Counter Play-Out: Executing Unrealizable Scenario-Based Specifications

Shahar Maoz
School of Computer Science
Tel Aviv University, Israel

Yaniv Sa'ar
Dept. of Computer Science
Weizmann Institute of Science, Israel

***Abstract*—The scenario-based approach to the specification and simulation of reactive systems has attracted much research efforts in recent years. While the problem of synthesizing a controller or a transition system from a scenario-based specification has been studied extensively, no work has yet effectively addressed the case where the specification is unrealizable and a controller cannot be synthesized. This has limited the effectiveness of using scenario-based specifications in requirements analysis and simulation.**

In this paper we present *counter play-out*, an interactive debugging method for unrealizable scenario-based specifications. When we identify an unrealizable specification, we generate a controller that plays the role of the environment and lets the engineer play the role of the system. During execution, the former chooses environment's moves such that the latter is forced to eventually fail in satisfying the system's requirements. This results in an interactive, guided execution, leading to the root causes of unrealizability. The generated controller constitutes a proof that the specification is conflicting and cannot be realized.

Counter play-out is based on a counter strategy, which we compute by solving a Rabin game using a symbolic, BDD-based algorithm. The work is implemented and integrated with PlayGo, an IDE for scenario-based programming developed at the Weizmann Institute of Science. Case studies show the contribution of our work to the state-of-the-art in the scenario-based approach to specification and simulation.

I. Introduction

The scenario-based approach to the specification and execution of reactive systems has attracted much research efforts over the last decade [9], [34]–[36]. The approach is based on two main ideas. The first is that scenarios — short 'stories' of interaction between system entities (and their environment) — provide an intuitive and natural way to think about and capture complex reactive behavior. Each scenario typically focuses on a specific feature or concern from the requirements of the system under development. The scenarios are often extended with modalities distinguishing mandatory, optional, and negative interactions. The second idea is that simulating a specification by synthesizing a transition system or a controller that meets its requirements provides useful and effective means for requirements analysis and rapid prototyping.

Several different scenario-based languages have been proposed and the problem of synthesizing a controller from a scenario-based specification has been studied extensively by many authors [4], [8], [11], [16], [18], [20], [21], [32], [34]–[36]. However, no work has yet effectively addressed the case where the scenario-based specification is unrealizable and a controller cannot be synthesized. For effective debugging of unrealizable scenario-based specifications one should not only identify unrealizability but also present its causes, using the abstractions defined by the scenarios.

To better understand the challenge of handling an unrealizable specification, in the case of synthesis, one may contrast it with the case of model checking. In a model-checking setup, an implementation (or a representation thereof) is checked against a specification. If the implementation does not satisfy the specification, a counter example is provided, illustrating how the given implementation can violate the specification. In a synthesis setup, however, the implementation is not part of the input. If the specification is realizable, a correct-by-construction implementation can be generated. However, if the specification is unrealizable, there is no implementation one could check against and thus no direct way to finding the causes of unrealizability.

In this work we present a debugging method for unrealizable scenario-based specifications. We do this in the context of live sequence charts (LSC) [6], [13] — a visual, formal, and expressive scenario-based specification language — and its play-out execution mechanism [15], [22], [23].

First, we show how to identify an unrealizable specification and compute a counter strategy, a strategy that shows how an adverse environment may force any system to eventually fail in satisfying the specification. This statically computed counter strategy constitutes a proof that the specification is unrealizable. We compute it by reducing the LSC specification into a Rabin game [29] and then solving this game using a BDD-based symbolic algorithm, based on [17], [27].

Second, we use the computed counter strategy to develop an interactive debugging method we term *counter play-out*. When an unrealizable specification is identified, we generate a reversed-roles controller that plays the role of the environment and lets the engineer play the role of the system. During its execution, the controller chooses environment's moves such that the engineer is forced to eventually fail in satisfying the system's requirements. This results in an interactive, guided execution, leading to the root causes of unrealizability, i.e., to the specific scenarios whose violation is unavoidable.

Thus, while we statically compute a complete counter strategy and present it to the engineer in a textual and visual tree-like formats, understanding it is difficult. To address this difficulty, counter play-out demonstrates the unrealizability

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

of the specification to the engineer in an interactive, direct experiential way. Note that in debugging a reactive system, a single execution path does not suffice to prove unrealizability; instead, guided interactivity is an advantage, as it allows to simulate and examine the choices made by both system and environment over time. Moreover, since we deal with scenario-based specifications, during execution we present all information about progress and violations to the engineer, using the abstractions defined by the scenarios (which scenarios were activated in the execution, which scenarios are violated etc.). We consider this to be a key contribution of our work.

Counter play-out is implemented in PlayGo [14], an eclipse-based IDE for scenario-based specification and programming developed at the Weizmann Institute of Science. At the back end, PlayGo computes the counter strategy using JTLV [28]. Code generation from LSC specifications is implemented using the S2A compiler [22], [23]. In addition to a textual output, graphical interface, which shows the scenarios' progress in sequence diagram format and the counter play-out session itself in an annotated tree-like format, assists the engineer in tracing the execution and pointing to the causes for unrealizability. We tested our implementation and its performance on many example specifications and validated its feasibility, see Sect. V.

Finally, on a more general note, one may refrain from using synthesis to create a final implementation, perhaps due to concerns about the scalability of the synthesis algorithm and the efficiency of the synthesized controller. Yet, checking for unrealizability and its causes may be done with very partial specifications and already with raw ideas during early design. Thus, the potential applicability of identifying unrealizable specifications and addressing them with an interactive debugging approach is not limited to setups where one plans to use synthesis to create a final implementation. We consider this to be an important feature of our work.

The next section provides background on LSC, play-out, and synthesis. Sect. III explains how we compute counter strategies. Sect. IV presents the counter play-out method using a running example. We discuss implementation and evaluation in Sect. V. Sect. VI discusses related work. Sect. VII concludes.

II. Preliminaries

A. Live Sequence Charts

Live sequence charts (LSC) [6], [13] is a scenario-based specification language, which extends classical message sequence charts (MSC) mainly with a universal interpretation and a distinction between mandatory and possible behavior. Importantly, LSC has an operational, executable semantics termed play-out [15]. We give a simplified language overview, emphasising the parts most relevant to our present work. More detailed descriptions are available in [6], [13], [15].

An LSC consists of lifelines, messages, and conditions. A *lifeline* represents an interacting entity, controlled either by the system under development or by its environment (other systems, users etc.). A *message* represents a call between one entity and another. A message is a *system message* (resp. *environment message*) if it is sent from a lifeline controlled by the system (resp. the environment). The LSC defines a partial order on its messages, induced by the vertical ordering of messages sent and received along the lifelines.

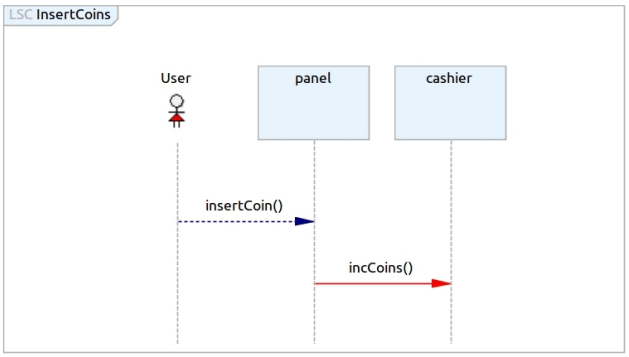

Fig. 1. The LSC `InsertCoins` of the vending machine specification

As an example, Fig. 1 shows the LSC `InsertCoins`, taken from a specification of a vending machine (which we use later in this paper as a running example). This LSC has one environment lifeline (controlled by the user) and two system lifelines, representing the system's `panel` and `cashier`; `insertCoin` is an environment message and `incCoins` is a system message.

The current state of an LSC is represented by a *system cut*, marking the progress of events along the LSC's lifelines. A cut induces a set of enabled and violating messages and conditions: a message is *enabled* in a cut if it appears immediately after the cut in the partial order defined by the chart; a message is *violating* in a cut of a chart if it appears in the chart, but is not enabled in the cut.

Messages have a hot or a cold temperature (red line or blue line syntax): a hot enabled message must eventually occur, while a cold enabled message may or may not eventually occur. A cut is hot if at least one of its enabled system messages is hot, and is cold otherwise. When an enabled message occurs, the chart progresses to the next cut. When a violating message occurs, progress depends on the temperature of the cut: if the cut was cold, the chart closes gracefully (the cut is set to be the minimal cut); if the cut was hot, it is a violation of the requirements, which should have never occurred. In the LSC `InsertCoins` (Fig. 1), the first message is cold and the second is hot.

Conditions have a hot or a cold temperature too and they are evaluated as soon as they are enabled. A hot enabled condition must be evaluated to true, while a cold enabled condition may or may not be evaluated to true. When a condition (hot or cold) is evaluated to true, the chart progresses to the next cut. When a condition is evaluated to false, progress depends on its temperature: if it was cold, the chart closes gracefully (the cut is set to be the minimal cut); if is was hot, it is a violation of the requirements, which should have never occurred.

System messages can be marked for either *execution* (solid line) or *monitoring* (dashed line). All environment messages are marked as monitoring. A chart is considered to be *active* if its current cut has an enabled (system) message for exe-

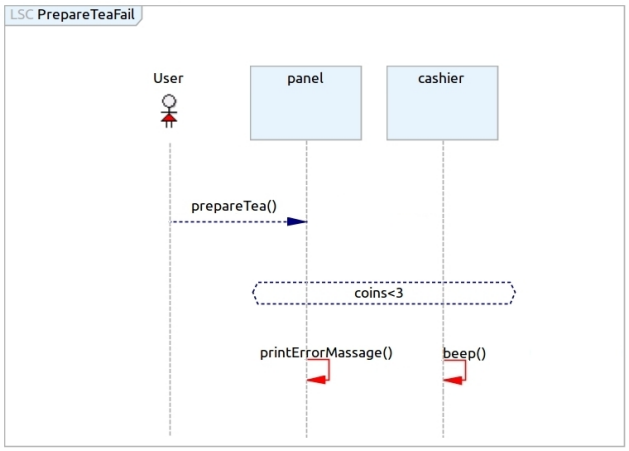

Fig. 2. The LSC `PrepareTeaFail` of the vending machine specification

cution. Similarly, a chart is considered to be *expecting* if its current cut has an enabled environment message. In the LSC `InsertCoins` of Fig. 1, the first message is marked for monitoring while the second message is marked for execution. As another example, Fig. 2 shows the LSC `PrepareTeaFail` taken from the same vending machine specification. This LSC has three lifelines. The condition `coins < 3` is a cold condition. The two hot messages `printErrorMessage` and `Beep` are not ordered.

B. LSC Semantics

The complete semantics of LSC and its formulation in the GR(1) form is available in [25]. We give here an overview.

The semantics of a single LSC uses the partial order on messages and conditions defined by the chart, adds a universal interpretation, and relates to the hot (mandatory) and cold (optional) elements in the chart. Messages that do not appear in a chart are not constrained by the chart to occur or not to occur at any time, including in between the occurrence of messages that do appear in the chart.

For example, the semantics of the chart `InsertCoins` of Fig. 1 specifies that *"whenever the environment sends the message `insertCoin` to the system's panel (the user has inserted a coin), eventually the system's panel must send the message `incCoins` to the system's cashier"*. Implicitly, this also means that after `insertCoin` occurs, the system message `incCoins` must come before another `insertCoin` message is sent by the environment. As another example, the semantics of the chart `PrepareTeaFail` of Fig.2 requires that *"whenever the user asks to prepare tea (sends the `prepareTea` message to the panel), the cold condition `coins < 3` is immediately evaluated. If it is true, the system should eventually play a beep sound and show an error message (in no specific order). If it is false, nothing more is required"* (note that if the condition was a hot condition, the semantics would have required that it is evaluated to true).

The semantics of an LSC specification, consisting of a set of LSCs, is defined using a two-player Streett game [33] between the environment and the system. In order to win the game, assuming the environment meets the assumptions specified in the scenarios, the system is required to fulfil all the guarantees specified in the scenarios.

More formally, the semantics of an LSC specification has three parts: the superstep semantics, which defines the encapsulation of finite series of messages sent by the system between two messages sent by the environment; the application-specific system's semantics, which guarantees that the system starts from a minimal, initial state (where all charts are not active), follows its transition semantics (the progress of events along the charts lifelines, system safety), and is always eventually stable (reaching an inactive or an expecting cut infinitely often, system liveness); and the application-specific environment's semantics, which represents the assumptions that the environment will not send certain sequences of messages (environment safety), and that if the chart is in an expecting cut in which the environment promises to send a certain message, the message will eventually be sent (environment liveness).

As we showed in [25], these three parts of the semantics can be presented in the form of a single GR(1) formula, defined below.

*Definition 1 (*GR(1)*):* [2]
Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a finite set of Boolean variables, $\mathcal{X} \subseteq \mathcal{V}$ a set of input variables, and $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$ a set of output variables. The class of generalized reactive specifications of rank 1 (GR(1)) is defined to be LTL formulae of the form

$$\psi : \left(\varphi_a^e \ \wedge \ \varphi_t^e \ \wedge \ \varphi_g^e\right) \longrightarrow \left(\varphi_a^s \ \wedge \ \varphi_t^s \ \wedge \ \varphi_g^s\right) \qquad (1)$$

where:

(i) φ_a^e and φ_a^s are Boolean formulae that characterize the initial values that are assumed of the environment and guaranteed by the system, respectively.

(ii) φ_t^e and φ_t^s are formulae of the form $\bigwedge_{i \in I} \square B_i$ where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc v$ where $v \in \mathcal{X}$ and $v \in \mathcal{X} \cup \mathcal{Y}$, respectively. Intuitively, φ_t^e characterizes possible input to the controller, and φ_t^s characterizes possible transition of the controller.

(iii) φ_g^e and φ_g^s are formulae of the form $\bigwedge_{i \in I} \square \diamondsuit B_i$ where each B_i is a Boolean formula. The formula φ_g^e characterizes liveness assumptions on the environment input, and the formula φ_g^s characterizes liveness guarantees on the controller.

For example, the semantics of `PrepareTeaFail` (Fig. 2) is reflected in the GR(1) formula as follows. φ_a^s characterises the initial (minimal) cut on the three lifelines (before `prepareTea`). φ_t^s characterises the transitions specified by the chart, e.g., when the system is in the cut just after the condition, the formula will restrict either `beep` or `printErrorMessage` to occur, before any other messages that appear in the chart may occur. Finally, φ_g^s characterises the liveness property of the chart, that is, that whenever the system reaches the cut just after the condition, both messages, `beep` and `printErrorMessage` should eventually occur and the chart should be completed.

C. Play-Out Strategies

Play-out is the direct execution of LSC specifications. As the specification may contain various kinds of underspecification, due to the partial order of events within and between charts, the core of play-out is a strategy mechanism responsible for choosing the next message to send. Several play-out mechanisms exist, with increasing expressive power.

1) Naïve and smart play-out: Naïve play-out arbitrarily chooses a non-violating message from among the current set of messages that are enabled for execution in at least one chart and are not violating in any chart (if any such message exists). Smart play-out [12] considers not only the current set of enabled non-violating messages, but also looks ahead and picks up a finite sequence of messages that will lead to a successful (non-violating) superstep (if any such sequence exists) — a series of system messages leading to a state with no system messages enabled for execution. However, both naïve and smart play-out may be viewed as unsound because following them may result in executions that cannot be continued to form infinite executions satisfying the semantics of LSC (for concrete examples of the limitations of naïve and smart play-out see [10]).

2) Synthesis-based play-out: Synthesis-based play-out [16], [19], [25] uses a synthesized controller to guide the system's actions during execution. Synthesis from LSC specifications has been studied before (see Sect. VI). Unlike naïve and smart play-out, synthesis-based play-out is sound and complete with respect to the defined semantics: if the synthesis process is successful, then it is guaranteed that in every state of the system and for every environment action, play-out would respond with a series of actions that will result in an ongoing behavior that satisfies the specification.

Specifically, the solution we use for synthesis is a winning strategy. Given a GR(1) specification, computing a winning strategy for the system is done by solving a Streett game where the system tries to either satisfy all its guarantees, or constantly falsify one of the environment's assumptions, following the symbolic fixpoint algorithms described in [2]. Roughly, the algorithm starts from the set of all states and iterates 'backwards' by removing states from which the system is unable to force the execution to either reach all of the system's liveness guarantees, or constantly violate one of the environment's assumptions (each set of states where the assumption is constantly violated is computed using another nested fixpoint). The fixpoint is reached when no additional states can be removed. If to every environment initial choice there exists a system initial choice in the fixpoint set, then the specification is realizable. A controller that implements the system's winning strategy is constructed from the intermediate values of the fixpoint computation (see [2]).

If the specification is realizable, then the construction of such a controller constitutes a solution to the synthesis problem. If the specification is unrealizable, then the synthesis computation fails. Handling this case is the challenge we address in this paper.

III. Handling Unrealizable Specifications

We are now ready to present the first contribution of our work, that is, handling of unrealizable specifications by computing a counter strategy. A controller that implements a winning strategy for the system does not always exists. In such a case, as the games involved are determined, a *counter strategy* must exist, which shows how an adverse environment can force the system into not satisfying the requirements.

A. Problem Formulation

Given a GR(1) specification as defined in Def. 1, we define its dual, Rabin counterpart as follows:

Definition 2 (Dual GR(1)*):*

$$\psi : \left(\varphi_a^e \ \wedge \ \varphi_t^e \ \wedge \ \varphi_g^e\right) \ \wedge \ \left(\overline{\varphi}_a^s \ \vee \ \overline{\varphi}_t^s \ \vee \ \overline{\varphi}_g^s\right) \qquad (2)$$

where φ_a^e, φ_t^e, and φ_g^e, are defined as in Def. 1, and:

(i) $\overline{\varphi}_a^s$ is a Boolean formula that characterizes states in which the system violates its initial guarantees.

(ii) $\overline{\varphi}_t^s$ is a formula of the form $\bigvee_{i\in I} \diamondsuit \overline{B}_i$ where each $\overline{B}_i$ is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc v$ where $v \in \mathcal{X} \cup \mathcal{Y}$. Intuitively, $\overline{\varphi}_t^s$ characterizes states from which the computation eventually reaches a deadend for the system.

(iii) $\overline{\varphi}_g^s$ is a formula of the form $\bigvee_{i\in I} \diamondsuit \square \overline{B}_i$ where each $\overline{B}_i$ is a Boolean formula. Intuitively, the formula $\overline{\varphi}_g^s$ is a disjunction of persistence properties, each of which characterizes states from which the computation eventually reaches a cycle that constantly falsifies one of the system's liveness guarantees.

To understand Def. 2 it is best to contrast it with Def. 1. Note that the implication from environment assumptions to system guarantees in Def. 1 is replaced in Def. 2 with a conjunction where the guarantees on the right hand side appear in negated form. That is, Equ. (2) is the negation of Equ. (1).

B. Computing a Counter Strategy

A controller that implements a winning counter strategy for the environment is a controller in which in every step of the execution, the environment's action is such that for every system reaction, the resulting ongoing behavior falsifies the specification. Intuitively, such a controller can roughly be viewed as a 'tree' where all leaves represent (safety) violations of the system's behavior, and every level of the tree allows cycles that constantly violate one of the system's (liveness) guarantees and whenever the guarantee is satisfied, the execution is bound to continue to the next, descending level of the tree.

We compute a counter strategy by solving a Rabin game [29] where the environment can force the execution to satisfy Def. 2. That is, the environment can force the execution to satisfy all of its assumptions, while falsifying at least one of the system's guarantees. Our solution to this game is based on a symbolic fixpoint algorithm described in [17], [27].

Roughly, the algorithm starts from the set of states from which the system has no valid possible successors (all possible

successors lead to states that violate one of the safety guarantees). It then iterates 'backwards' by adding states from which the environment can either force the execution to previously found losing states, or force the execution to constantly violate one of the system's guarantees. Each set of states where the guarantee is constantly violated is computed using another nested fixpoint and saved in a vector of intermediate values.

The fixpoint is reached when no additional losing states can be found. If there exists an environment initial choice such that every system initial choice is in the computed set of losing states, then the specification is unrealizable. A controller that implements the winning counter strategy can be constructed from the intermediate values saved during the fixpoint computation, as each set of such values represent a set of states where one of the guarantees is constantly violated.

Note that the intermediate values of the fixpoint computation are the real reason to define and solve the dual game. If a simple Boolean answer is enough, then there is no need for anything other than the GR(1) formulation.

In this work we use the resulting counter strategy trees to construct *counter play-out*, an interactive debugging method for LSC specifications, described in the next section.

IV. Counter Play-Out

We now present the second contribution of our work, that is, the interactive debugging method for unrealizable specifications. Given a counter strategy, we define *counter play-out* to be an interactive LSC execution with reversed roles: a generated controller plays the role of the environment and lets the engineer play the role of the system. The generated controller uses the counter strategy to choose its (environment's) moves such that the engineer is forced to eventually fail in satisfying the requirements.

We define positions in a counter play-out execution. Recall that according to the semantics for superstep (see [25]), the environment can send only one message at a time, and can send a message only when the system is ready to receive one.

We partition the states of the reversed controller during counter play-out execution into three kinds of states. The controller is in a:

(i) *system position* – if the current state of the execution is a state in which either the system or the environment just performed an action, i.e., either the system did not complete its superstep and is not ready to receive an environment message yet, or the environment has just performed an action that should be answered with a system reaction. In a system position, the engineer is required to choose a message from the set of messages that are enabled in the cut.

(ii) *environment position* – if the current state of the execution is a state in which both the environment and the system did not perform any action, i.e., the system has completed its superstep and is ready to respond to a new environment action, and the environment did not perform its action yet. In an environment position, the controller needs to send an environment message that is predetermined and embedded within its strategy.

(iii) *violating position* – if the current state of the execution is a state in which the system just performed an action that violates at least one of the charts in the given LSC specification.

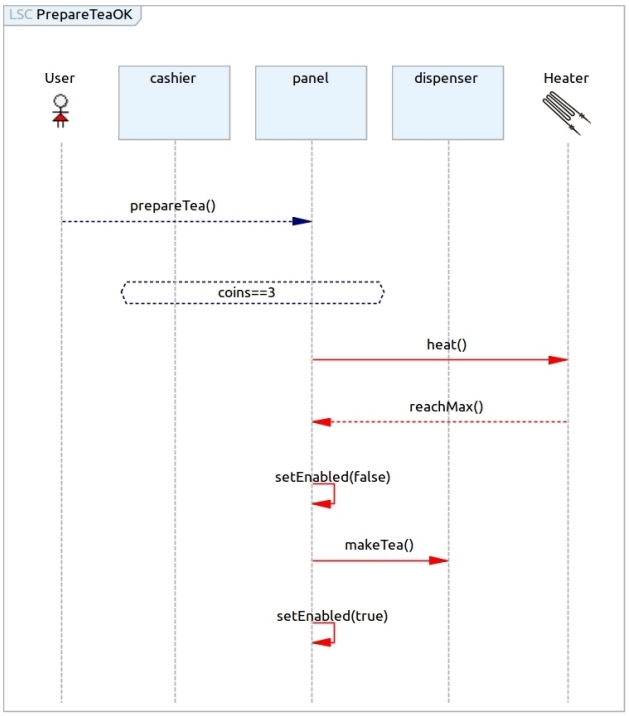

Fig. 3. The LSC `PrepareTeaOK` of the vending machine specification

Each state of the (reversed-roles) controller represents the global cut (the cuts of all LSCs), the state of each of the scenarios (active or inactive), and the value assigned to each of the properties. This allows us to easily interpret the state of the controller and present it, as it executes, in terms of the scenarios, i.e., using the abstractions known to the engineer.

A. Example Specification

We demonstrate counter play-out using a running example, a specification of a simple vending machine consisting of five LSCs, two of which were already mentioned in Sect. II. While the specification is rather small, we have carefully designed it to fit into a paper and demonstrate the need for counter play-out and the resulting debugging method. Additional examples are available with the PlayGo tool.

LSC `InsertCoins` (Fig. 1) specifies the basic scenario of coin insertion: whenever the user inserts a coin (the user sends an `insertCoin` message) to the panel, the panel should eventually send `incCoins` message to the cashier (this increases the cashier's coins property).

LSC `PrepareTeaFail` (Fig.2) describes the use case where the user asks the system to prepare tea but the number of coins is less than 3: whenever the user asks to prepare tea (sends the `prepareTea` message to the panel), if `coins <`

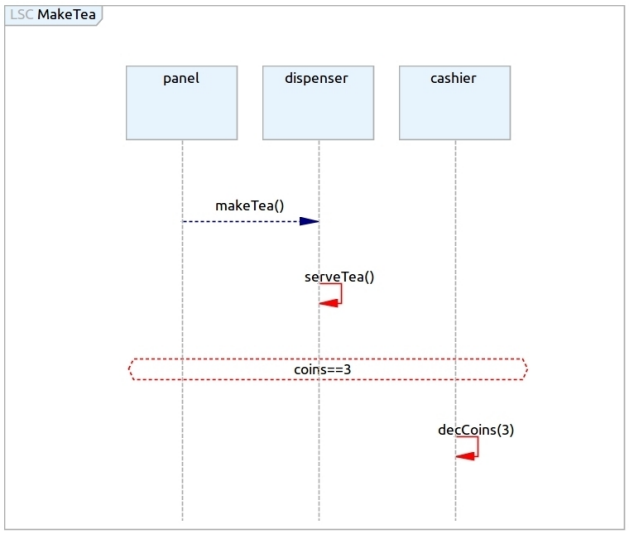

Fig. 4. The LSC `MakeTea` of the vending machine specification

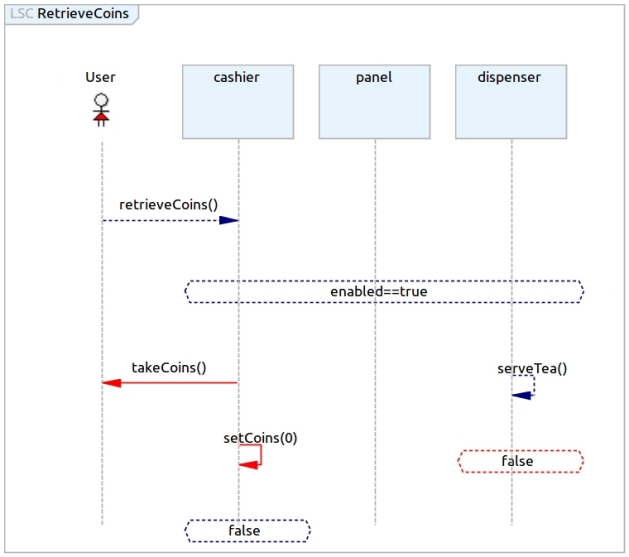

Fig. 5. The LSC `RetrieveCoins` of the vending machine specification

`3` then the system should eventually play a beep sound and show an error message (in no specific order). If it is not the case that `coins < 3`, then the chart closes gracefully (the condition is cold, not hot, thus there is no violation).

LSC `prepareTeaOK` (Fig. 3) describes the use case where the user asks the system to prepare tea and the number of coins is exactly 3. Whenever the user sends a `prepareTea` message, the cold condition `coins==3` is evaluated. If it is false, the scenario exits gracefully. Otherwise, the chart continues: the system's panel must eventually ask the heater (controlled by the environment) to heat the water, followed by an assumption that the heater will eventually send a `reachMax` message back to the panel. When a `reachMax` message is eventually received, the panel should eventually reset its `enabled` property, send a `makeTea` message to the dispenser, and set its `enabled` property back to `true`.

LSC `MakeTea` (Fig. 4) describes the behavior the system should follow whenever the panel sends the dispenser a `makeTea` message. In this case, the dispenser should send a self message to `serveTea`, an abstraction of a different scenario that entails the proper way to serve the tea. The chart continues to specify that the cashier's `coins` property must be exactly 3 (it is a hot condition), and be followed by a `decCoins(3)` message that will consume 3 coins (decrease the coins property by 3).

The last LSC, `RetrieveCoins` (Fig. 5), enables a cancellation functionality. If the user sends a `retrieveCoins` message to the panel, and if the panel is enabled (note, a cold condition), then the system must send the user a `takeCoins` message (give back the coins to the user) and follow with a `setCoins(0)` message that sets the cashier's `coins` property to 0. Furthermore, the chart also specifies that during the process of cancellation, the dispenser cannot send a `serveTea` message (sending this message would make the hot `false` condition on the right hand side of the chart enabled, and thus result in a hot violation of the requirements).

Finally, the specification includes initial values for two properties: `coins` is set to 0 and `enabled` is set to `true`.

B. First Counter Play-Out Session

An attempt to synthesize a controller for the vending machine specification reveals that it is unrealizable. Even though the specification is rather small, it is not easy for an engineer to identify the cause of unrealizability. What's wrong? To answer this, we compute a counter strategy, generate a reversed controller, and begin an interactive debugging session of counter play-out, as follows.

At first, the controller inserts a coin (playing the role of the environment), and the engineer (playing the role of the system) must follow LSC `InsertCoins` and send `incCoins` message to increase the cashier's `coins` property. The engineer could of course choose not to follow this obvious step to increase the cashier's `coins` property and do something else or just yield control back to the controller. In such a case, however, the controller would immediately insert another coin, leading to a hot violation of the LSC `InsertCoins`. Thus, in the remaining descriptions of counter play-out sessions below we leave out such cases of obvious, immediate failures.

Coin insertion repeats three times. After the third time, a `prepareTea` message is sent by the controller, and the cold condition in `PrepareTeaOK` is satisfied (while the cold condition in `PrepareTeaFail` message is falsified, and the chart closes gracefully). So, the engineer (again, playing the role of the system) must send a `heat` message to the heater and idle (wait) to expect the `reachMax` message that the controller (playing the environment role) must eventually send.

However, before sending `reachMax`, as it has to, eventually, our generated reversed controller invokes the cancellation functionality described in `RetrieveCoins`, by sending a `retrieveCoins` message. Since the panel is enabled, LSC `RetrieveCoins` forces the system's cashier (as played by the engineer) to send a `takeCoins` message and set the

cashier's coins property to 0, by sending a self `setCoins(0)` message. Only when the LSC `RetrieveCoins` closes gracefully, due to the `false` cold condition after `setCoins(0)`, the controller sends the `reachMax` message (it has to eventually do that in order to satisfy the environment assumption), and the engineer is forced to reset the panel's `enabled` property and send a `makeTea` message.

Finally, following LSC `MakeTea`, the engineer must now send a `serveTea` message that leads the system to a violation in which the hot condition in this LSC, which states that the coins property must be 3, is false.

As part of counter play-out, the interactive executions described here are accompanied by visualizations that show the progress of LSC cuts on the charts themselves as well as a tree showing an overview of the execution as it unfolds during the counter play-out session according to the choices made by the counter strategy and the engineer's actions. Information about charts' progress, values of properties, and violations is shown on top of the nodes of this tree (see Sect. V). Screen capture from this interactive debugging session appears in Fig. 6.

C. First Fix and Second Counter Play-Out Session

The above counter play-out session reveals that having the coins retrieval functionality enabled (available for the environment to invoke) while waiting for the `reachMax` message from the heater, may lead (an adverse user) to abuse the system and prevent it from satisfying its requirements.

This suggests to fix the problem by moving the `enabled` lock (implemented in `PrepareTeaOK`) to wrap the `heat` request and response. Thanks to the cold condition at the beginning of LSC `RetrieveCoins`, this fix would disable the coins retrieval functionality during heating.

Unfortunately, however, an attempt to synthesize a controller for the fixed vending machine specification fails again, and reveals that it is unrealizable. What's wrong now? Again, to answer this question, we compute a counter strategy, generate a reversed controller, and begin a second interactive debugging session of counter play-out, as follows.

To begin, the generated controller (playing the role of the environment) starts as in the previous debugging session: it inserts three coins and then calls the `prepareTea` message. As the cold condition in LSC `PrepareTeaOK` is satisfied, the engineer must set the panel's `enabled` property to false (following her fix in the previous debugging session) and send a `heat` message to the heater. After doing so it idles to wait for the expected `reachMax` message.

Now, rather than invoking the cancellation functionality (as in the previous counter play-out session), the controller (playing the role of the environment) sends another `insertCoin` message, which again forces the engineer (playing the role of the system) to follow the LSC `InsertCoins` and increase its cashier's `coins` property. Only then, the reversed controller 'chooses' to satisfy its assumption (as it has to, eventually), by sending `reachMax` message.

To continue following LSC `PrepareTeaOK`, the engineer, as she must, sends a `makeTea` message to the dispenser. This makes LSC `MakeTea` active and forces the engineer to follow with the `serveTea` message and inevitably reach a violation: the hot condition `coins==3` is evaluated to false.

Note that during the counter play-out sessions described above, at any stage of the execution, the reversed-roles synthesized controller, playing the role of the environment (user), could have asked the system to prepare tea (and activate scenarios `PrepareTeaFail` and `PrepareTeaOK`) or ask the system to retrieve coins (and activate the LSC `RetrieveCoins`) before inserting three coins. If the controller would have done that, the system (played by the engineer) should have responded with the expected reactions according to these LSCs. Yet, our generated reversed controller does not choose to do this, specifically because these are not necessary steps on the way to proving unrealizability.

D. Second Fix and Closure

At the end of the second counter play-out session described above, one can immediately see that the conditions in both `PrepareTeaOK` and `MakeTea` may be too restrictive. Thus, the engineer may attempt to fix this bug in the specification by specifying that the system should not be forced to hold exactly 3 coins but rather to hold at least 3 coins in the specific stages during the execution of these two scenarios. Thus, the engineer applies a second fix and changes these two conditions to read `coins >= 3`.

Indeed now, after the second fix, the specification is realizable and we are finally able to synthesize a system controller (rather than a reversed-roles controller). The system's strategy allows the user to insert coins, ask the system to prepare tea, and retrieve coins, all with no 'risk' of violation.

When at least 3 coins are inserted, and the user continues with a `prepareTea` message, the system sets the panel's `enabled` property to `false` before the panel sends a `heat` message to the heater. Thus the environment (user) cannot initiate the coins retrieval functionality of `RetrieveCoins` (if the user asks to retrieve coins at this state, nothing happens). When the `reachMax` message is eventually sent, the system sends a `makeTea` message followed by a `serveTea` message, and the three coins are consumed when a `decCoins(3)` message is sent. Note that after `makeTea` is called, LSC `PrepareTeaOK` and `MakeTea` do not explicitly define an order between the completion of `MakeTea` (serving the tea, decreasing the coins) and the completion of `PrepareTeaOK` (re-enabling the panel). That is, in this case, any order chosen by the strategy would work and make sure both charts are indeed completed without violation.

V. Implementation and Evaluation

Links to download PlayGo, related technical documentation, the example specifications mentioned in the paper, and a standalone implementation to run synthesis (and reproduce the tests mentioned below), are all available from [1].

We have implemented our ideas using JTLV APIs [28] and integrated them into PlayGo [14]. JTLV is a Java-based framework for the development of formal verification

algorithms, providing developer-friendly high-level APIs for symbolic BDD-based algorithms. PlayGo is an eclipse-based IDE built around the language of LSC and the play-in/play-out approach [15]. It includes a compiler that translates LSCs (given in a UML-compliant form, using a profile, see [13]) into AspectJ code (based on [22], [23]), and provides means for visualization and exploration of LSC executions.

We extended PlayGo's IDE to support a counter play-out mode where a reversed-roles controller is generated and executed so that the engineer can play against it.

A. Performance

In the vending machine specification, where the number of LSCs is 5, the state space is 2^{27}, the size of the controller (for realizable specifications) is about 8K states, and the size of the reversed-roles controller (for the unrealizable specifications) ranges from 95 to 190 states. Computing (counter) strategies and generating the controller took only up to 3 seconds (on a regular laptop, 2.8 GHz, using JavaBDD).

To further test and demonstrate our work, we have created 15 additional example specifications (unrealizable and realizable), where the number of LSCs ranges from 2 to 12, the state space ranges from 2^{17} to 2^{39}, and the controller or reversed controller size ranges from 38 to 28K states. Computing (counter) strategies and generating the controller took from 120 milliseconds up to about 40 seconds (on a regular laptop, 2.8 GHz, using JavaBDD). Note that our example specifications are much larger than ones presented in recent comparable works on synthesis [7], [32]. The scalability and applicability of our work to mid-size systems is due to the use of symbolic, BDD-based, algorithms, as shown in [2].

In our current implementation, the entire computation is done once before the execution of the interaction. If performance was our main focus, then a better approach was to solve the game symbolically but evaluate each concrete step only on demand. In all our examples mentioned above, solving the game ranges from 23 milliseconds to at most 25 seconds.

B. The Counter Play-Out View

Fig. 6 shows a screen capture from PlayGo, taken during the first counter play-out session of the vending machine specification. During execution, LSC cuts are visually shown as horizontal lines on top of the charts in the specification (Fig. 6, top). Property values are shown in PlayGo's system model tree (Fig. 6, left). Most importantly, a *counter play-out view* shows a graph representation of (part of the) counter play-out strategy as it unfolds in the session according to the choices made by the controller and the engineer (Fig. 6, bottom).

The graph visualization distinguishes three types of positions during counter play-out execution: system, environment, and violating (defined earlier in Sect. IV). Each system position (e.g., state 20) is represented by a node that is annotated with a question mark icon, the state ID, the description of the active cuts, and the values of the properties. Each system node has its own popup menu, which allows the engineer to select and execute each of the enabled messages in the cut.

Each environment position (e.g., state 16) is represented by a node annotated with a user icon (and like system nodes contains the state ID, the description of the active cuts, and the values of the properties). During execution, states that represent environment positions are not controlled by the engineer and thus environment nodes do not provide a popup menu for interactive selection of the next message to execute. Instead, an environment message (as selected by the reversed controller) is immediately executed and the tree unfolds.

Finally, each violating state (e.g., state 19) is represented by a node that is annotated with an X error icon and displays the name of the violated chart. This provides the engineer with exact information about the reason for unrealizability.

The main window pane shows the LSC specification and presents the cut after performing the last message in the execution using a horizontal red line (Fig. 6 presents the cut corresponding to state 22).

VI. Related Work

Handling unrealizable scenario-based specifications Controller synthesis from LSC specifications has been studied before, using different approaches, by Larsen et al. [20], by Harel and Kugler [11], by Kugler et al. [19], by Harel and Segall [16], by Bontemps et al. [3], [4], and by Greenyer [8]. Except the last two, none discussed unrealizability. While Bontemps et al. [3], [4] suggested to address the unrealizable case using a counter strategy, their solution to the synthesis problem uses an inefficient double exponential algorithm, unlike our use of the efficient algorithms from [2], [27]. While Greenyer [8] describes the ability to synthesize a counter strategy, he suggests the extension of his tool to support the simulation of the environment as future work. Thus, both works do not use the counter strategy to construct a controller to guide an actual reversed-roles execution that explains the root causes of unrealizability, as done in our work.

Other authors have suggested different scenario-based specification languages, almost all of which, like LSC, are variants of Message Sequence Charts (MSCs). Some have been used for various kinds of synthesis. We discuss them in chronological order, focusing on the context of unrealizability.

Krueger et al. [18] consider a translation of MSCs into a state-based model, in particular, statecharts. A case where the specification cannot be realized is not discussed.

Uchitel et al. [35] use an exact, existential interpretation, similar to [18]. They discuss how the generated labeled transition system can be checked for dead-locks etc. but a case where the specification cannot be realized is not discussed.

Whittle et al. [36] generate statecharts from scenarios. The scenarios are labeled with additional state information (in OCL) that is used by the synthesis algorithm, so some explicit state conflicts, which prevent synthesis, may be detected during analysis. The user is responsible for fixing these conflicts.

Sibay et al. [32] present a conditional, existential branching-time semantics for LSC with an algorithm to synthesize a modal transition system (MTS), characterizing all possible conforming implementations. A case where the specification

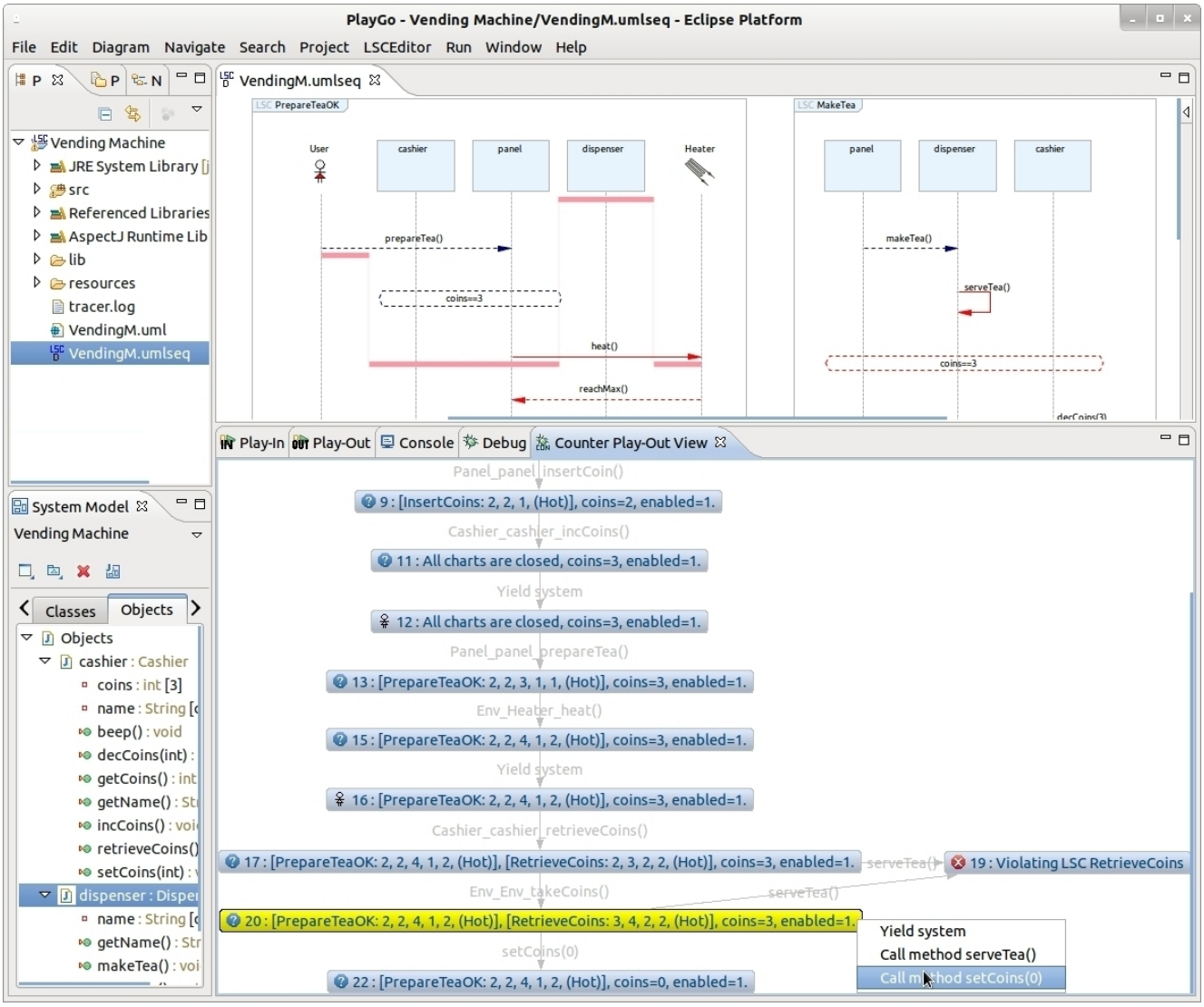

Fig. 6. A screen capture from PlayGo taken during the first counter play-out session of the vending machine specification. The LSCs appear at the upper part, with a cut (as horizontal line) showing their progress and current state. Note the three types of nodes in the graph in the lower part (system, marked by a question mark, environment, marked by a user icon, and violation, marked by an X sign). The edges are labeled with the messages that have been sent: sender name, receiver name, and the message name. The nodes are annotated with explanatory information, which scenarios are active, what are the values of properties, which scenarios are violated etc., that is, explaining the execution using the abstractions known to the engineer. See Sect. V-B.

cannot be realized is not discussed. In a related work, Uchitel et al. [34] present synthesis of an MTS from a combination of existential scenarios and safety properties. The work mentions that two MTSs may be inconsistent (have no common refinement), however, this case is not addressed.

To conclude, almost no work on synthesis from scenarios discusses unrealizability. This may be in part because the expressive power of some of these scenario languages is limited to existential 'positive examples', where conflicts cannot happen. Also, some variants do not explicitly distinguish between the system and the environment. The need to address unrealizability is the 'price' one has to pay for choosing to use an expressive scenario-based language such as LSC.

Dealing with inconsistent specifications Outside the scenarios domain, two related works that deal with inconsistency in declarative specifications should be mentioned. Sassolas et al. [30] use a pseudo-merge operation to explore and explain inconsistencies between two MTSs. Shlyakhter et al. [31] show how to take advantage of SAT solvers' unsatisfiable core to debug Alloy modules. These two works are close to ours in terms of the motivation to identify and explain inconsistencies (in our case, unrealizability) in specifications. However, the technical setups and solutions are entirely different.

Counter strategies for LTL specifications Counter strategies for synthesis from LTL in general and from its GR(1) fragment in particular have been studied in [17], [27]. In [26] we used these algorithms in the context of AspectLTL [24]. In the present paper we reduce the LSC specification into a (dual) GR(1) form, and compute counter strategies by using the algorithms described in these papers.

VII. Conclusion and Future Work

We have presented counter play-out, an interactive debugging method for unrealizable scenario-based specifications. Counter play-out allows an engineer to play against a synthesized adverse environment controller that forces her to fail in satisfying the system's requirements and leads the execution to the specific scenarios whose violation is unavoidable, i.e., to the root causes of unrealizability. It is based on a counter strategy computed by solving a Rabin game, and is implemented and demonstrated in this paper using a running example. Scalability and applicability to mid-size systems is due to the use of symbolic, BDD-based, algorithms, as shown in [2]. The work advances the state-of-the-art in the scenario-based approach to specification and simulation.

One future work direction deals with the computation of *unrealizable cores*, i.e., minimal unrealizable subsets of the specification. For some unrealizable specifications, especially ones consisting of many scenarios, it may be the case that there are strict subsets of the specification that are unrealizable. For example, in our unrealizable vending machine specification, before fixing, the LSC `PrepareTeaFail` is not part of an unrealizable core; the specification without it is unrealizable too. Identifying unrealizable cores can be useful as it may enable the generation of smaller reversed controllers. Of course we could find unrealizable cores by checking all subsets of the specification. However this would be inefficient. Some recent works have considered the computation of unrealizable core in the context of LTL (GR(1)) synthesis (see, e.g., [5]). In the case of LSC specifications, however, we would like the unrealizable core to be computed and presented using the abstractions defined by the scenarios, rather than at the granularity of the LTL formulae they induce. Moreover, LSC specifications are not monotonic: adding an LSC to an unrealizable specification may render it realizable. Thus, we leave the definition and efficient computation of unrealizable cores for LSC as challenges for future work. Another future work direction relates to the usability of counter play-out in the presence of large reversed-roles controllers where the interaction required to reach violation is long. In these cases we would like to use a summarized representation of the counter strategy and a time-travel feature, to jump between sets of states that are close to violating ones (such sets may be computed symbolically). We leave this interesting topic for future work.

Acknowledgements

Part of this research was funded by an Advanced Research Grant awarded to David Harel of the Weizmann Institute from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

References

[1] "Supporting materials for counter play-out paper," http://smlab.cs.tau.ac.il/cpo/.

[2] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *JCSS*, vol. 78, no. 3, pp. 911 – 938, 2012.

[3] Y. Bontemps, P. Heymans, and P.-Y. Schobbens, "From Live Sequence Charts to State Machines and Back: A Guided Tour," *IEEE Trans. Software Eng.*, vol. 31, no. 12, pp. 999–1014, 2005.

[4] Y. Bontemps, P.-Y. Schobbens, and C. Löding, "Synthesis of open reactive systems from scenario-based specifications," *Fundam. Inform.*, vol. 62, no. 2, pp. 139–169, 2004.

[5] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev, "Diagnostic information for realizability," in *VMCAI*, 2008, pp. 52–67.

[6] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *FMSD*, vol. 19, no. 1, pp. 45–80, 2001.

[7] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models," in *SIGSOFT FSE*, 2010.

[8] J. Greenyer, "Scenario-based design of mechatronic systems," Ph.D. dissertation, University of Paderborn, Dept. of Computer Science, 2011.

[9] D. Harel, "From play-in scenarios to code: An achievable dream," *IEEE Computer*, vol. 34, no. 1, pp. 53–60, 2001.

[10] D. Harel, A. Kantor, and S. Maoz, "On the power of play-out for scenario-based programs," in *Concurrency, Compositionality, and Correctness*, ser. LNCS, vol. 5930. Springer, 2010, pp. 207–220.

[11] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications," *Found. Comp. Sci.*, vol. 13, no. 1, pp. 5–51, 2002.

[12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart play-out of behavioral requirements," in *FMCAD*, 2002.

[13] D. Harel and S. Maoz, "Assert and negate revisited: Modal semantics for UML sequence diagrams," *Software and Systems Modeling*, vol. 7, no. 2, pp. 237–252, 2008.

[14] D. Harel, S. Maoz, S. Szekely, and D. Barkan, "PlayGo: towards a comprehensive tool for scenario based programming," in *ASE*, 2010.

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

[16] D. Harel and I. Segall, "Synthesis from scenario-based specifications," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 970–980, 2012.

[17] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications using simple counterstrategies," in *FMCAD*. IEEE, 2009.

[18] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," in *DIPES*, 1998, pp. 61–72.

[19] H. Kugler, C. Plock, and A. Pnueli, "Controller synthesis from LSC requirements," in *FASE*, 2009.

[20] K. G. Larsen, S. Li, B. Nielsen, and S. Pusinskas, "Scenario-based analysis and synthesis of real-time systems using UPPAAL," in *DATE*. IEEE, 2010, pp. 447–452.

[21] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and control in scenario-based requirements analysis," in *ICSE*, 2005, pp. 382–391.

[22] S. Maoz and D. Harel, "From multi-modal scenarios to code: compiling LSCs into AspectJ," in *SIGSOFT FSE*, 2006, pp. 219–230.

[23] S. Maoz, D. Harel, and A. Kleinbort, "A compiler for multimodal scenarios: Transforming LSCs into AspectJ," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, p. 18, 2011.

[24] S. Maoz and Y. Sa'ar, "AspectLTL: an aspect language for LTL specifications," in *AOSD*, 2011, pp. 19–30.

[25] ——, "Assume-guarantee scenarios: Semantics and synthesis," in *MoDELS*, ser. LNCS, vol. 7590. Springer, 2012, pp. 335–351.

[26] ——, "Two-way traceability and conflict debugging for AspectLTL programs," in *AOSD*, 2012, pp. 35–46.

[27] N. Piterman and A. Pnueli, "Faster solutions of Rabin and Streett games," in *LICS*. IEEE Computer Society, 2006, pp. 275–284.

[28] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "JTLV: A Framework for Developing Verification Algorithms," in *CAV*, 2010.

[29] M. O. Rabin, "Decidability of second-order theories and automata on infinite trees," *Trans. Amer. Math. Soc.*, vol. 141, pp. 1–35, 1969.

[30] M. Sassolas, M. Chechik, and S. Uchitel, "Exploring inconsistencies between modal transition systems," *SoSyM*, vol. 10, no. 1, 2011.

[31] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri, "Debugging overconstrained declarative models using unsatisfiable cores," in *ASE*, 2003, pp. 94–105.

[32] G. Sibay, S. Uchitel, and V. A. Braberman, "Existential live sequence charts revisited," in *ICSE*, 2008, pp. 41–50.

[33] R. S. Streett, "Propositional dynamic logic of looping and converse is elementarily decidable," *I&C*, vol. 54, no. 1/2, pp. 121–141, 1982.

[34] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 384–406, 2009.

[35] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Tran. Soft. Eng.*, vol. 29, no. 2, pp. 99–115, 2003.

[36] J. Whittle, J. Saboo, and R. Kwan, "From scenarios to code: An air traffic control case study," in *ICSE*, 2003, pp. 490–497.