-tree. The LSC-root, `MVCSetState`, has two non flat lifelines. The `view` lifeline is decomposed into the part-LSC `ViewDetailed` and the `model` lifeline is decomposed into the part-LSC `ModelDetailed`. The restrictions described above hold for both part-LSCs. In the following we refer to this LSC-tree by its root's name.

The second restriction above induces a natural correspondence relation between the events that appear in a parent-LSC and cover the decomposed lifeline and (some of) the events that appear in its part-LSC. We call these *corresponding events*. For example, the method `changeView(state)` in the part-LSC `ViewDetailed` *corresponds to* the method with the same signature in the parent-LSC `MVCSetState`. For the time being we require that corresponding events preserve temperature and execution modes. In Sec. 5 we present a setting where this rule is refined, and we discuss additional issues regarding corresponding events.

3.2 Operational semantics

The notions of cuts, enabled/violating events, and minimal events, are carried over from the semantics of flat LSC to the semantics of the LSC-tree. The differences manifest themselves mainly by modifications in the definitions of minimal-events, violating-events, and most importantly lifeline bindings and event unification rules.

We start by adding restrictions to the lifeline binding rules. Consider an LSC L and a lifeline I in L that is decomposed into a part-LSC L_I. Lifelines in L_I can only be bound to objects that are indeed parts-of the object (class) represented by I in L. During execution, for a given instance of L, once I binds to an object O, all the lifelines in L_I that represent its parts may be bound to O's parts only. Bindings may occur in the other direction too: when a lifeline in an instance of L_I binds to a (part) object P, I binds to the owner of P.

Message unification rules are also extended in a natural way. Messages that are unifiable in the flat setting are still unifiable. In addition, instead of requiring identical callers (receivers) or unifiable types, we also allow a setting where one caller (receiver) is (recursively) part-of the other (specifically, corresponding

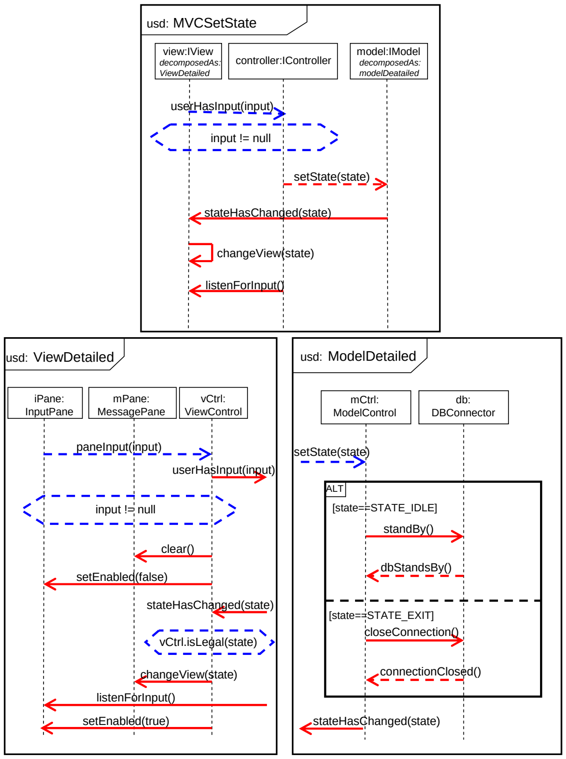


Fig. 2. The MVCSetState LSC-tree; the tree is composed of a root (MVCSetState) and two part-LSCs, (ViewDetailed and ModelDetailed).

events are unifiable). Note that this still allows identical lifeline bindings and message unification across different LSCs in the specification, including between different LSC-trees (see the examples in [4]).

The set of minimal events must be considered with regard to the partial order induced by the LSC-tree as a whole. Thus, a minimal event of an LSC-node might not be minimal for the tree. When a minimal event with regard to the tree occurs, an active instance of the whole tree is created (see the examples in [4]). Allowing part-LSCs to introduce new minimal events has its advantages and disadvantages. In Sec. 5.5 we discuss a setting where part-LSCs are not allowed to introduce new minimal events.

Recall that a cut of an LSC induces a set of violating events and a set of enabled events. When dealing with LSC-trees, each LSC-node in an instance of an LSC-tree has its own (local) cut, and hence its own sets of enabled and violating events. The rule for enabled events does not change: when an enabled event in an LSC-node occurs, the cut of this node advances accordingly. As usual, due to event unification, a single event might cause several cuts in several LSCs in the same tree to advance simultaneously.

When an event that violates one of the LSCs in an LSC-tree occurs, we interpret it as violating the tree. The violation is hot if one of the LSCs in the tree (not necessarily the one where the local violation 'occurred') was in a hot cut when the violation occurred, and it is cold otherwise. When a cold violation occurs in an instance of an LSC-tree, the entire instance is discarded; when a hot violation occurs an appropriate exception is thrown, as in the flat LSC setting.

4 Lifeline Composition Algorithm

The composition algorithm receives as input a single LSC-tree and if the LSC-tree is consistent it outputs an annotated (implicit) flat-LSC that captures exactly the behavior specified by the tree. If the LSC-tree is inconsistent, i.e., it violates rules R1, R2 defined in Section 3.1, an appropriate output message is given. For example, Fig. 3 displays the (implicit) flat-LSC created as a result of running the algorithm on the LSC-tree `MVCSetState` (of Fig. 2).

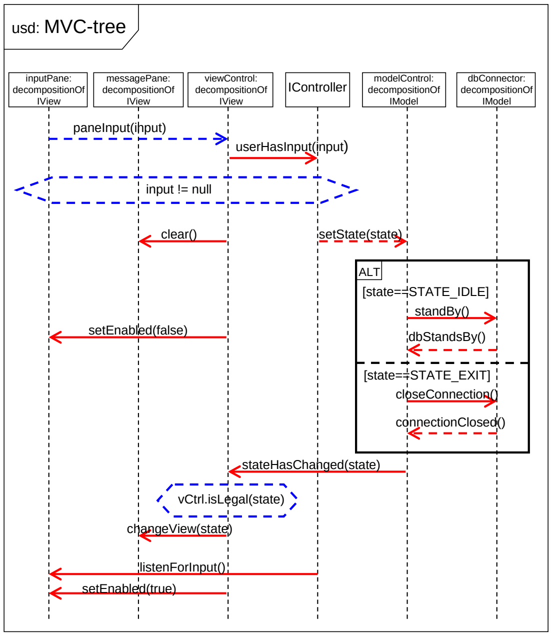


Fig. 3. The (implicit) composed flat-LSC

The basic procedure used by the algorithm merges two LSCs, a part-LSC and its parent-LSC, into a single annotated flat-LSC. We use a DAG to represent the partial order of events of each LSC, and merge the two DAGs using a merging algorithm inspired by the classic way to merge sorted lists [14].

By applying the merging procedure recursively from leaves to root, the algorithm collapses the tree into a single flat-LSC. The resulting LSC's set of lifelines is the union of all the LSC-node's sets of lifelines, where we (recursively) replace each decomposed lifeline with the lifelines of its corresponding part-LSC. The pseudo-code for the (essence of the) merging procedure appears in Proc. 1. A proof of correctness and a detailed explanation of the algorithm are given in [4].

procedure 1 *composeLSCs*(LSC L_{parent},LSC L_{part})

```
let $I$ be the lifeline in $L_{parent}$ that is decomposed into $L_{part}$
create new LSC $L_{flat}$
set $L_{flat}$ lifelines to be $L_{parent}$'s lifelines but replace $I$ with $L_{part}$'s lifelines
let $E_{parent}$ be $L_{parent}$'s event graph
let $E_{part}$ be $L_{part}$'s event graph
while $E_{parent} \neq \emptyset$ do
    let $e$ be a minimal event in $E_{parent}$
    $E_{parent} \leftarrow E_{parent} \setminus \{e\}$
    if $e$ does not cover $I$ then
        append $e$ to $L_{flat}$
    else {//$e$ covers $I$}
        $addPartLSCEvents(E_{part}, e, L_{flat})$
    end if
end while
$addPartLSCEvents(E_{part}, null, L_{flat})$
return $L_{flat}$
```

procedure 2 *addPartLSCEvents*(Event graph E_{part},Event e_{parent},LSC L_{flat})

```
while $E_{part} \neq \emptyset$ do
    let $e_{part}$ be a minimal event in $E_{part}$
    if $e_{part}$ corresponds to $e_{parent}$ then
        $E_{part} \leftarrow E_{part} \setminus \{e_{part}\}$
        let $e_u$ be the static unification of $e_{part}$ and $e_{parent}$
        append $e_u$ to $L_{flat}$
        return
    else if $e_{part}$ corresponds to another event in $L_{parent}$ then
        if there are more minimal events in $E_{part}$ then
            choose another minimal event and continue the loop with it.
        else {//no more minimal events}
            throw 'Error: Violation of the partial order'
        end if
    else {// $e_{part}$ does not correspond to any event in the parent-LSC}
        append $e_{part}$ to $L_{flat}$
    end if
end while
if $e_c == null$ then
    return $\emptyset$
else
    throw 'Error: No corresponding event for $e_{parent}$ in the part-LSC'
end if
```

Complexity The *composeLSCs* procedure runs in time linear in the total number of events in the two input LSCs, since it traverses each event in the two DAGs only once. We assume that operations such as static event unification and adding an event to a list take constant time. To allow extraction of minimal events in constant time we keep a dynamic set of sources for each DAG.

Let T be an LSC-tree with n LSC-nodes. Denote the set of events of an LSC L by E_L. Let $k = \max\{|E_L| : L \in T\}$. The complexity of the composition algorithm is $O(n^2k)$. The report in [4] contains a detailed analysis of the algorithm's complexity and suggests a simple optimization (already used in our implementation), which results in a running time of $O(nk)$.

4.1 Implementation

We have implemented the algorithm in the S2A compiler [9] by adding the ability to compile LSC-trees. S2A uses the optimized version of the composition algorithm presented above to reduce each LSC-tree in the specification to a single annotated flat-LSC. It then translates this implicit flat-LSC into a *scenario aspect* [16]. Some modifications of the original scenario aspect code are required in order to support the extended lifeline binding and unification rules.

5 Advanced Issues

We now discuss several advanced issues that arise in our work.

5.1 Lifeline bindings and scope

Integrating object composition into scenario-based specifications introduces a new notion of scope, which materializes in lifelines binding and event unification rules. Consider a parent-LSC L_1 with lifeline I_1 of type C_1, decomposed into a part-LSC L_2. Let I_2 be a symbolic lifeline in L_2 representing a class C_2. I_2 cannot be bound to any object of type C_2. Rather, it may be bound only to an object of type C_2 that is also *part-of* an object of type C_1. The part-LSC scenario is thus considered only within the scope of its parent.

5.2 The partial order

In general, the events covering a decomposed lifeline must all appear in the part-LSC and must induce the same partial order. It is possible, however, that the total order of events along a decomposed lifeline is relaxed in the corresponding part-LSC. For example, in the LSC `ViewDetailed`, the methods `changeView(state)` and `listenforInput()` are unordered, while in the parent-LSC `MVC` the corresponding methods are ordered (see Fig. 2).

We consider two possible design intentions. The designer may have intended to indeed specify a total order between the events but could not express this in the part-LSC due to the disjoint sets of covered lifelines. Alternatively, the designer may have intended an explicit partial order but could not express it in the parent-LSC because the events share a covered lifeline.

The S2A compiler produces a warning when this issue occurs in an LSC-tree. In our current work and its implementation, we support the first design intention by default. That is, in addition to compiling the composed annotated implicit flat-LSC, each LSC-node in the tree is compiled separately. This allows monitoring the order specified in any parent-LSC, and enforcing the total order of execution (or reacting otherwise in case this order is violated).

The second design intention can be supported by allowing the user to use the *co-region* operator, which relaxes the total order along a single lifeline (see [3, 7, 19]). Our current implementation does not support this.

5.3 Identifying corresponding events

Our composition algorithm depends on the ability to decide whether two events on different levels of the tree correspond. This assumption, however, is not obvious. For example, a given method in a parent-LSC may have several legal corresponding methods in the part-LSC. The use of conditions and symbolic parameters may create similar complex situations. One may consider this as a form of under-specification.

In practice, this problem can be partly addressed by requiring the designer to statically mark corresponding events, or, as in our current implementation, when more than one possibility exists, the composition algorithm could emit an appropriate warning message and make a non-deterministic choice.

5.4 Hot/cold modalities in an LSC-tree

Recall that LSCs are modal scenarios. Each event has a temperature attribute: cold events *may* eventually occur, while hot events *must* eventually occur (see [7, 10]). These modalities should also be considered in the semantics of LSC-trees. The simplest approach is to require that corresponding events be given identical temperatures. This constraint, however, is neither sufficient nor necessary. To see that it is not sufficient consider a cold event added in the part-LSC in between two hot events that have corresponding parent-LSC events (e.g., Fig. 2, the cold condition `vCtrl.isLegal(state)` in the `ViewDetailed` part-LSC) . A cold violation that occurs at this part-LSC when this cold event is enabled will always result in a hot violation for the LSC-tree. Thus, in terms of trace languages (see [10]), a run accepted by the composed flat-LSC may not be accepted by the parent-LSC.

Therefore, to carry over the modal characteristics of the language from the single LSC to the LSC-tree, we define the following rules, which ensure the consistency between a part-LSC and its parent in terms of accepted runs:

R3 Hot events in a parent-LSC must remain hot in the part-LSC. (Cold events in a parent-LSC can be either cold or hot in the part-LSC)

R4 For any event that appears in a part-LSC and does not appear in the parent-LSC, if there is a next (minimal) event after it that has a corresponding hot event in the parent-LSC, then the new event must be hot.

Roughly, these additional rules ensure that a finite trace inducing a hot cut in the parent-LSC, will also induce a hot cut in the LSC-tree (that is, in the composed flat-LSC). In [4] we show that they are indeed necessary for the soundness of our work. Note that a check for these restrictions can be easily integrated into our composition algorithm. Indeed, our implementation checks these conditions and outputs an appropriate warning if necessary.

5.5 Minimal events in an LSC-tree

Recall that minimal events have a special role in LSC semantics: whenever a minimal event of a chart occurs, an instance of the chart is created and becomes active. Thus, every occurrence of a minimal event must be followed eventually by a successful completion of the chart. Accordingly, whenever a minimal event in the partial order induced by an LSC-tree (essentially, its corresponding composed flat-LSC) occurs, an instance of the LSC-tree is created and becomes active.

Note, however, that if part-LSCs introduce new minimal events with respect to their parent, a run accepted by the composed flat-LSC may not be accepted by the parent-LSC. Depending on the context and usage of LSC-trees (e.g., formal verification, testing, execution), this may or may not be considered problematic. Thus, to ensure trace containment between an LSC-tree and its root-LSC, we suggest restricting part-LSCs from introducing new minimal events:

R5 Every minimal event in a part-LSC must have a corresponding event in the parent-LSC.

In [4] we give the formal definition of R5 and show that it is necessary for the trace containment property. Still, we believe that deciding whether or not to apply R5 should depend on the specific application.

5.6 Existential LSC-trees

The language of LSC defines two types of charts, universal and existential [7, 10]. We have concentrated on universal LSCs here, since they are the ones involved in play-out. The lifeline decomposition extension, however, may be applied to existential LSCs too (which can be used, e.g., for testing and monitoring), creating existential LSC-trees that specify, as usual, scenarios that must be satisfied by at least one possible run of the system model. Existential LSCs do not use the hot/cold modalities and their minimal events have no special semantic significance. Hence, the basic rules defined in Sec. 3.1 (R1 and R2) suffice to ensure the consistency of an existential LSC-tree. Specifically, the composition algorithm checks the consistency of existential LSCs too and outputs a corresponding existential flat-LSC. Thus, our implementation works for existential LSCs and indeed checks the consistency of 'classical' UML2 sequence diagrams that use part-decomposition with an existential interpretation and no modalities.

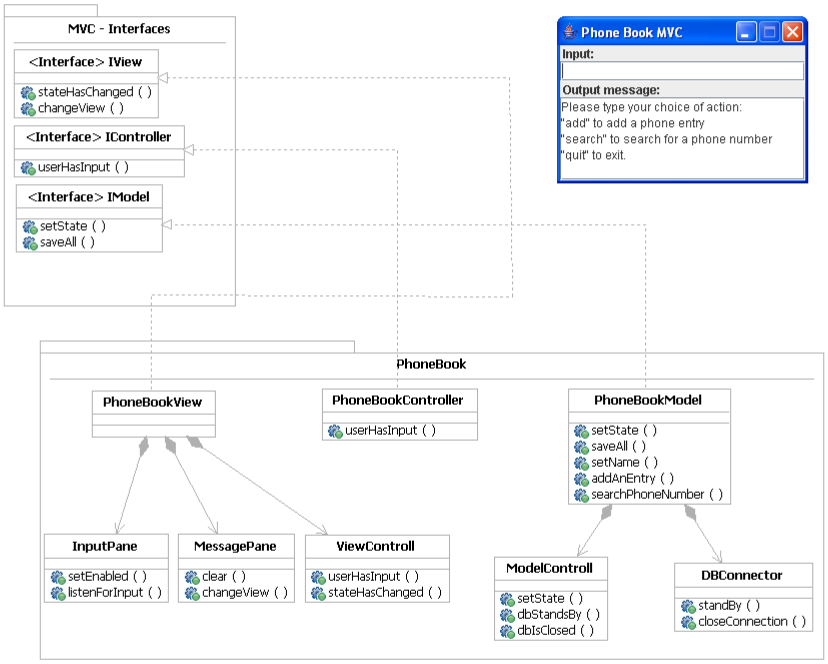
6 Example: Phone Book Application

We demonstrate the key features of our work using a small example of a Phone Book application[3]. It has a simple user interface, allowing a user to add name/number pairs to the phone book, search a number by name, etc. A detailed description appears in [4]. The example UML model and code are available for download from the S2A website [1].

Fig. 4 displays the class diagram of the application and the application's GUI. The application uses the MVC design pattern. The classes in the diagram are implemented in Java. The code for the interactions between these classes, however, was generated from a UML2 compliant LSC specification using the S2A compiler.

The application uses three interfaces, `IView`, `IController`, and `IModel`, implemented by the classes `PhoneBookView`, `PhoneBookController`, and `PhoneBookModel` respectively. The generic interaction between the three interfaces is specified in the `MVCSetState` LSC (Fig. 1) mentioned earlier. The behavior described in `MVCSetState` is relevant for many applications that are based on this design pattern; i.e, it is reusable. In order to use it in the phone book application we

[3] The example was partly inspired by an IBM tutorial for Rational Software Architect written by Tinny Ng, available from http://www.ibm. com/developerworks/.



Please type your choice of action:
"add" to add a phone entry
"search" to search for a phone number
"quit" to exit.

Fig. 4. The Phone Book application's class diagram and GUI

apply the part-of decomposition mechanism to two lifelines: the IView lifeline is decomposed into the ViewDetailed part-LSC, and the IModel lifeline is decomposed into the ModelDetailed part-LSC (for additional LSCs used in the application, see [4]).

The class PhoneBookView is responsible for the phone book's GUI (Fig. 4), consisting of two visible parts, the MessagePane (which is where the messages for the user are displayed), and the InputPane (which is where the user enters textual input). In addition, PhoneBookView has another (logical) part, the ViewController, which coordinates between the GUI parts and is used as a gateway object. The inner behavior of the PhoneBookView with respect to MVCSetState is specified in the ViewDetailed part-LSC (Fig. 2).

The class PhoneBookModel is responsible for the phone book's storage and holds the current state of the application. Its inner behavior (with respect to the MVCSetState scenario) is specified in the ModelDetiled LSC (Fig. 2), which reveals the events that take place 'inside' the PhoneBookModel after its state is changed. The specific behavior depends on the new state.

Note that while the MVCSetState LSC describes the behavior of general interfaces that can be implemented in different applications, the decomposed part-LSCs are application specific. This shows the power of symbolic instances [17] and their implementation in S2A in creating *reusable* behavioral specifications.

Also note the flexibility and modularity that object-composition adds to the scenario-based specification. For example, we have created another implementation of the IView interface, replacing the class PhoneBookView with a new class

PhoneBookExtendedView of enhanced GUI. The inner behavior of the latter (in the context of this scenario!) is specified in a new LSC ExtendedViewDetailed which by itself is not flat; i.e., it contains a lifeline that is decomposed into another LSC, involving the new GUI elements. We were able to change the application's behavior by simply replacing a subtree in the original LSC-tree.

The complete example, including UML model and code is available from the S2A website [1]. Additional details including a snippet from the generated code appear in [4].

7 Related Work

The notion of lifeline decomposition appears already in [13] and in the UML2 standard [19], which includes syntactic restrictions for part-decomposition. These were used as a basis for our basic syntactic rules.

STAIRS [12] is an approach for the compositional development of UML interactions. Among the refinement relations formally defined in STAIRS is *detailing*, which is based on lifeline decomposition.

Krüger [15] defines various refinement relations between MSCs, one of which is *structural refinement*, which relates object refinement in the system model with MSC refinement. Krüger suggests syntactic rules for the substitution of one MSC with another, based on the object composition in the system model.

The Rhapsody tool [2] allows the user to define a 'decomposed-as' reference from a lifeline to a sequence diagram. However, no syntactic rules are checked by the tool.

Our work uses the part-decomposition mechanism presented in the standard and defines similar syntactic restrictions for part-LSCs. We explicitly describe and implement a composition algorithm that reduces an LSC-tree into a flat-LSC and checks the consistency of the LSC-tree with respect to the basic rules adopted from the standard (R1, R2) and the more advanced rules (R3, R4); we not only handle a trace-based semantics but focus on the operational semantics and the execution of the composed scenarios. Also, we handle both the classical partial order semantics of sequence diagrams and the part-decomposition extension in the context of the more expressive must/may (hot/cold) modal semantics of LSC. Most of our work is applicable to the aforementioned work too.

8 Discussion and Future Work

We have extended the LSC language to allow the specification and interpretation of scenario hierarchies — trees of scenarios — based on an object composition hierarchy in the underlying model. The present paper grew out of previous work done in our group [3], in which LSC was extended with support for object composition by defining LSC-trees. Here we have decided to omit some of the more complicated issues dealt with in [3].

Composition and inheritance are two complementary concepts in OOD. Partial support for inheritance was introduced to LSC using symbolic instances in [17], and was implemented in the Play-Engine tool [11]. In [9, 16] this was explicitly extended to support class inheritance and interface implementation in

Java. The present paper integrates object composition into the scenario-based context, and thus may be viewed as complementing this previous work.

While we focus on LSCs and their direct execution, the presented ideas are applicable to UML2 sequence diagrams in general, and to their use throughout the development cycle. Planned future work includes the development of design methodologies that will take advantage of our work, the implementation of additional case studies using the S2A compiler, and related compiler optimizations.

References

1. S2A Website. http://www.wisdom.weizmann.ac.il/~maozs/s2a/.
2. Telelogic Rhapsody. http://www.telelogic.com/.
3. Y. Atir. Object Refinement and Composition in Scenario-Based Programming: An LSC Extension. Master's thesis, The Weizmann Institute of Science, 2005.
4. Y. Atir, D. Harel, A. Kleinbort, and S. Maoz. Object Composition in Scenario-Based Programming. Technical report, The Weizmann Institute of Science, 2007.
5. G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.
6. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, second edition, 2005.
7. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
8. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.
9. D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *Proc. Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 121–124. Springer, 2007.
10. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling*, 2007. To appear.
11. D. Harel and R. Marelly. *Come , Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003.
12. Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4(4):355–357, 2005.
13. ITU. Recommendation Z.120: Message Sequence Charts. Technical report, 1996.
14. D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1998.
15. I. Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2000.
16. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proc. 14th Int. ACM/SIGSOFT Symp. Foundations of Software Engineering (FSE-14)*, Portland, Oregon, November 2006.
17. R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proc. 17th ACM Conf. on Object-Oriented Prog., Systems, Lang. and App. (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.
18. S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.*, 29(2):99–115, 2003.
19. UML. Unified Modeling Language Superstructure Specification, v2.0. OMG spec., OMG, August 2005. Available from http://www.omg.org.
20. J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: An air traffic control case study. *Software and Systems Modeling*, 4(1):71–93, 2005.