Assume-Guarantee Scenarios: Semantics and Synthesis*

Shahar Maoz[1] and Yaniv Sa'ar[2]

[1] School of Computer Science, Tel Aviv University, Israel
[2] Dept. of Computer Science, The Weizmann Institute of Science, Israel

Abstract. The behavior of open reactive systems is best described in an assume-guarantee style specification: a system guarantees certain prescribed behavior provided that its environment follows certain given assumptions. Scenario-based modeling languages, such as variants of message sequence charts, have been used to specify reactive systems behavior in a visual, modular, intuitive way. However, none have yet provided full support for assume-guarantee style specifications.

In this paper we present *assume-guarantee scenarios*, which extend live sequence charts (LSC) — a visual, expressive, scenario-based language — syntax and semantics, with an explicit distinction between system and environment entities and with support not only for safety and liveness system guarantees but also for safety and liveness environment assumptions. Moreover, the semantics is defined using a reduction to GR(1), a fragment of LTL that enables game-based, symbolic, efficient synthesis of a correct-by-construction controller.

1 Introduction

It has long been recognized that the behavior of open reactive systems [11], discrete event systems that interact with their environment over time, and of other systems, is best specified using an assume-guarantee style specification: a system guarantees certain prescribed behavior provided that its environment follows certain given assumptions (see, e.g., [14, 20]). Environment assumptions may be related to the laws of physics (when interacting with the physical world) or to knowledge about the behavior of external systems (when interacting with other systems). They are crucial in many application domains, because some system requirements may only be realizable under assumptions about behaviors the environment will never or will always eventually exhibit. Scenario-based languages, however, which have been used to specify reactive systems behavior in a visual, modular, intuitive way, have not yet provided full support for the assume-guarantee paradigm.

* This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science. In addition, 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).

R.B. France et al. (Eds.): MODELS 2012, LNCS 7590, pp. 335–351, 2012.
© Springer-Verlag Berlin Heidelberg 2012

One such language is live sequence charts (LSC) [5, 8], a visual language for scenario-based modeling, which extends classical sequence diagrams with a distinction between mandatory-universal behavior (hot elements) and provisional-existential behavior (cold elements). While LSC allows one to specify possible and mandatory scenarios that a system should follow, and negative scenarios that a system should never allow, its current syntax and semantics do not allow one to condition the realization of these system guarantees on the fulfilment of certain behaviors of the environment. In other words, it does not support environment assumptions. This limits the expressive power of the language and its applicability to specifying real-world systems.

In this paper we present *assume-guarantee scenarios*, an extension of the LSC language with support for environment assumptions. We define the syntax and semantics of the extended language, allowing one to express safety assumptions, that is, what the environment is assumed never to do, and liveness assumptions, what the environment is assumed to always eventually do. The extension does not add external constructs to the language. Rather, it is defined by embedding assumptions implicitly in the LSCs, in keeping with the scenario-based nature of the language, just like safety and liveness system guarantees are specified in LSCs implicitly in the scenarios, using the distinction between hot and cold elements.

Moreover, we formulate the semantics of the extended language in GR(1), a fragment of linear temporal logic (LTL). The GR(1) formulation allows us to build on the game-based, symbolic, efficient synthesis algorithm of [26] and generate a correct-by-construction, executable controller. Assuming the environment adheres to the assumptions, the generated controller behavior meets the guarantees.

We have implemented our ideas using JTLV APIs [28] and integrated them into PlayGo [9]. We extended PlayGo's visual editor to support the extended language syntax, and implemented both the reduction to the GR(1) fragment and the solution of the GR(1) synthesis. The resulting controller is realized in a standalone, generated executable Java application.

We discuss related work below. Sect. 2 recalls the LSC language and presents a semi-formal overview of the assume-guarantee extension using examples. Sect. 3 recalls the GR(1) fragment of LTL, which we use as the target for the definition of the semantics of the extended language. Sect. 4 presents our main contribution: the semantics of assume-guarantee scenarios, formulated in GR(1) form. Sect. 5 presents a running example, and the second contribution of our work: synthesis of assume-guarantee scenarios. Sect. 6 describes our implementation and Sect. 7 concludes with a discussion and future work directions.

1.1 Related Work

Several scenario-based specification languages have been suggested in the literature, each with a different semantics. We discuss some of these here, focusing on the distinction between system and environment and on the ability to specify assumptions and guarantees.

Haugen et al. [13] present STAIRS, a requirements specification methodology based on UML2.0, where the semantics of interactions is given using

interaction obligations. STAIRS does not distinguish between system and environment controlled objects. Thus, one may interpret its semantics to include only system guarantees and no environment assumptions.

Knapp and Wuttke [15] use UML2.0 interactions as a specification in a model-checking setup. They interpret a sequence diagram as an observer of the message exchanges and state changes in a system. Again, no distinction is made between system and environment entities / controlled messages and thus the work can be viewed as checking system guarantees, with no environment assumptions.

Whittle and Schumann [30] generate a statechart from a set of scenarios annotated with OCL constraints. The construction distinguishes messages sent by the user from messages sent by the system, and thus may be viewed as relying on implicit assumptions. However, these are only safety assumptions.

Krueger et al. [17] consider a translation of an MSC specification into a statechart. In the process, scenarios are projected onto each of the components participating in them. This may be viewed as considering each component alone to be a system and the other components as its environment. However, an explicit distinction between assumptions and guarantees is not discussed.

Additional scenario-based specification languages, such as VTS [1] and PST [2] do not explicitly distinguish between system controlled and environment controlled events, and thus do not support assumptions.

Greenyer [6] presents a translation of timed and untimed modal sequence diagrams [8] specifications into UPPAAL-TIGA [3], for the purpose of synthesis. Environment assumptions are supported through the use of *assumption MSDs*, scenarios explicitly tagged as specifying assumptions. In contrast, we chose to integrate assumptions into the same scenarios, so that a single scenario can specify a combination of system guarantees and environment assumptions. We believe this provides more flexibility. To the best of our knowledge, [6] is the only previous work that supports liveness and safety assumptions in the context of scenario-based specifications and synthesis.

Finally, Kugler et al. and Harel and Segall [12,18] present controller synthesis from LSC. These works, however, consider 'classic' LSCs, where the semantics ignores the temperature of environment controlled events, and thus do not support environment assumptions.

2 An Overview of Assume-Guarantee Scenarios

We start off with background about classic LSC and then demonstrate the contribution of the assume-guarantee extension through a presentation of a small example, a scenario-based specification of a vending machine. Part of the specification is presented here. Additional LSCs are presented in Sect. 5. The overview is semi-formal. Required formal definitions are given in the following sections.

2.1 Background on LSC

Live sequence charts (LSC) [5,8] 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. We give

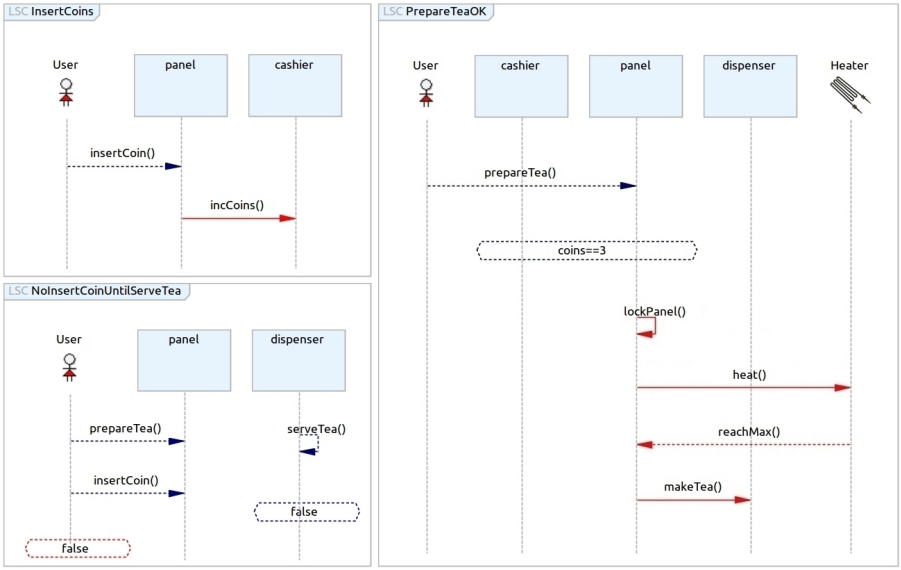

Fig. 1. Three scenarios from the vending machine specification. Note the distinction between system entities (panel, cashier, dispenser) and environment entities (user, heater). Also note the distinction between hot and cold elements, the hot environment controlled message `reachMax` in LSC `PrepareTeaOK` and the hot `false` condition on the user lifeline in LSC `NoInsertCoinUntilServeTea` (see Sect. 2).

a short, simplified overview of the language, with an emphasis on the parts most relevant to our present work. Detailed descriptions are available in [5, 8].

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* if it is sent from a lifeline controlled by the system, and is an *environment message* otherwise (if it is sent from a lifeline controlled by the environment). The LSC defines a partial order on its messages, induced by the vertical ordering of messages sent and received along the lifelines.

As an example, Fig. 1 (top left) shows the LSC `InsertCoins`. This LSC has one environment lifeline (controlled by the user) and two system lifelines, representing the system's `panel` and `cashier`. The first message `insertCoin` is an environment message and the second message `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. The *minimal cut* represents the state where the chart is closed. A cut induces a set of enabled and violating messages and conditions: a message is *enabled* in a cut of a chart 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, this is a violation of the requirements and thus should have never occurred. In the LSC `InsertCoins` the first message is cold and the second message 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 the condition was cold, the chart closes gracefully (the cut is set to be the minimal cut); if the condition was hot, this is a violation of the requirements and thus should have never occurred.

System messages can be marked as either *execution* (solid line) or *monitoring* (dashed line). All environment messages are marked as monitoring. A chart is *active* if its current cut has an enabled (system) message for execution. In the LSC `InsertCoins` the first message is marked for monitoring while the second is marked for execution. The cut after the first message is sent is active.

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 it. 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 it.

For example, the semantics of the chart `InsertCoins` 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). Implicitly, this also means that after `insertCoin` is sent, the system message `incCoins` must come before another `insertCoin` message is sent by the environment.

2.2 LSCs with Environment Assumptions

`InsertCoin` is a classic LSC: it specifies a system guarantee. What is impossible to specify in classic LSC are assumptions on the behavior of the environment. This is possible in the extended language. We give two examples below.

LSC `PrepareTeaOK` (Fig. 1 (top right)) describes the use case where the user asks the system to prepare tea and the number of coins inserted 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. If it is true, the chart continues: the system's panel must eventually send its own `lockPanel` message and then ask the heater (controlled by the environment) to heat the water. This is followed by an assumption that the heater will eventually send a `reachMax`

message back to the system's panel (note that `reachMax` is a hot message controlled by the environment). When a `reachMax` message is eventually received, the panel should eventually send a `makeTea` message to the dispenser.

LSC `NoInsertCoinUntilServeTea` (Fig. 1 (bottom left)) involves the user, the panel, and the dispenser. It specifies that whenever the user sends a `prepareTea` message to the panel, the user must not send an `insertCoin` message unless the dispenser has sent its own `serveTea` message. Note that if the user sends `insertCoin` after she sends `prepareTea` and before the dispenser has sent the `serveTea` message, then the LSC would reach a hot `false` condition on the user's lifeline, that is, this would constitute a violation of the assumption (when the `serveTea` message is sent, the chart closes gracefully because it reaches a cold `false` condition).

These two LSCs demonstrate the power of assume-guarantee scenarios in combining system guarantees and environment assumptions within a scenario-based specification setup.

3 Generalized Reactive Specification

We recall the definition of the class of generalized reactive of rank 1 specifications (GR(1)) [4, 26], a fragment of LTL, which we use as the target for the definition of the semantics of assume-guarantee scenarios.

Linear temporal logic (LTL) [21,27] extends propositional logic with operators that describe variables valuations along infinite computation paths. Given a finite set of atomic propositions P, LTL formulae are constructed as $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi$, where $\bigcirc\varphi$ is the *next* temporal operator, roughly meaning that φ is true in the next step in the computation, $\varphi \mathcal{U} \psi$ is the *until* operator, roughly meaning that in any sequence of future steps φ is true *until* ψ is true. We use the usual abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. Additional future temporal operators, $\diamondsuit$ (*eventually*) and $\square$ (*globally*), are defined as abbreviations to $\mathsf{true}\mathcal{U}\varphi$ and $\neg\diamondsuit\neg\varphi$, respectively.

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

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

where:

(i) φ_a^e and φ_a^s are Boolean formulae which characterize the initial values that are assumed by 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 formula that is a 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* $\bigwedge_{i \in I} \Box\Diamond B_i$ *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.*

Open systems are systems that interact with their environment, that is, receive some inputs and react to them. For such systems specifications are usually partitioned into assumptions and guarantees. The intended meaning is that if all assumptions hold then all guarantees should hold as well. That is, if the environment behaves as expected then the system will behave as expected as well. In the next section we present the semantics for assume-guarantee scenarios.

4 Assume-Guarantee Scenarios: Semantics

We are now ready to present our main contribution, i.e., an LTL-based semantics for a specification consisting of a set of assume-guarantee scenarios. We form the semantics within the GR(1) fragment.

4.1 Formal Settings

To model LSC behavior, we present formal settings similar to those presented in [7, 12, 19]. Given a set of LSCs, $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, we define $\mathcal{M}^s(\mathcal{L})$ (resp. $\mathcal{M}^e(\mathcal{L})$) to be the set of messages that the system (resp. environment) can send in the charts. The sets of system and environment messages are disjoint, i.e., $\mathcal{M}^s(\mathcal{L}) \cap \mathcal{M}^e(\mathcal{L}) = \emptyset$. We define a formal model using the following variables:

- m_e is an input *environment message variable* over the domain of all possible messages that the environment can send in $\mathcal{L}$, and an additional no-op value for doing nothing. Intuitively, to every message $m \in \mathcal{M}^e(\mathcal{L}) \cup \{$ *"no-op"*$\}$ sent by the environment, the synthesized strategy will "know" how to react.
- m_s is an output *system message variable* over the domain of all possible messages that the system can send in $\mathcal{L}$, and an additional no-op value for doing nothing. Intuitively, in every state, the synthesized strategy entails which system message $m \in \mathcal{M}^s(\mathcal{L}) \cup \{$ *"no-op"*$\}$ should be sent.
- $l_1, \ldots, l_n$ is a set of output *cut variables*. Each l_i encodes a *cut-automaton* for $\mathcal{L}_i$. The domain of l_i, which we denote by $Dom(l_i)$, consists of all possible cuts in $\mathcal{L}_i$, including a *minimal cut* of the chart (denoted by the value MIN). We add two unique sink values VIO^s and VIO^e, to represent hot-violation of the system guarantees and environment assumptions, respectively. The variable l_i maintains where the execution is at the moment along each lifeline in $\mathcal{L}_i$.[1]

Each $\mathcal{L}_i$ semantics is captured by the transitions of the cut-automaton that update its corresponding l_i variable according to the taken steps. The minimal cut value MIN indicates that the chart is currently closed. If a message is not a part of the chart then the cut-automaton can perform an idle step. The value

[1] Note that there are other ways to encode a cut (e.g., a variable per lifeline per LSC). Our formal settings is independent of any one specific encoding.

VIO^s captures the fact that the system performed a hot violation, and the chart can no longer be satisfied. On the other hand, the value VIO^e indicates that the environment did not fulfil its assumptions, and the chart is vacantly satisfied. We denote by $\rho_{\mathcal{L}_i}$ the transition of the cut-automaton for $\mathcal{L}_i$.

4.2 Superstep Requirements

Formally, a superstep is a series of messages sent by the system, encapsulated between two messages sent by the environment. When assumptions are not included in the semantics, as in the closed LSCs synthesis handled in [12, 18], an artificial technical step is needed in order to enforce the superstep semantics. This exposes the internals of GR(1) and requires to deal with the mechanics of the game structure. Thus, it ties the solution to the GR(1) synthesis algorithm.

When assumptions are allowed, as in our open scenarios, a more natural and elegant way to describe the superstep semantics is available. Rather than working with the internals of the GR(1) game structure, we define the superstep semantics using two guarantees and two assumptions: G.1, G.2, A.1, and A.2 (see below). Thus, our approach defines a standalone LTL semantics that is independent of the mechanics of the synthesis algorithm (it could be solved with any LTL synthesis solution given that it is expressive enough to cover our specification).

First, we require the system to perform only a finite number of messages and give the environment a fair chance to communicate its messages.

Guarantee 1 (superstep: system fair turn): *The system always stops sending messages eventually.*

$$\square\diamondsuit\,(m_s = \textit{no-op}) \tag{G.1}$$

We also require the system to perform a message only if the environment is not sending a message.

Guarantee 2 (superstep: system safe turn): *If the environment sent a message (i.e. m_e is different from no-op) then the system cannot send a message.*

$$\square\bigcirc\,(m_e \neq \textit{no-op} \rightarrow m_s = \textit{no-op}) \tag{G.2}$$

Next, we require the environment to send one message at a time, allowing the system a fair chance to react to each message sent by the environment.

Assumption 1 (superstep: alternating turn): *If the environment sent a message in the last step (i.e. m_e is different from no-op) then the environment cannot send a message in the next step.*

$$\square\left(m_e \neq \textit{no-op} \rightarrow \bigcirc\,(m_e = \textit{no-op})\right) \tag{A.1}$$

Finally, we require the environment to send a message only when the system is ready to receive one, allowing the system a fair chance to finish its (guaranteed to be) finite number of steps.

Assumption 2 (superstep: environment fair turn): *If the system sent a message in the last step (i.e. m_s is different from no-op) then the environment cannot send a message in the next step.*

$$\square\left(m_s \neq \textit{no-op} \rightarrow \bigcirc\,(m_e = \textit{no-op})\right) \tag{A.2}$$

Note that the superstep requirements are fixed; they are not part of the application-specific semantics of the LSC specification. That is, the superstep requirements model our settings, whereas the additional requirements for the system (Subsect. 4.3) and for the environment (Subsect. 4.4, described below) model the application-specific semantics of the given LSC specification.

4.3 System Requirements

Given a set of LSCs $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, the application-specific system's semantics is defined using three guarantees: G.3, G.4, and G.5 (see below). To identify stable states in $\mathcal{L}_i$, we define $Act_i \subseteq Dom(l_i)$ to be the subset of active cuts from the domain of all cuts in the cut-automaton, i.e., cuts that contain an executable message that the system should perform.

First, we require the system to guarantee that each chart starts from its minimal cut.

Guarantee 3 (system: initial state): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$*, the system starts from a state in which the cut variable* l_i *is set to the minimal cut.*

$$\bigwedge_{i=1}^{n} (l_i = \text{MIN}) \tag{G.3}$$

Second, we require the system to guarantee that each chart follows its transitional semantics as discussed in Subsect. 4.1.

Guarantee 4 (system: transition): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$*, the system continuously preserves the transitions of the cut-automaton of* $\mathcal{L}_i$*.*

$$\bigwedge_{i=1}^{n} \Box \rho_{\mathcal{L}_i} \tag{G.4}$$

Finally, we require the system to guarantee to infinitely often visit a stable state, i.e., that infinitely often all charts visit inactive cuts in which there are no executable messages to be performed by the system.

Guarantee 5 (system: stable state): *The system always eventually reaches a state where every* $\mathcal{L}_i \in \mathcal{L}$ *is not active.*

$$\Box\Diamond \bigwedge_{i=1}^{n} (l_i \notin Act_i) \tag{G.5}$$

4.4 Environment Requirements

Given a set of LSCs $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, the application-specific environment's semantics is given using three assumptions. Assumptions A.3, A.4 characterize liveness requirements, and A.5 characterizes safety requirements (see below). To identify states in which the system is waiting for messages from the environment in $\mathcal{L}_i$, we define $Exp_i \subseteq Dom(l_i)$ to be the subset of expecting cuts from the domain of all cuts in the cut-automaton, i.e., cuts that contain executable message to perform by the environment.

Furthermore, given a cut $c \in Dom(l_i)$ we define $\mathcal{E}^e(c)$ to be the set of hot environment messages enabled in cut c (i.e., if $c \in Dom(l_i) \setminus Exp_i$, then $\mathcal{E}^e(c) = \emptyset$). Intuitively, in cut c the system assumes that the environment messages $\mathcal{E}^e(c)$ are bound to happen eventually. On the other hand, the semantics of cold environment messages do not require any assumption, and are treated just like cold

system messages (that is, a violation of a cold environment cut closes the chart gracefully and is not considered a violation of the requirements).

First, we require the environment to comply with a restricting (safety) property stating that if the system is in an expecting cut, then the next message sent by the environment is either *no-op* or one of the messages from the set of enabled hot environment messages. That is, the environment must focus on the hot messages at hand.

Assumption 3 (environment: active environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$ *and every expecting cut* $c \in Exp_i$*, if in the last step the system was in cut* c*, then in the next step the environment sends either no-op message, or a message from the set of hot enabled environment messages.*

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \square \left(l_i = c \rightarrow \bigcirc \left(m_e \in \{\mathcal{E}^e(c) \cup \textit{no-op}\} \right) \right) \tag{A.3}$$

Note that the requirement needs a rather loose restriction on the next step message since there could be cases where there are more than one possible hot environment message that is enabled. In such cases we would like to consider all possible combinations in which the environment meets its assumptions.

On the other hand, we require the environment to comply with the liveness property that states that when the system is in an expecting cut, then each enabled hot environment message must eventually be sent.

Assumption 4 (environment: fair environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$*, every expecting cut* $c \in Exp_i$*, and every hot enabled environment message* $m \in \mathcal{E}^e(c)$*, the environment always eventually either sends message* m*, or the system is not in cut* c*.*

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \bigwedge_{m \in \mathcal{E}^e(c)} \square \Diamond \left(l_i = c \rightarrow (m_e = m) \right) \tag{A.4}$$

The transition system semantics makes sure that if two hot environment messages are enabled in an expecting cut, and the first is being sent, then the following cut is also an expecting cut, which still awaits for the second hot environment message to be sent. Furthermore, unless the chart is closed, the execution cannot return to the previous expecting cut, thus the second hot environment message is bound to eventually be sent.

Note that from the system's perspective, the expecting cut is cold, that is, the system is allowed to violate it. However, as long as the system does not violate the expecting cut, the left side of the implication in both assumptions A.3 and A.4 hold, and the environment must follow in a way that would satisfy the right sides of these implications.

Finally, we would like to support explicit environment safety assumptions. The transition semantics makes sure that whenever a cut reaches a hot environment violation caused by an environment condition, l_i is set to the sink value VIO^e. Even though variable l_i is a system output (that the environment cannot control directly), the guarantee of the transition semantics to indicate a hot environment violation, enables the environment to reason in its strategy all possible future violations of its assumptions.

Formally, we require the environment to avoid letting the system reach (in the future) the sink value that indicates a hot environment violation.

Assumption 5 (environment: safe environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$, *the environment is never allowed to reach a hot environment violation.*

$$\bigwedge_{i=1}^{n} \Box \left(l_i \neq \text{VIO}^e\right) \tag{A.5}$$

4.5 Summary

The combination of all the above LTL assumptions and guarantees (A.1–5 and G.1–5), consists of a semantics for an LSC specification. We formalize it in a GR(1) form, as shown in Equ. (2).

environment	$A.3$	$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \Box \left(l_i = c \rightarrow \bigcirc \left(m_e \in \{\mathcal{E}^e(c) \cup \textit{no-op}\}\right)\right)$	$\bigwedge$
	$A.4$	$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \bigwedge_{m \in \mathcal{E}^e(c)} \Box\Diamond \left(l_i = c \rightarrow (m_e = m)\right)$	$\bigwedge$
	$A.5$	$\bigwedge_{i=1}^{n} \Box \left(l_i \neq \text{VIO}^e\right)$	$\bigwedge$
superstep	$A.1$	$\Box \left(m_e \neq \textit{no-op} \rightarrow \bigcirc \left(m_e = \textit{no-op}\right)\right)$	$\bigwedge$
	$A.2$	$\Box \left(m_s \neq \textit{no-op} \rightarrow \bigcirc \left(m_e = \textit{no-op}\right)\right)$	
		implies	
	$G.1$	$\Box\Diamond \left(m_s = \textit{no-op}\right)$	$\bigwedge$
	$G.2$	$\Box\bigcirc \left(m_e \neq \textit{no-op} \rightarrow m_s = \textit{no-op}\right)$	$\bigwedge$
system	$G.3$	$\bigwedge_{i=1}^{n} \left(l_i = \text{MIN}\right)$	$\bigwedge$
	$G.4$	$\bigwedge_{i=1}^{n} \Box \rho_{\mathcal{L}_i}$	$\bigwedge$
	$G.5$	$\Box\Diamond \bigwedge_{i=1}^{n} \left(l_i \notin Act_i\right)$	

(2)

5 Assume-Guarantee Scenarios: Synthesis

The formulation of the semantics in the GR(1) form allows us to take advantage the game-based, symbolic, efficient synthesis algorithm of [26] and generate a correct-by-construction, executable controller from a specification consisting of a set of assume-guarantee scenarios. Below we motivate the need for synthesis, in comparison with weaker forms of execution. We then give an overview of the synthesis algorithm.

5.1 Running Example

As a running example we use a scenario-based specification of a vending machine. The specification consists of six LSCs, the three LSCs presented earlier

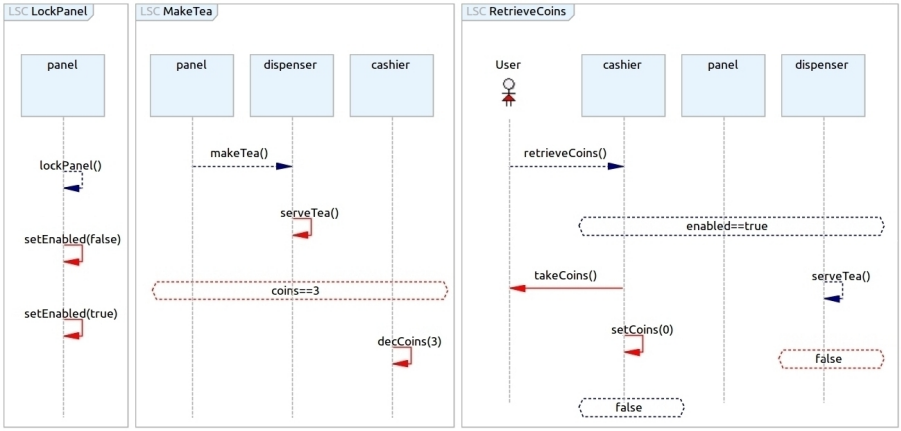

Fig. 2. Three additional LSCs from the vending machine specification (see Sect. 5)

in Fig. 1 and discussed in Sect. 2, namely `InsertCoins`, `PrepareTeaOK`, and `NoInsertCoinUntilServeTea`, and three additional LSCs, as shown in Fig. 2.

LSC `LockPanel` (Fig. 2 (left)) describes the implementation of the panel's locking mechanism. Whenever the `lockPanel` message is sent, the panel should eventually lock itself by setting its `enabled` property to false, and then eventually unlock itself by setting the `enabled` property to true.

LSC `MakeTea` (Fig. 2 (middle)) 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 after `serveTea`, 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).

LSC `RetrieveCoins` (Fig. 2 (right)) enables a cancellation functionality. If the user sends a `retrieveCoins` message to the panel and 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`.

5.2 Why Do We Need Assume-Guarantee Synthesis?

LSC specifications are underspecified, since the language allows various kinds of non-determinism. Thus, a special mechanism is needed in order to execute an LSC specification. This execution mechanism is generically termed play-out [10]. The core of the play-out process is a strategy mechanism that is responsible for

choosing the next method to execute. The choice is based on the specification and the current state of the system. Different kinds of play-out mechanisms may be defined. Each may be viewed as a different operational semantics for LSC. However, only synthesis can guarantee deadlock free execution (if one exists, see Subsect. 5.3), where if the environment behavior satisfies the assumptions then the system behavior would satisfy the guarantees. To motivate the need for assume-guarantee synthesis, we demonstrate the weaknesses of previously suggested play-out mechanisms below.

A naive operational semantics, termed play-out in [10], chooses a single system message that is enabled in some active LSC and that does not violate the current cut in all active LSCs, and executes it. Naive play-out does not guarantee that no violations will eventually occur (or rather that at each step there will be an enabled message that is not violating). Violations might happen since naive play-out makes its choices locally, without considering their future consequences.

For example, in the vending machine, after the user inserts a coin, the system must increase the coins property (LSC `InsertCoins`). After three coin insertions and sending `prepareTea`, the cold condition `coins==3` in `PrepareTeaOK` will hold and naive play-out would send a `lockPanel` message and a `heat` message to the `Heater`. Moreover, to follow LSC `LockPanel`, and since naive play-out does not consider future executions, it will immediately execute `setEnabled(false)` and `setEnabled(true)`. Now, if the user chooses to send `retrieveCoins`, the panel is enabled, the coins property will be set to zero, and so after `reachMax` is sent and `makeTea` is sent, a hot violation will be unavoidable in LSC `MakeTea`.

A better operational semantics for direct execution of scenario-based specifications is smart play-out (SPO) [7]. SPO can reason about possible violations within a single superstep. It guarantees to lead the system to a state where no LSC is active (a stable state), in preparation for the next environment message (if such a superstep exists).

However, looking only one superstep ahead is insufficient. For example, consider our vending machine specification, when `prepareTea` message is sent, smart play-out would fail to see the consequences of completing the superstep in `LockPanel`, since the violation is bound to occur only after two more supersteps (after the user will send `retrieveCoins` and the heater will send `reachMax`).

Both operational semantics presented above are rather weak and may in fact be viewed as unsound, as they may result in (partial) executions that cannot be extended to ones that satisfy the semantics of the LSC specification.

A stronger operational semantics for direct execution of scenario-based specifications is the synthesis presented in [12, 18], which we term closed synthesis. Closed synthesis reasons about the ongoing interaction between the environment and the system, and guarantees that in every state that the execution may reach, there exists a superstep that leads the system to a stable state. However, closed synthesis does not support environment assumptions. Thus, in our example, it will not be able to rely on the assumption induced by LSC `NoInsertCoinUntilServeTea` and thus would conclude that a controller cannot be synthesized: without this assumption a controller cannot be synthesized

because the user may insert a coin while the heater heats the water, and thus force the system to serve tea when `coins > 3`, which would violate the hot condition in LSC `MakeTea`.

This discussion shows that assume-guarantee synthesis is indeed required.

5.3 Assume-Guarantee Scenarios Synthesis

The solution we use for synthesis requires a winning strategy. Given a GR(1) specification, computing a winning strategy for the system is done by solving a Streett game [29] where the system tries to either satisfy all its guarantees, or constantly falsify one of the environment's assumptions. We do this following the symbolic fixpoint algorithms described in [4, 26]. 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 can be constructed from the intermediate values of the fixpoint computation (see [4, 26]). If the specification is realizable, then the construction of such a controller constitutes a solution to the synthesis problem.[2]

Going back to our example, assume-guarantee synthesis generates a controller that meets the specification. Specifically, it avoids the problems encountered by naive and smart play-out by sending the `setEnabled(false)` message but not sending the `setEnabled(true)` message until after the heater has sent `reachMax` (as it has to eventually). It also relies on the assumption that after `prepareTea` is sent, the user will not send an `insertCoin` message to the panel until `serveTea` is sent (as specified in LSC `NoInsertCoinUntilServeTea`).

6 Implementation

We have implemented our ideas using JTLV APIs [28] and integrated them into PlayGo [9]. PlayGo is an eclipse-based IDE built around the language of LSC and the play-in/play-out approach [10]. It includes a compiler that translates LSCs (given in a UML compliant form, using a profile, see [8]) into AspectJ code (based on [22, 23]), and provides means for visualization, exploration, and debugging of LSC executions. JTLV is a Java-based framework for the development of formal verification algorithms, implemented as an Eclipse plug-in. The framework provides editors and developer-friendly high-level APIs.

We extended PlayGo's visual editor to support the extended language syntax, and implemented the reduction to the GR(1) game setup. The synthesis algorithm

[2] If the specification is unrealizable, then the synthesis computation fails. We have work in progress on addressing this case [24].

itself is implemented using JTLV. Finally, the resulting controller, as computed by the algorithm, is not only statically presented to the engineer. Rather, we translate it back and represent it using a play-out strategy, by generating the Java code PlayGo can use to guide the execution of the system.

7 Conclusion and Future Work

We have presented an extension of live sequence charts that supports environment assumptions. The semantics of the extended language is given in the form of a GR(1) formula, and thus enables the efficient synthesis of a correct-by-construction controller. The work is implemented and demonstrated with running examples.

In a related work in progress [24] we deal with the debugging of unrealizable scenario-based specifications (with or without assumptions). When a specification is unrealizable, we reverse the roles of the system and the environment and compute a counter strategy [16,25]. The counter strategy serves as a formal proof that shows how an adverse environment can adhere to the assumptions (if any) while forcing any system to fail in fulfilling its guarantees.

In Sect. 4 we have defined a global stability guarantee G.5. An alternative, weaker semantics, could have used a local stability guarantee: $\bigwedge_{i=1}^{n} \Box\Diamond(l_i \notin Act_i)$. Note that this semantics may induce a more complex synthesis solution, but which is still of course within the GR(1) fragment. Moreover, the global variant implies the local one. Although we have chosen to present the global stability guarantee, we believe that the local one may be useful in some contexts and may perhaps be more in line with the breakup of the specification into scenarios. We leave the choice between the two alternative semantics open for discussion.

Finally, one may consider an alternative, tighter semantics for LSC, using the stronger GR(K) form (see Chap. 4. of [21]), which handles formulae consisting of k conjunctions of GR(1) implications. GR(K) is more expressive than GR(1) (in fact GR(K) is as expressive as LTL), however solving it is computationally harder (exponential in k, [25]). GR(1) can serve as an efficient precondition to the more locally aware formulation of GR(K).

In the context of scenario-based specifications, the essence of the difference between GR(1) and GR(K) is in the question of whether all assumptions should be grouped together into a single conjunct on the left side of the GR(1) implication, or whether each scenario should induce its own local implication between assumptions and guarantees. It is not clear whether the GR(K) semantics captures the idea of scenario-based specifications better than the GR(1) semantics. We leave the formal definition of the alternative GR(K) semantics and its evaluation against the GR(1) semantics for future work.

References

1. Alfonso, A., Braberman, V.A., Kicillof, N., Olivero, A.: Visual timed event scenarios. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) ICSE, pp. 168–177. IEEE Computer Society (2004)

2. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Autom. Softw. Eng. 14(3), 293–340 (2007)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
4. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences 78(3), 911–938 (2012)
5. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
6. Greenyer, J.: Scenario-based Design of Mechatronic Systems. PhD thesis, University of Paderborn, Department of Computer Science (2011)
7. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
8. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling 7(2), 237–252 (2008)
9. Harel, D., Maoz, S., Szekely, S., Barkan, D.: PlayGo: towards a comprehensive tool for scenario based programming. In: ASE, pp. 359–360. ACM (2010)
10. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer (2003)
11. Harel, D., Pnueli, A.: On the Development of Reactive Systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. ATO ASI Series, vol. F-13, pp. 477–498. Springer (1985)
12. Harel, D., Segall, I.: Synthesis from scenario-based specifications. Journal of Computer and System Sciences 78(3), 970–980 (2012)
13. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software and Systems Modeling 4(4), 355–367 (2005)
14. Jackson, M.: The world and the machine. In: Perry, D.E., Jeffrey, R., Notkin, D. (eds.) ICSE, pp. 283–292. ACM (1995)
15. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007)
16. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications using simple counterstrategies. In: FMCAD, pp. 152–159. IEEE (2009)
17. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: DIPES, pp. 61–72 (1998)
18. Kugler, H., Plock, C., Pnueli, A.: Controller Synthesis from LSC Requirements. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer, Heidelberg (2009)
19. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 77–91. Springer, Heidelberg (2009)
20. Kupferman, O., Vardi, M.Y.: Module Checking Revisited. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 36–47. Springer, Heidelberg (1997)
21. Manna, Z., Pnueli, A.: The temporal logic of concurrent and reactive systems: specification. Springer (1992)
22. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: SIGSOFT FSE, pp. 219–230. ACM (2006)
23. Maoz, S., Harel, D., Kleinbort, A.: A compiler for multimodal scenarios: Transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. 20(4), 18 (2011)
24. Maoz, S., Sa'ar, Y.: Counter play-out: Executing unrealizable scenario-based specifications (in preparation, 2012)

25. Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: LICS, pp. 275–284. IEEE Computer Society (2006)
26. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
27. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
28. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A Framework for Developing Verification Algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 171–174. Springer, Heidelberg (2010)
29. Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. Information and Control 54(1/2), 121–141 (1982)
30. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: ICSE, pp. 314–323. ACM (2000)