Semantically Configurable Analysis of Scenario-Based Specifications

Barak Cohen and Shahar Maoz

School of Computer Science, Tel Aviv University, Israel

Abstract. Scenarios, represented using variants of sequence diagrams, are popular means to specify systems requirements. Live sequence charts (LSC), is a formal and expressive scenario-based specification language, which has been extensively studied over the last decade. Careful reading of the LSC literature, however, reveals many variations and ambiguities in the semantics of LSC, as it is used by different authors in different contexts. Moreover, different works define their semantics of LSC using different means. This variability, in both language features and means of semantics definition, creates a challenge for researchers and tool developers.

In this paper we address this challenge by investigating semantically configurable analysis. We define and formalize the variability in the semantics of LSC using a feature model and develop an analysis technique that can be instantiated to comply with each of its legal configurations. Thus, the analysis is semantically configured and its results change according to the semantics induced by the selected feature configuration. The work is implemented and demonstrated using examples. It advances the state-of-the-art in the area of scenario-based specifications and provides an example for a formal and automated approach to handling semantic variability in modeling languages.

"...the world don't move to the beat of just one drum..."
Diff'rent Strokes (1978)

1 Introduction

Scenarios, represented using variants of sequence diagrams, are popular means to specify systems requirements. A scenario tells a 'short story' of interaction between system and environment entities. Live sequence charts (LSC), originally presented by Damm and Harel [4], is a formal and expressive scenario-based specification language. LSC has been extensively studied over the last decade in the context of execution and synthesis (e.g., [9,16,23,25,31,33,40]), in the context of consistency checking and formal verification (e.g., [6,11,21,26]), specification mining and testing (e.g., [27]), expressive power and standardization (e.g., [5,12,13,22,30,41]). Moreover, several tools which support various analyses that involve the LSC language have been developed by different research groups, including, e.g, PlayGo [14], the Modal Transition System Analyzer [7], and ScenarioTools [10,39].

Careful reading of the LSC literature, however, reveals many variations and ambiguities in the semantics of LSC, as it is used by different authors in different contexts, for different purposes and in different tools. Moreover, different works define their semantics of LSC using different means, e.g., by transformation to temporal logic formulas, by describing an execution mechanism (play-out), by translation into various types of automata, etc. This variability, in both language features and means of semantics definition, creates a challenge for researchers and tool developers.

In this paper we address this challenge by investigating *semantically configurable analysis of LSC*. First, we define and formalize the variability in the semantics of LSC, as it is found in the literature, using a feature model: each configuration that the feature model permits, induces a different semantics mapping (over the same domain). Second, we develop a parametrized analysis technique that can be instantiated to comply with every legal configuration of the feature model. Thus, the resulting analysis, e.g., verification or synthesis, is semantically configured and its results change according to the semantics induced by the selected feature configuration.

There are several advantages to using a feature model to describe a language's semantic variability. First, the feature model provides a means to formally structure the various semantic choices; this supports human comprehension of the semantics, allows comparison of different variants, and, significantly, enables the parsing required in order to support an automatically configurable analysis. Second, the use of a feature model provides a formal means to define logical dependencies between the semantic choices, e.g., mutual exclusion, implication etc. This is indeed necessary, because not all theoretically possible combinations induce well-defined and useful semantics that are found in the literature.

To present the semantics of LSC in our work, we chose a single, uniform semantic domain — traces of events — and a uniform formalism — alternating one pair Streett automata (see Appendix A and, e.g., [8]), which is expressive enough to faithfully support the representation of all variants we have found in the literature. This uniform representation enables human comprehension and comparison between variants, and serves as a basis for building semantically configurable automated analysis tools.

Our feature model for the semantics of LSC consists of 19 features. One feature, for example, relates to whether the LSC should be interpreted universally or existentially. Another feature relates to the question of whether the chart's semantics is tolerant or strict with regard to partial-order violations by events that appear in it. One feature relates to the semantics of pre-charts in existential charts, a set of features differentiates between invariant, and iterative modes of interpretation, a set of features relates to the use of environment assumptions, and another set of features differentiates between true and interleaving modes of concurrency. Each feature is formally defined as part of the LSC semantics

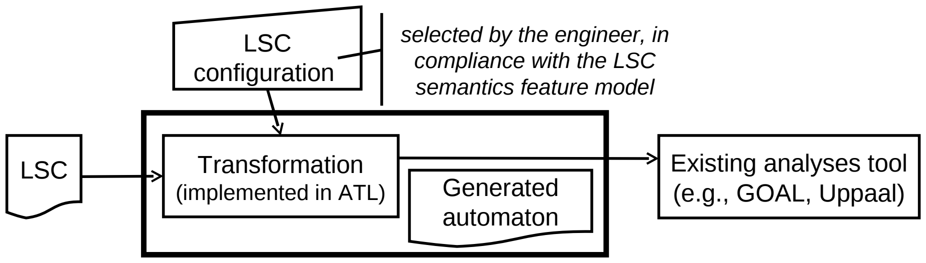

LSC configuration
selected by the engineer, in compliance with the LSC semantics feature model
LSC
Transformation (implemented in ATL)
Generated automaton
Existing analyses tool (e.g., GOAL, Uppaal)

Fig. 1. The architecture of our solution to semantically configurable analysis of LSC

definition. The feature model organizes the different features so that each of its configurations induces a specific overall semantics.

The semantic mapping itself is realized using a model-to-model (M2M) transformation. The input for the transformation consists of (1) an LSC and (2) a valid configuration of the feature model. The output is an automaton, which can be used as input for downstream analysis tools. An overview of the architecture of our solution is shown in Fig. 1.

Our work is fully automated and implemented in a prototype Eclipse plug-in, where one can edit an LSC, select a semantic configuration, and generate an automaton corresponding to the LSC semantics according to the chosen configuration. For LSC editing we use components from PlayGo [14]. For feature model definitions and implementation of feature selection we use components from FeatureIDE [19]. The M2M transformation is implemented in ATL [18].

The remainder of the paper is organized as follows. Sect. 2 discusses related work. Sect. 3 provides an overview and an example. Sect. 4 describes the LSC language and the feature model of its semantics. Sect. 5 presents our technique for semantically configurable analysis. Sect. 6 presents the implementation and a discussion. Sect. 7 concludes.

2 Related Work

The question of how to deal with semantic variability in a modeling language has been investigated before. A series of works by Atlee et al., e.g., [29,37,38], used *template semantics* to configure the semantics of state machines, and demonstrated configured translations of state machines into SMV and into Java. Different from these works, we use a feature model to model semantic variability. Moreover, these works relate to state machine models while our present work focuses on scenario-based models.

Cengarle et al. [2] have presented a taxonomy of variability mechanisms in language definitions syntax and semantics, and demonstrated the use of feature diagrams to model possible variants. The present work builds on these previous ideas while focusing on semantic variability, specifically, semantic mapping variability (rather than syntactic variability) and on its application to semantically configurable analysis, specifically demonstrated and implemented in the context of live sequence charts.

A recent survey [36] has explored the many meanings of UML 2 sequence diagrams. Indeed, we share similar concerns about the challenges set by the existence of many different semantics for sequence diagrams. The survey, however, does not formalize the various semantics mathematically and under a single, uniform formalism as a semantic domain. Thus, unlike our work, it cannot provide a basis for a semantically configurable automated analysis. Moreover, the survey does not focus on LSC and ignores many LSC-related works (e.g., [25,41]), whose semantics we do cover in this paper.

Many previous works provide various analyses for LSC, e.g., formal verification, specification mining, and synthesis (e.g., [9,25,27,34,41]). To the best of our knowledge, none of the works in the LSC literature supports variability-based semantically configurable analysis.

Most recently, the second listed author et al. [32] presented semantically configurable consistency checking of class and object diagrams. The work motivates the use of feature models to support semantically configurable analysis. It uses a feature model to specify variations in the semantics of CDs and ODs, and a parameterized translation of CDs and ODs to Alloy, which is expressive enough to support all the considered variants. This work has inspired us to apply a similar solution to address the challenge of variability in the semantics of LSC.

3 Example and Overview

We use a simple example as an overview of our work. The description is partial and semi-formal. We refer back to this example later in the paper.

We consider a single small LSC, related to the vending machine specification presented in [33,34]. The LSC `OnHeatRequest` (Fig. 2, left) consists of one environment lifeline (`heater`) and two system lifelines, representing the system's `panel` and `thermometer`. The minimal event of the LSC is a cold `heat` message that is sent from the panel to the heater. It is followed by two hot messages, with no particular order between them: (1) the panel's own `lockPanel` message, and (2) the heater's `reachMax` message to the thermometer.

We define the semantics of an LSC by translation to an automaton. The language accepted by the automaton consists of the runs that satisfy the LSC.

The construction of the automaton consists of a common part and a variable part. The common part (marked in black in Fig. 2, right) includes the states (one for each LSC cut) and the transitions induced by the unwinding of the LSC's partial order. The variable part, marked in several colors according to the corresponding semantic features, consists of (1) additional transitions, (2) a reject state, (3) a quantification on the initial state, and (4) an acceptance condition.

For example, for a universal semantics, the red transitions and reject state are added on top of the common construction. For the choice between a strict and a tolerant interpretation, the purple or the orange transitions are added.

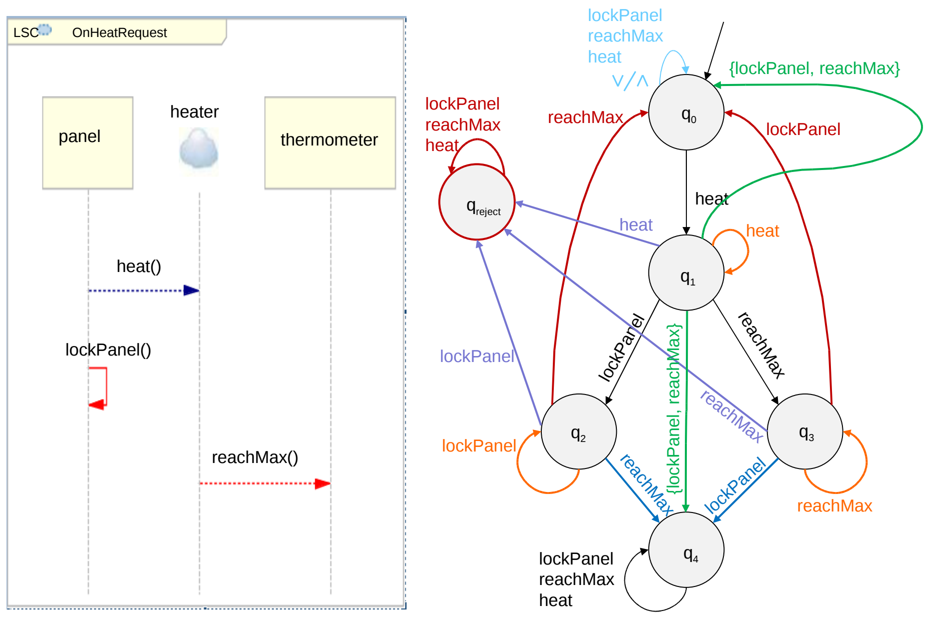


Fig. 2. An example LSC and a sketch of a corresponding automaton. The automaton sketch is color-coded based on the LSC semantics feature model: the common parts are black while the parts corresponding to selected semantic features are marked in different colors, one color per feature.

To support true concurrency rather than interleaving semantics, the green transitions are added. The choice of whether to consider environment assumptions changes the acceptance condition (not shown in Fig. 2).

Characterizing and formalizing the required variability, and showing how it is implemented in a single, configurable analysis solution, are the challenges we address in this paper.

4 LSC Semantics Variability

We start off with an overview of LSC's common syntax and semantics. We then describe the feature model that organizes LSC's semantic variability.

4.1 Live sequence charts common syntax and semantics

A live sequence chart consists of a set of lifelines and messages, depicted in the concrete syntax using vertical lines and arrows between them. Message send and receive events are placed in the intersection of messages and lifelines. On each lifeline, events are fully ordered from top to bottom. Events appearing on different lifelines are not ordered, except that a receive event cannot happen

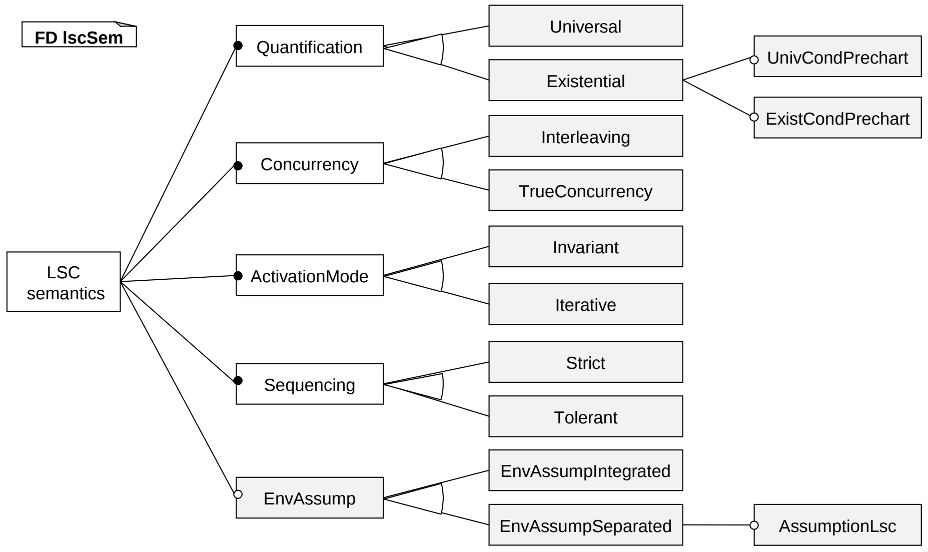


Fig. 3. The LSC semantics feature model, presented using a feature diagram

before its corresponding send event. Thus, the LSC syntax induces a partial order over events. This partial order is common to all LSC variants found in the literature (and in fact also to all message sequence charts and UML sequence diagrams variants).

Another common syntactic feature of all LSC variants is message temperature, which can be either hot or cold. In the concrete syntax, the temperature is reflected by the message's color, red or blue.

These syntax and semantics are common to all LSC variants found in the literature. Next we discuss the semantic variants found in the literature and the feature model we use to formalize and structure them.

4.2 The LSC semantic variability feature model

A feature model describes a structured set of features and their logical dependencies [1,3]. Feature models are commonly used in the area of software product lines. They may be visually represented using feature diagrams, which are basically and-or trees, extended with textual cross-tree logical constraints. Here we use a feature model to formalize variability in the semantics of LSC. The model includes several cross-tree logical constrains. In the feature diagrams we use the standard notation: for mandatory features, a line ending with a filled circle; for alternative features of which exactly one must be selected (xor), an empty slice covering the lines leading to the different alternatives.

Our feature model consists of 19 features, as shown in the feature diagram in Fig. 3. Roughly, a valid feature configuration of this model specifies whether the semantics is universal or existential (with the kind of pre-chart specified), invariant or iterative, of true concurrency or interleaving, using strict or tolerant sequencing, and whether the semantics includes environment assumptions. Some of these features have sub-features. Each of the features and sub-features represents a semantic choice used in one or more works from the LSC literature, as we detail next.

The `Universal` vs. `Existential` semantic choice was first presented in [4] and appeared in almost all works (although many works support only the universal variant). An existential LSC specifies an interaction example and requires that at least one system run exhibits the events appearing in it (in compliance with the partial order specified by the chart). A universal LSC specifies a rule that all system runs are expected to satisfy. A run satisfies a universal LSC iff each time the LSC is activated its hot enabled messages eventually occur, or an enabled cold message is violated.

A pre-chart appeared already in [4]. Many works use an LSC variant with pre-chart, but not all (e.g., [9,33]). When a pre-chart is used in an existential LSC, two variants are found: one variant, introduced in [4] and called in our feature model existential conditional pre-chart (`ExistCondPrechart`), states that there should be at least one system run in which if the pre-chart is traversed successfully, then the main chart is fulfilled as well. Another variant with a stronger interpretation requires that whenever the pre-chart is fulfilled there is at least one execution from that point on that satisfies the main chart. This semantic variant is defined in [41] and called in our feature model universal conditional pre-chart (`UnivCondPrechart`)

The model of concurrency assumed by an LSC may vary. Almost all works use an `Interleaving` interpretation, where no two events happen at the same point in time. Some works, e.g., [4,21], however, do allow true concurrency (`TrueConcurrency`).

The kind of sequencing is another distinction found in the literature. Two sequencing kinds are considered: `Strict` and `Tolerant`. According to the strict interpretation, events that appear in the chart but are not currently enabled cause a violation. According to the tolerant interpretation, these events do not cause a violation. In all variants, the chart's sequencing ignores events not appearing in the chart. Almost all works use strict sequencing. Tolerant sequencing is formalized and investigated in [15].

The mode of activation is another variation found in the literature. We consider two activation modes from the literature: `Invariant` and `Iterative`. In invariant activation mode, every occurrence of a minimal event activates the chart. In iterative mode, a chart is not activated if it is currently active (so at most one instance of the chart may be active at each point in time). The invariant mode is used in many works, e.g., [4,6,12,21,41]. The iterative mode is used in other works, e.g., [10,21,25].

Finally, the use of liveness environment assumptions is another variation point in the LSC literature. Some works make a distinction between environment and system lifelines, which is reflected in the LSC semantics; when the environment violates its assumptions, the system is no longer required to fulfill its guarantees. An integrated variant, which allows to specify environment assumptions and system guarantees in a single LSC, is presented in [33] and used in [14]. A variant where assumptions must appear in separate LSCs is presented in [9,21] and used in [10]. However, while most works do make a syntactic distinction between environment and system lifelines, many of them, e.g., [12,16,41], do not support liveness environment assumptions.

Cross-tree constraints. To complete the feature model, we add to the feature diagram cross-tree logical constraints that define dependencies between the different features, for us, the semantic choices, e.g., mutual exclusion, implication etc. This is indeed necessary, because, as we have found, not all theoretically possible combinations (feature configurations) induce well-defined and useful semantics (which appeared in the literature). Specifically, we add the following:

$$\texttt{EnvAssump}\ \textbf{implies}\ \texttt{Universal} \tag{1}$$

$$\textbf{not}\ (\texttt{UnivCondPrechart}\ \textbf{and}\ \texttt{ExistCondPrechart}) \tag{2}$$

We add constraint 1 because the provisional behavior specified by an existential semantics is too weak to be useful as an assumption on the environment's behavior (*"The quantification is always universal, because assumptions express universal constraints on the behavior of the environment"*, [20, p. 196]). Indeed, in the literature all works that support environment assumptions (e.g., [9,33]) support only universal LSCs. We add constraint 2 because a pre-chart in an existential LSC may have either a universal conditional semantics or an existential conditional semantics, but not both.

Overall, our feature model contains 19 features, 5 of which are core features, i.e., features that are included in all configurations. The model has 56 valid configurations. The complete feature model we have defined is available in [28], in formats compliant with S.P.L.O.T [35] and with FeatureIDE [19], to allow others to inspect it and use it.

5 Semantically Configurable Analysis

We start with a short overview of the target formalism we use as the semantic domain for LSC. We then describe the model-to-model transformation we have defined to support semantically configurable analysis.

5.1 Alternating one pair Streett automata

The key to the semantically configurable analysis is a transformation to a single, uniform formalism, in our case, an alternating one pair Streett automaton [8].

Roughly, an alternating automaton's transition function maps a state and an alphabet symbol to a positive Boolean expression over states. It thus allows expressing both non-determinism (disjunction) and concurrency (conjunction). Streett acceptance condition consists of a set of pairs of bad and good sets of states; a run is accepted iff for each pair, if it visits the bad set infinitely often it also visits the good set infinitely often. A formal definition appears in Appendix A.

For our purposes of representing the semantics of all LSC variants found in the literature, a simpler automaton is sufficient. Specifically, we need only one pair $(F, E) \subseteq Q^2$ of bad and good sets of states for the Streett acceptance condition, and we can limit quantification to the automaton's initial state (and in the case of pre-chart to at most one more state).

In addition to an LSC, the input for the transformation includes one valid configuration of the LSC semantics feature model described in Sect. 4. We now describe the transformation as it is implemented in ATL [18]. We use the LSC presented earlier in Sect. 3 as a running example.

5.2 Overview of the transformation

The ATL transformation uses three meta models: (1) for LSC (input), (2) for a semantics configuration (input), and (3) for an alternating one pair Streett automaton (output). As the transformation is quite complex we use many ATL helpers, to define required data structures and functions.

Most importantly, the transformation uses two kinds of rules: **common rules**, which are invoked based on the LSC input model only and apply to all variants (e.g., construct the unwinding structure based on the partial order), and **feature specific rules**, which are invoked based on the input feature model configuration (rule per feature, see details below).

The notions of unwinding structure, cut and the events it induce, etc. are common to all LSC variants, see, e.g., [12]. For lack of space we do not repeat their definition here and do not show the common transformation rules. The complete transformation is available from [28]. Here we focus only on the feature specific rules.

5.3 Handling semantic variability: feature specific rules

Handling variability is technically realized using feature specific rules. Below we show how some of the features are handled and demonstrate the application of the rules to the example LSC shown in Sect. 3.

Universal vs. Existential. List. 1.1 shows two rules. The rule `universal` (lines 1-13) matches the feature `Universal` in the configuration `Conf`. It stores the universal quantification enumerator in a global variable and if a pre-chart exists it connects its unwinding structure to the main chart's unwinding structure. In addition, it generates transitions that represent the LSC's successful traversal. Next, it creates a reject state, turns it into a sink vertex and adds it to the set F of the acceptance condition. Finally, it sets the set E of the acceptance condition to include all cold states. The rule is applied iff the input configuration includes the feature `Universal`.

For example, in Fig. 2, the resulting contribution of the rule `universal` is marked in red: it consists of the transitions going from states q_2 and q_3 to

```
 1 rule universal {
 2   from f : Conf!Universal
 3   do {
 4     self.initialStateQuant <- #Universal;
 5     if (self.isPrechartSet)
 6       self.joinCharts();
 7     self.generateBackTransitions();
 8     self.generateRejectState();
 9     self.selfTrans(self.mappedSymbols, self.rejectState);
10     self.addToFstates(self.rejectState);
11     for (state in self.unwindingStructureStates) {
12       if (self.isCold(state))
13         self.addToEstates(state);}}}
14 rule existential {
15   from f : Conf!Existential
16   do {
17     self.initialStateQuant <- #Existential;
18     self.ignoreTemperature <- true;
19     self.connectAcceptingSinkState();}}
```

Listing 1.1. ATL rules to support existential and universal semantics

state q_0 as well as the additional state q_{reject} and its self-transition. With this construction, a run that starts $(heat, reachMax, lockPanel)$ returns to the initial state q_0, ready for another activation of the LSC.

The rule `existential` (lines 14-19) matches the feature `Existential` in the configuration `Conf`. It stores the existential quantification enumerator in a global variable and instructs the transformation to ignore message temperature. It is applied iff the input configuration includes the feature `Existential`.

As an example, in Fig. 2, the resulting contribution of the rule `existential` is marked in blue: it consists of the transitions going from states q_2 and q_3 to state q_4. After applying this rule, a run that starts $(heat, reachMax, lockPanel)$ reaches the state q_4 and stays there forever.

Invariant vs. Iterative. List. 1.2 shows two rules. The rule `invariant` (List. 1.2 lines 1-9) matches the feature `Invariant` in the configuration `Conf`. It creates a self-transition (loop) on the initial state, labeled with all used alphabet symbols, and sets the quantification on the transitions outgoing the initial state to the quantification stored by the `universal` / `existential` rules. If a pre-chart is present, it sets the quantification on transitions outgoing the initial state of the main chart in a similiar manner. It is applied iff the input configuration includes the feature `Invariant`.

Fig. 2 demonstrates the contribution of the rule `invariant`: a self-transition on q_0 and a quantification on q_0's outgoing transitions (marked in light blue). For instance, in combination with universal strict semantics, a run that starts $(heat, heat)$ branches out to three states: q_0, q_1 and q_{reject}, representing three copies of the LSC.

```
1  rule invariant {
2    from f: Conf!Invariant
3    do {
4      self.selfTransMappedSymbols(self.initialState);
5      self.initialState.quantification <-
6                self.initialStateQuant;
7      if (self.isPrechartSet)
8        self.prechartToMainState.quantification <-
9                self.prechartToMainStateQuant;}}
10 rule iterative {
11   from f: Conf!Iterative
12   do {
13     self.selfTransUnboundedSymbols(self.initialState);}}
```

Listing 1.2. ATL rules to support invariant and iterative semantics

The rule `iterative` (List. 1.2 lines 10-13) matches the feature `Iterative` in the configuration `Conf`. It creates self-transitions on the initial state, for all symbols that do not yet appear on any outgoing transition from the initial state. It is applied iff the input configuration includes the feature `Iterative`.

Strict vs. Tolerant. List. 1.3 shows two rules. The rule `strict` (lines 1-7) matches the feature `Strict` in the configuration `Conf`. It iterates over the states in the unwinding structure and for each state creates outgoing transitions either to the initial state or to the reject state, in correspondence to the messages causing violations in it. It is applied iff the input configuration includes the feature `Strict`.

Fig. 2 shows the contribution of the rule `strict` in the context of universal semantics. It includes three transitions going from $q1, q2$ and $q3$ to q_{reject} (marked in purple). For instance, all the runs that start $(heat, heat)$ visit q_{reject} and stay there forever.

The rule `tolerant` (lines 8-12) matches the feature `Tolerant` in the configuration `Conf`. It iterates over the states in the unwinding structure and for each state generates a self-transition that carryies all the symbols that are included in any outgoing transition. It is applied iff the input configuration includes the feature `Tolerant`.

The contribution of the rule `tolerant` is depicted in Fig. 2: three self-transitions on the states $q1, q2$ and $q3$ (marked in orange). For instance, in this case the infinite word $(heat, heat, reachMax, lockPanel)^{\omega}$ is accepted by both in universal and existential semantics.

Environment assumptions. List. 1.4 shows two rules. The rule `assumptionLsc` (lines 1-5) matches the feature `AssumptionLsc` in the configuration `Conf`. It sets the set E of the acceptance condition to include all states. It is applied iff the input configuration includes the feature `AssumptionLsc`.

The rule `envAssumpIntegrated` (lines 6-16) matches the feature `EnvAssump-Integrated` in the configuration `Conf`. It sets the set F of the acceptance condition to include all cold-environment-hot-system states and the set E of the

```
rule strict {
  from f: Conf!Strict
  do {
    for (state in self.unwindingStructureStates) {
      if (state <> self.initialState and
        state <> self.acceptingSinkState)
        self.violatingTrans(state);}}}
rule tolerant {
  from f: Conf!Tolerant
  do {
    for (state in self.unwindingStructureStates) {
      self.addUnboundedSymbolsToSelfTrans(state);}}}
```

Listing 1.3. ATL rules to support tolerant and strict semantics

```
rule assumptionLSC {
  from f: Conf!assumptionLSC
  do {
    for (state in Automaton!State.allInstances()) {
      self.E_states <- self.E_states.including(state);}}}
rule envAssumpIntegrated {
  from f: Conf!envAssumpIntegrated
  do {
    self.F_states <- Set{};  self.E_states <- Set{};
    if (not self.rejectState.oclIsUndefined())
      self.F_states <- Set{self.rejectState};
    for (state in self.unwindingStructureStates) {
      if (self.isCEHS(self.matchingCut(state)))
        self.F_states <- self.F_states.including(state);
      else if (self.isCS(self.matchingCut(state)))
        self.E_states <- self.E_states.including(state);}}}
```

Listing 1.4. ATL rules to support assumptions

acceptance condition to include all cold-system states. It is applied iff the input configuration includes the feature `EnvAssumpIntegrated`. In the example shown in Fig. 2, in the context of universal semantics and in the case where the feature `EnvAssump` is not selected, the acceptance condition is set to $F = Q$ (the complete set of states) and $E = \{q_0, q_4\}$. In the case where the features `EnvAssump` and `EnvAssumpIntegrated` are selected, the acceptance condition is set to $F = \{q_3, q_{reject}\}$ and $E = \{q_0, q_2, q_4\}$.

6 Implementation, Validation, and Discussion

Implementation. We have created a prototype implementation of our work, packaged as an Eclipse plug-in. For the representation of the LSC semantics feature model and the selection of valid configurations we use components from FeatureIDE [19]. For editing LSCs we use the UI and APIs of PlayGo [14]. The

M2M transformation is implemented in ATL [18]. The prototype plug-in together with several examples is available from [28].

Validation. We validated our solution as follows. First, we implemented tests that iterate and apply all possible configurations of the feature model to a set of non-trivial LSCs, and check various properties of the resulting automata (e.g., the existence of a specific self-transition etc.). The tests are available from [28].

Second, we used the output of our solution as input for GOAL [42] (a Graphical Tool for ω-Automata and Logics), and thus executed several usage scenarios, including (1) verifying an automaton against a property LSC, (2) checking the consistency of an LSC specification, while applying several, different semantic configurations. Details of some of these examples of usage scenarios are available in Appendix B.

Choice of variability modeling language. Our choice of feature diagrams as a variability modeling language was motivated by its wide use in the literature, its tool support (we use FeatureIDE [19]) and its expressive power, which is good enough for our purposes. Considering other means to model variability in our context, e.g., the Common Variability Language (CVL) [17], is outside the scope of this paper.

Choice of target formalism. Our choice of alternating one pair Streett automata as the target formalism for the semantics definition was motivated by its expressive power, which covers all variants found in the literature. Alternatively, one may use a Buchi acceptance condition, however, we consider this to be less intuitive for the variants involving environment assumptions.

Choice of transformation language. Our choice of ATL for the implementation of the model-to-model transformation was motivated by its tool and standard support. It allowed us to create a rather high-level readable code that reflects the one-to-one mapping between features and rules.

7 Conclusion

In this paper we have investigated the idea of semantically configurable analysis in the context of live sequence charts. We formalized semantic variability in LSC using a feature model and presented a semantically configurable fully automated analysis solution based on a transformation to an alternating one pair Streett automaton, capable of expressing all semantic variants found in the literature. The work was implemented in an Eclipse plug-in and demonstrated with examples.

We consider the following possible future work. First, extending our work to support additional LSC language features, e.g., conditions and various interaction fragments (alternatives, loops). The semantics of some of these language features does not seem to vary in the literature, but they are used extensively and are thus necessary for a more comprehensive solution. Second, integrating our semantically configurable analysis with existing tools that are using LSC, such as PlayGo [14] and ScenarioTools [10,39].

The paper continues our previous work on semantically configurable analysis [32] and is part of our larger project on investigating formal and automated ways to handling variability in modeling languages syntax and semantics.

References

1. D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
2. M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within modeling language definitions. In *MoDELS*, volume 5795 of *LNCS*, pages 670–684. Springer, 2009.
3. K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, 2000.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
5. W. Damm, T. Toben, and B. Westphal. On the expressive power of live sequence charts. In *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 225–246. Springer, 2006.
6. W. Damm and B. Westphal. Live and let die: LSC-based verification of UML-models. In *FMCO*, volume 2852 of *LNCS*, pages 99–135. Springer, 2002.
7. D. Fischbein, N. D'Ippolito, G. Sibay, and S. Uchitel. Modal Transition System Analyzer (MTSA). http://sourceforge.net/projects/mtsa/. Accessed 9/13.
8. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.
9. J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *ES-EC/SIGSOFT FSE*, pages 433–443. ACM, 2013.
10. J. Greenyer, C. Brenner, and V. P. L. Manna. The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions. *ECEASST*, 58, 2013.
11. J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans. Efficient consistency checking of scenario-based product-line specifications. In *RE*, pages 161–170. IEEE, 2012.
12. D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.
13. D. Harel, S. Maoz, and I. Segall. Some results on the expressive power and complexity of LSCs. In *Pillars of Computer Science*, volume 4800 of *LNCS*, pages 351–366. Springer, 2008.
14. D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, pages 359–360. ACM, 2010.
15. D. Harel and R. Marelly. *Come, let's play - scenario-based programming using LSCs and the play-engine*. Springer, 2003.
16. D. Harel and I. Segall. Synthesis from scenario-based specifications. *J. Comput. Syst. Sci.*, 78(3):970–980, 2012.
17. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, pages 139–148. IEEE Computer Society, 2008.
18. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
19. C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *ICSE*, pages 611–614, 2009.
20. J. Klose. *Live sequence charts: a graphical formalism for the specification of communication behavior*. PhD thesis, University of Oldenburg, 2003.

21. J. Klose, T. Toben, B. Westphal, and H. Wittke. Check it out: On the efficient formal verification of live sequence charts. In *CAV*, volume 4144 of *LNCS*, pages 219–233. Springer, 2006.
22. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In *TACAS*, volume 3440 of *LNCS*, pages 445–460. Springer, 2005.
23. H. Kugler, C. Plock, and A. Pnueli. Synthesizing reactive systems from LSC requirements using the play-engine. In *OOPSLA Companion*, pages 801–802. ACM, 2007.
24. O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001.
25. K. G. Larsen, S. Li, B. Nielsen, and S. Pusinskas. Scenario-based analysis and synthesis of real-time systems using Uppaal. In *DATE*, pages 447–452. IEEE, 2010.
26. S. Li, S. Balaguer, A. David, K. G. Larsen, B. Nielsen, and S. Pusinskas. Scenario-based verification of real-time systems using Uppaal. *Formal Methods in System Design*, 37(2-3):200–264, 2010.
27. D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. *Autom. Softw. Eng.*, 19(4):423–458, 2012.
28. LSC semantic variability supporting materials. http://smlab.cs.tau.ac.il/lscvar/.
29. Y. Lu, J. M. Atlee, N. A. Day, and J. Niu. Mapping template semantics to SMV. In *ASE*, pages 320–325. IEEE Computer Society, 2004.
30. S. Maoz. Polymorphic scenario-based specification models: semantics and applications. *Software and Systems Modeling*, 11(3):327–345, 2012.
31. S. Maoz, D. Harel, and A. Kleinbort. A compiler for multimodal scenarios: Transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 20(4):18, 2011.
32. S. Maoz, J. O. Ringert, and B. Rumpe. Semantically configurable consistency analysis for class and object diagrams. In *MoDELS*, volume 6981 of *LNCS*, pages 153–167. Springer, 2011.
33. S. Maoz and Y. Sa'ar. Assume-guarantee scenarios: Semantics and synthesis. In *MoDELS*, volume 7590 of *LNCS*, pages 335–351. Springer, 2012.
34. S. Maoz and Y. Sa'ar. Counter play-out: executing unrealizable scenario-based specifications. In *ICSE*, pages 242–251. IEEE / ACM, 2013.
35. M. Mendonça, M. Branco, and D. D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA Companion*, pages 761–762, 2009.
36. Z. Micskei and H. Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software and Systems Modeling (SoSyM)*, 10(4):489–514, 2011.
37. J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE Trans. Software Eng.*, 29(10):866–882, 2003.
38. A. Prout, J. M. Atlee, N. A. Day, and P. Shaker. Code generation for a family of executable modelling notations. *Software and Systems Modeling*, 11(2):251–272, 2012.
39. ScenarioTools. http://www.scenariotools.org/. Accessed 9/13.
40. G. E. Sibay, V. A. Braberman, S. Uchitel, and J. Kramer. Synthesizing modal transition systems from triggered scenarios. *IEEE Trans. Software Eng.*, 39(7):975–1001, 2013.
41. G. E. Sibay, S. Uchitel, and V. A. Braberman. Existential live sequence charts revisited. In *ICSE*, pages 41–50. ACM, 2008.
42. Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In *TACAS*, volume 4424 of *LNCS*, pages 466–471. Springer, 2007.

A Alternating Streett Automaton

The definitions of alternating and Streett automata are well known and appear for example in [8]. We use here a combined definition, specific to our needs. The notation is adopted from [24].

For a given set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas `true` and `false`. For $Y \subseteq X$, we say that Y *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ iff the truth assignment that assigns `true` to the members of Y and assigns `false` to the members of $X \setminus Y$ satisfies θ.

An *alternating Streett automaton on infinite words* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, S \rangle$ where Σ is an input alphabet, Q is a finite set of states, $q_{in} \in Q$ is an initial state, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is a transition function, and $S = \{(F_i, E_i) \mid 1 \leq i \leq l\}$ is a family of pairs of sets of states serving as a Streett acceptance structure.

A run of an alternating automaton is a tree $r : Tr \to Q$ for some $Tr \subseteq \mathbb{N}^*$. Formally, a tree is a (finite or infinite) nonempty prefix-closed set $T \subseteq \mathbb{N}^*$. The elements of T are called nodes, and the empty word ϵ is the root of T. For every $x \in T$, the nodes $x \cdot c \in T$ where $c \in \mathbb{N}$ are the children of x. A node with no children is a `leaf` . We refer to the length $|x|$ of x as its `level` in the tree. A path π of a tree T is a set $\pi \subseteq T$ such that $\epsilon \in \pi$ and for every $x \in \pi$, either x is a leaf, or there exists a unique $c \in \mathbb{N}$ such that $x \cdot c \in \pi$. Given a finite set Θ, a Θ`-labeled tree` is a pair $\langle T, V \rangle$ where T is a tree and $V : T \to \Theta$ maps each node of T to a letter in Θ. A run of $\mathcal{A}$ on an infinite word $w = \sigma_0 \cdot \sigma_1 \cdots$ is a Q-labeled tree $\langle Tr, r \rangle$ such that the following hold:

- $r(\epsilon) = q_{in}$.
- Let $x \in Tr$ with $r(x) = q$ and $\delta(q, \sigma_{|x|}) = \theta$. There is a (possibly empty) set $R = \{q_1, \ldots, q_k\}$ such that R satisfies θ and for all $1 \leq c \leq k$, we have $x \cdot c \in Tr$ and $r(x \cdot c) = q_c$.

For example, if $\delta(q_{in}, \sigma_0) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$, then possible runs of $\mathcal{A}$ on w have a root labeled q_{in}, have one node in level 1 labeled q_1 or q_2, and have another node in level 1 labeled q_3 or q_4. Note that if $\theta =$ `true`, then x need not have children. This is the reason why Tr may have leaves. Also, since there exists no set S as required for $\theta =$ `false`, we cannot have a run that takes a transition with $\theta =$ `false`.

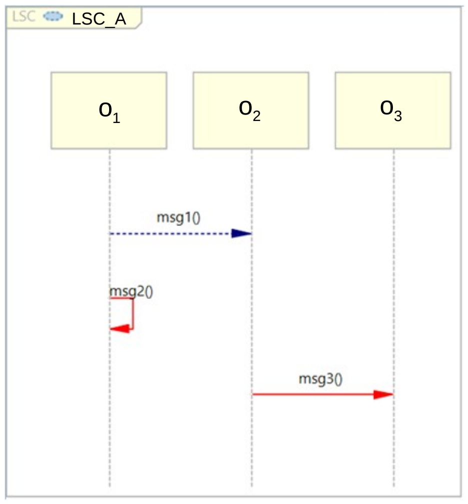
Let $\mathit{inf}(\pi)$ be the set of states that r visits infinitely often. A path π is accepting iff it satisfies Streett's acceptance condition on the structure S, i.e., for all $1 \leq i \leq l$ if $\mathit{inf}(\pi) \cap F_i \neq \emptyset$ then $\mathit{inf}(\pi) \cap E_i \neq \emptyset$. A run $\langle Tr, r \rangle$ is accepting if all its infinite paths, which are labeled by words in Q^ω , are accepting. A word w is accepted by the automaton $\mathcal{A}$ if there exists an accepting run on it.

B Examples of analysis usage scenarios

We show three examples of semantically configurable analysis of LSCs, as supported by our prototype implementation.

B.1 LSC verification with strict vs. tolerant semantics

Fig. 4 shows a property LSC `LSC_A` and a controller `C1`. A controller satisfies a property LSC iff the language defined by the controller is contained in the language defined by the LSC. Does `C1` satisfy `LSC_A`? The answer depends on the choice of semantics.



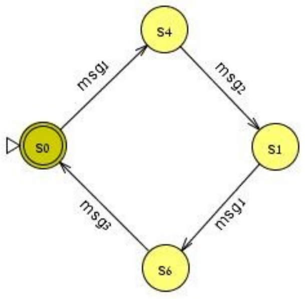


Fig. 4. Property LSC `LSC_A` and controller `C1`

Consider two semantic configurations:
$conf_s = \{universal, interleaving, iterative, strict\}$
$conf_t = \{universal, interleaving, iterative, tolerant\}$

The two configurations are legal configurations of the LSC semantics feature model (we use a minimal representation, which omits core and derived features). Note that $conf_s$ and $conf_t$ differ only in the choice between the two kinds of sequencing, strict and tolerant.

Given `LSC_A` and the two configurations, we used our prototype implementation to generate the two automata shown in Fig. 5 (we set the tool to generate these automata using Buchi acceptance condition rather than Streett acceptance condition, in order to be able to preform analysis later on in GOAL [42]). The two automata are similar. However, they differ in the labeling on the self-transitions and the reachability of the reject state (`S5`).

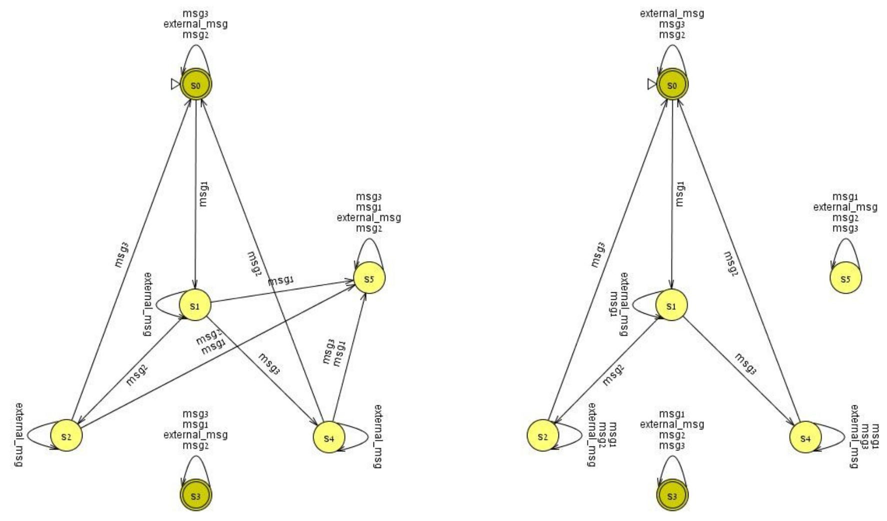


Fig. 5. Two automata, generated by our tool for `LSC_A` and the two semantic configurations $conf_s$ (left) and $conf_t$ (right)

We then used GOAL to check whether the controller `C1` satisfies `LSC_A` under $conf_s$ and under $conf_t$ (by checking language containment). Indeed, GOAL reported that the language accepted by `C1` is contained in the language defined by the automaton that was generated with $conf_t$. In contrast, GOAL reported that the language accepted by `C1` is not contained in the language defined by the automaton that was generated with $conf_s$ and gave the word $(msg1, msg2, msg1)(msg3, msg1, msg2, msg1)^{\omega}$ as a counterexample.

B.2 LSC verification with existential vs. universal semantics

Consider the property LSC `LSC_B` and the controller `C2`, shown in Fig. 6, left and top right respectively. Does `C2` satisfy `LSC_B`? Here too, the answer depends on the choice of semantics.

Consider two semantic configurations:
$conf_e = \{existential, interleaving, iterative, strict\}$
$conf_u = \{universal, interleaving, iterative, strict\}$

Note that $conf_u$ and $conf_e$ differ only in the choice between the two kinds of quantification, universal and existential.

Given `LSC_B` and the two configurations, we used our prototype implementation to generate the two automata shown in Fig. 6: $\mathcal{A}_e$ (middle right) and $\mathcal{A}_u$ (bottom right). The two automata are similar. However, while in $\mathcal{A}_e$ the initial state is not accepting and the accepting sink state (`S2`) is reachable, in $\mathcal{A}_u$ the initial state is accepting and the accepting sink state (`S2`) is unreachable. In

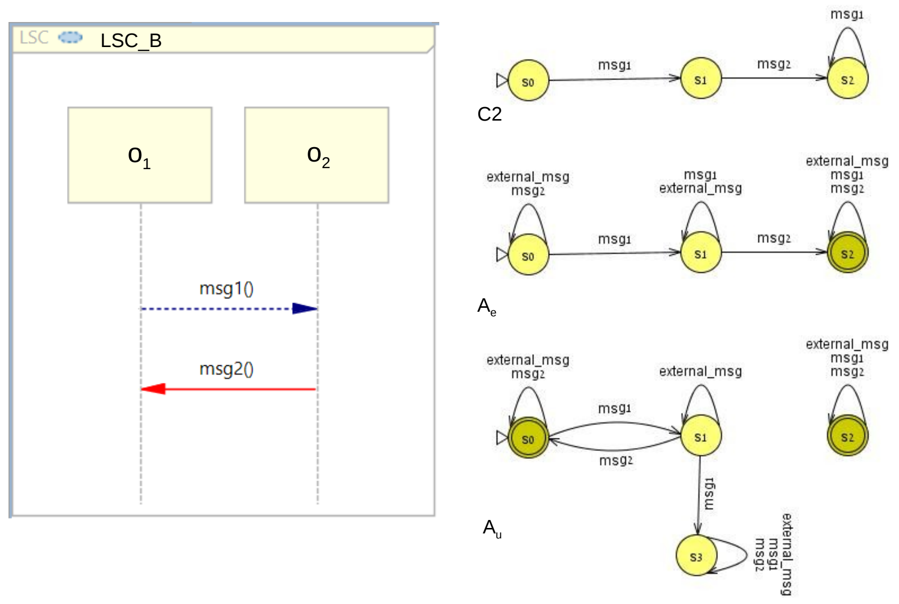


Fig. 6. Property LSC `LSC_B` (left), controller `C2` (top right), and two automata, $\mathcal{A}_e$ and $\mathcal{A}_u$, generated by our tool for `LSC_B` and the two semantic configurations $conf_e$ (middle right) and $conf_u$ (bottom right)

addition, $\mathcal{A}_u$ contains a reject state (`S3`) and a transition that goes back to the initial state, while $\mathcal{A}_e$ contains neither.

We then used GOAL to check whether the controller `C2` satisfies `LSC_B` under $conf_e$ and under $conf_u$ (by checking language containment). Indeed, GOAL reported that the language accepted by `C2` is contained in the language defined by the automaton $\mathcal{A}_e$ that was generated with $conf_e$. In contrast, GOAL reported that the language accepted by `C2` is not contained in the language defined by the automaton $\mathcal{A}_u$ that was generated with $conf_u$ and gave the word $(msg1, msg2, msg1, msg1)(msg1, msg1)^\omega$ as a counterexample.

B.3 LSC consistency with existential vs. universal semantics

Fig. 7 shows three property LSCs: `LSC_C`, `LSC_D` and `LSC_E`. A set of LSCs is consistent iff the intersection of the languages defined by all LSCs is not empty. Is the set {`LSC_C`, `LSC_D`, `LSC_E`} consistent? Again, the answer depends on the choice of semantics.

Consider two semantic configurations:
$conf_e = \{existential, interleaving, iterative, strict\}$
$conf_u = \{universal, interleaving, iterative, strict\}$

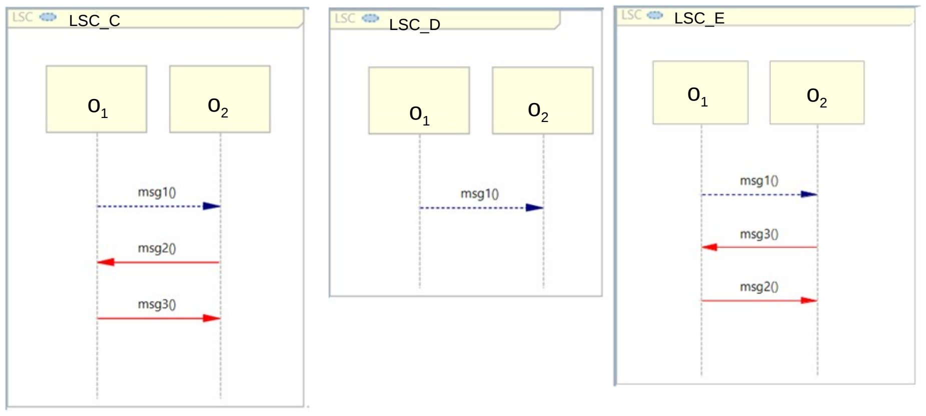


Fig. 7. Three property LSCs: `LSC_C`, `LSC_D`, and `LSC_E`

Note that $conf_e$ and $conf_u$ differ only in the choice between the two kinds of quantification, universal and existential.

Given `LSC_E` and the two configurations, we used our prototype implementation to generate the two automata shown in Fig. 8: $\mathcal{A}_e$ (left) and $\mathcal{A}_u$ (right). The two automata are different. For instance, while in $\mathcal{A}_e$ the initial state is not accepting and the accepting sink state (`S3`) is reachable, in $\mathcal{A}_u$ the initial state is accepting and the accepting sink state (`S3`) is unreachable. In addition, $\mathcal{A}_u$ contains a reject state (`S4`) while $\mathcal{A}_e$ does not.

Suppose that $conf_{LSC\_C} = \{existential, interleaving, iterative, strict\}$ and $conf_{LSC\_D} = \{existential, interleaving, iterative, strict\}$ are the semantic configuration for `LSC_C` and `LSC_D` respectively. Again, given `LSC_C`, `LSC_D`, and the

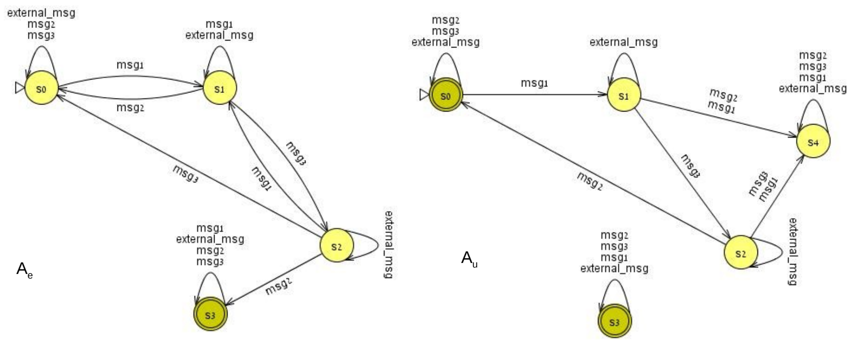


Fig. 8. Two automata, $\mathcal{A}_e$ and $\mathcal{A}_u$, generated by our tool for `LSC_A` and the two semantic configurations $conf_e$ (left) and $conf_u$ (right)

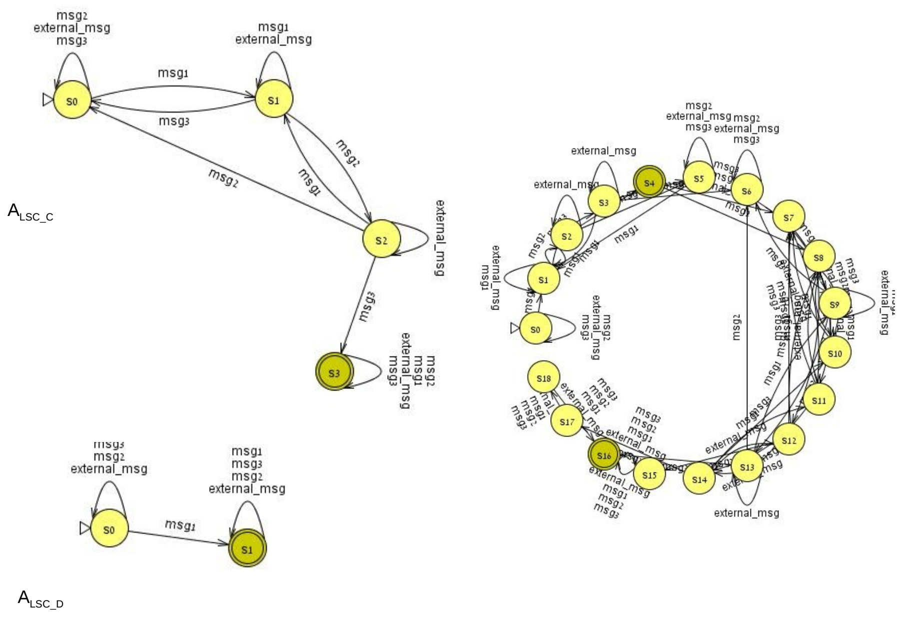


Fig. 9. An automaton, $\mathcal{A}_{LSC\_C}$, generated by our tool for `LSC_C` and the semantic configurations $conf_{LSC\_C}$ (top left); an automaton, $\mathcal{A}_{LSC\_D}$, generated by our tool for `LSC_D` and the semantic configurations $conf_{LSC\_D}$ (bottom left); and the automaton, generated by GOAL, that accepts the intersection of the languages defined by `LSC_C`, `LSC_D`, and `LSC_E` in the case where the semantics of `LSC_E` is set to $conf_e$

two configurations, we used our prototype implementation to generate the two automata shown in Fig. 9: $\mathcal{A}_{LSC\_C}$ (top left) and $\mathcal{A}_{LSC\_D}$ (bottom left).

We then used GOAL to check whether the set $\{$`LSC_C`, `LSC_D`, `LSC_E`$\}$ is consistent in the case where the semantics of `LSC_E` is set to $conf_e$, and in the case where the semantics of `LSC_E` is set to $conf_u$. To do so, we asked GOAL to find the intersection of the languages defined by $\mathcal{A}_{LSC\_C}$, $\mathcal{A}_{LSC\_D}$, and $\mathcal{A}_e$, as well as the intersection of the languages defined by $\mathcal{A}_{LSC\_C}$, $\mathcal{A}_{LSC\_D}$ and $\mathcal{A}_u$.

Indeed, GOAL reported that $\mathcal{L}(\mathcal{A}_{LSC\_C}) \cap \mathcal{L}(\mathcal{A}_{LSC\_D}) \cap \mathcal{L}(\mathcal{A}_u)$ is empty. In contrast, GOAL reported that $\mathcal{L}(\mathcal{A}_{LSC\_C}) \cap \mathcal{L}(\mathcal{A}_{LSC\_D}) \cap \mathcal{L}(\mathcal{A}_e)$ is not empty and gave the word $(msg1, msg3, msg2, msg1, msg2, msg3)(msg3, msg3)^{\omega}$ as a witness. GOAL generated the automaton that accepts $\mathcal{L}(\mathcal{A}_{LSC\_C}) \cap \mathcal{L}(\mathcal{A}_{LSC\_D}) \cap \mathcal{L}(\mathcal{A}_e)$. See Fig. 9, right.