Realizability Checking of Live Sequence Charts

Daniel Ejsing-Duun, Lisa Fontani, Jonas Finnemann Jensen,
Lars Kærlund Østergaard

Department of Computer Science, Aalborg University, Denmark

Abstract. Live Sequence Charts (LSC) is a graphical formalism for defining inter-object communication in an intuitive way. Their precise semantics makes it possible for developers to check for inconsistencies in the specification of a system, composed of a set of LSCs, and to automatically synthesize a conforming controller.
We propose a syntax and semantics for a subset of LSCs and present a tool that enables the user to specify system requirements with this subset of the LSC formalism. The tool is able to determine by means of a game-theoretic approach, also introduced in this work, whether the specification of a system is realizable. In this way the tool allows for the specification of a system in an intuitive way and immediately states whether the design can be implemented or if there are any contradicting requirements.

1 Introduction

Communication between users and developers of systems has been a longstanding problem, since they often express wishes or requirements in different and incompatible ways. While users find it natural to describe the scenarios they wish to model in a system, the developers are more concerned with knowing the exact state of the system at certain points in time and what should happen in between.

Live Sequence Charts (LSC) were introduced by Damm and Harel in 1999 [5] as an extension to Message Sequence Charts (MSC), a visualization tool widely used within the industry to specify requirements and design based on inter-object communication. They use messages between system components to describe the specification of a system, thereby focusing on inter-object behavior. This way of specifying requirements makes it easy for users to express their wishes by means of scenarios that describe legal and illegal actions, without having to consider the internal state of each component and how this state may evolve. Each scenario can either specify a sequence of actions that must be executed every time a set of conditions is met through a so-called universal chart, or it can express that the system should allow the execution of a given scenario through a so-called existential chart.

An example of such a scenario can be seen in Figure 1, where a universal chart for how a coffee machine should behave when given money and asked for coffee is specified. There are two "actors" in this scenario, the coffee machine

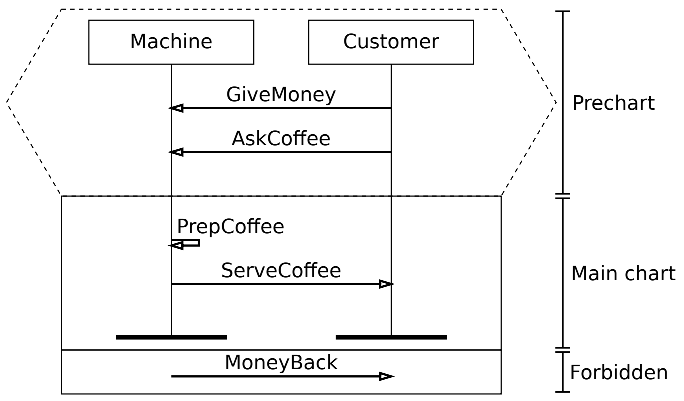


Fig. 1: A universal Live Sequence Chart specifying that each time the coffee machine receives money and is afterwards asked for coffee, it should provide the coffee without giving any money back.

and the costumer, represented by the boxes on top of the figure. The first part of the chart, delimited by a dashed line, makes up the prechart that specifies which conditions activate the next part of the chart. This is the main chart, where a sequence of actions that must happen if the prechart is satisfied, is specified. The section labeled forbidden in the Figure shows which messages are not allowed between certain objects during the execution of the chart.

So, an implementation of a coffee machine satisfying this chart must prepare and serve coffee each time it receives money and is asked for coffee, in that order. However, the machine is not allowed to give money back at any moment during the chart. If it does during the prechart, the chart is reset, while it would lead to a violation if it happened during the main chart, where it has already begun preparing the coffee. Of course, for the sake of completeness it would be necessary to add another chart stating that money cannot be given back if the coffee has already been made (assuming correct change was given) and no more money was put into the machine.

The graphical notation makes the specification and understanding of a scenario intuitive and straightforward for the user. At the same time, developers can use the charts to form the base of an implementation that satisfies the requirements specified, provided these can be satisfied.

The LSC formalism is a possible solution to the limitations related to the expressiveness of MSCs, since they only specify a certain deterministic execution of a system. Now it is also possible to define whether the specified execution is mandatory, and to include branching structures, advanced timing features and much more.

In the following sections we consider a subset of LSCs that does not include all control-flow capabilities nor conditions and messages/locations are only *hot*, meaning that they must be executed eventually. This subset provides sufficient expressive power to model non-trivial systems [1]. Furthermore, it makes it easier to develop a usable, automated tool that is practical and still allows verification and synthesis to be tractable.

In this work we will construct a graphical tool for specifying LSCs directly by means of their graphical notation. The tool will also allow checking the realizability of a system specification, expressed through LSCs, by translating it to a two-player game.

A running demo of the tool can be found on `http://jopsen.github.com/LiveSC` and the source code is available on `https://github.com/jopsen/LiveSC`.

1.1 Related Work

Automatic synthesis of systems based on LSC specifications has first been explored in [8], where it is proven that if an LSC specification is consistent then there exists an implementation satisfying it. An algorithm for checking the consistency of the specification is presented. This constructs a global system automaton which is used to construct a controller strategy that can then be distributed to and used at the different components of a system.

This idea is revisited to avoid the high complexity of the synthesis algorithms from [8] that do not scale to large systems [10]. Also, a sound but not complete implementation that generates statechart models and uses model checking to ensure realizability is presented.

Unfortunately, even for the subset of LSCs that we consider, automatic checking of realizability, also called synthesis, has been proven to be EXPTIME-complete in a centralized setting [3] and undecidable for distributed systems [19]. In the former setting we wish to obtain a common strategy for all components, while in the latter we require a strategy for each component that can be distributed and is independent of the strategies of the other components. Certain heuristics, though, can help in this area. Verifying if an implementation satisfies a specification has been shown to be PSPACE-complete for both centralised and distributed systems with environment objects [3]. The same result applies the reachability problem in general.

An unwinding approach has been used to construct Büchi automata from LSCs extended with some notion of invariants and conditions [14]. Unwinding also allows to construct a partially ordered timed symbolic automaton for the prechart and main chart of each LSC. Consequently, the problem of verifying the LSCs corresponds to a reachability problem in this automaton [13].

The synthesis problem is translated to a game structure in [15,4], where all the inputs to the system represent the environment, the system's adversary in the game, and all parts of the system together constitute the player that we wish to find a winning strategy for.

Various tools have been developed to both design LSCs and for realizability checking. The Play Engine [11] allows users to "Play in" scenarios by working with a GUI symbolizing the various functions of the system, which in turn produces a set of charts. It is also possible to check cases against the specification by "Playing out" the scenarios, i.e. the user performs the role of the environment by interacting with the GUI and observes the reaction of the system, without the need to synthesize a controller for the system. The system responds to inputs by keeping track of the conditions that make charts become active and, when needed, executing the behavior mandated by universal charts. In the meantime the system also keeps track of the execution of existential charts and marks them when they have been executed. Finally, the method has been extended with model checking to help find a correct superstep, i.e. a sequence of system messages that when executed does not violate any active universal chart [9]. This extension, called Smart play-out, is however strictly weaker than synthesis [7], since it only looks one step ahead, i.e. the next sequence of events that makes all universal charts become inactive, and does not take into consideration whether the system from this next state could receive an input that forces it to violate the specification.

An extension to the Play Engine that also implements a game theoretic approach has been presented in [15]. This extension can check for the realizability of the specification and produce a controller for the system, provided that there are no inconsistencies in the specification. The winning strategy can then be used to control the responses of the system in a play-out approach.

On the other side, the possibility of directly translating a LSC into code has been made available by the development of the BPJ library that allows the developer to use behavioral programming in Java [12]. A complex behavior between different objects can be described through simple scenario descriptions of relatively independent events. These scenarios are implemented in behavioral threads. These threads are allowed to wait for some event to happen, block because of other threads requesting it and allowing some actions to trigger other threads. Conflicts and deadlocks of behavioral threads can be checked using the same techniques as for Smart play-out for LSCs. Behavioral threads have later been extended to Erlang [21].

Unfortunately the tools mentioned either do not include realizability checking or are no longer available. Our contributions are a concise syntax and semantics for LSCs, a translation from LSCs to games and an easily accessible tool with the goal of making the LSC formalism more attractive, especially for new users. On top of that we implement on-the-fly realizability checking directly into the working environment using symbolic model checking.

This paper describes the theoretical background of the tool. First we give an introduction to the syntax and semantics of LSCs in Section 2, inspired by the formalisms presented in [2,6]. In Section 3 we describe a game structure that we shall use later on. A translation of the LSCs into games is presented in Section 4. A description of the tool itself is provided in Section 5. Finally, we provide conclusions in Section 6.

2 Live Sequence Charts

An LSC may be used to specify the system in two ways: it can either define a possible execution of the system with an existential chart or it can describe an execution that must always be satisfied given some conditions with a universal chart. These conditions can either quantify over the state of the components through a boolean expression or over a sequence of messages that has been sent. The two types are expressed through activation conditions and in precharts, respectively. When these conditions are met the scenario expressed in the body of the chart must be satisfied.

Moreover, it is possible to specify when the activation of a chart may occur by defining its activation mode, which can be *initial*, *invariant* or *iterative*. The initial mode signifies that the chart can only be activated at the start of an execution. The invariant mode activates the chart each time both the activation condition and the requirements of the prechart are met. In this way more instances of the same chart may be active at the same time. Finally, the iterative mode allows a chart to be activated again only after it has finished any initiated execution of the chart.

Each component relevant to a chart is represented by a vertical line to and from which a message can be sent. Messages can be synchronous, i.e. the message is sent and received at the same time, or asynchronous where the message is received at a later point in time. A message that is required to be sent is *hot* and is represented with a solid line in a chart. A message that may be sent is *cold* and is represented with a dashed line in a chart.

The interpretation of an LSC can either be *strict* or *weak*. A strict interpretation disallows duplicate messages, while a weak interpretation discards duplicates.

Finally an LSC focusses only on the ordering of messages that are defined in it, and allows any other message to take place in the execution of the scenario, unless it is explicitly forbidden. These constructs correspond to the constant LSC introduced in [11] and represent the core elements of a LSC.

In this work we will deal with the realizability and synthesis of a system defined through universal charts with an invariant activation mode, where only hot synchronous messages appear, with a strict interpretation and with purely synchronous messages.

We can relate this limited formalism to other extensions of LSCs that have been presented, such as invariants and conditions on the state of the components in a chart, synchronizations, loop constructs, etc. which give a higher degree of expressiveness when providing LSC descriptions. Some of these advanced features can be approximated by the subset of LSCs considered here.

2.1 Syntax

A LSC, as described in the introduction, can be divided into three main areas: the prechart, the main chart, and the forbidden messages. The first two are similar and the only difference lies in their interpretation. Therefore we can describe these commonly as basic charts. The main elements in a basic chart are the objects that send and receive messages on their associated lines, called instance lines, and the messages exchanged in the chart.

Definition 1 (Basic Chart). *A basic chart over the alphabet Σ and a finite set of objects $\mathcal{O}$ is a tuple $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$, where*

- *$\mathit{Inst} \subseteq \mathcal{O}$ is a finite set of objects where each has an associated instance line in the basic chart,*
- *$\mathit{End} : \mathit{Inst} \to \mathbb{N}_0$ maps an object of the chart to the number of messages on the object's instance line, and*
- *$\mathit{Msg} \subset \mathit{Inst} \times \mathbb{N} \times \Sigma \times \mathit{Inst} \times \mathbb{N}$ is the finite set of message exchanges, such that exchange $e = (O_1, a, m, O_2, b) \in \mathit{Msg}$ when m starts as the a'th message on the instance line for O_1 and ends as the b'th message on the instance line for O_2.*

Each number $a \in \mathbb{N}_0$, $a \leq \mathit{End}(O)$, is a location on the instance line for O. We require that any location $a > 0$ on the instance line for O_1 is used in exactly one message exchange $e \in \mathit{Msg}$, where $e = (O_1, a, m, O_2, b)$ or $e = (O_2, b, m, O_1, a)$ for any $O_2 \in \mathit{Inst}$ and $b \in \mathbb{N}$.

We shall denote all messages exchanged from O_1 to O_2 by the set Σ_{O_1,O_2}, i.e. $m \in \Sigma_{O_1,O_2}$ if there exists a message exchange $(O_1, a, m, S, O_2) \in \mathit{Msg}$ for some basic chart $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$ over the alphabet Σ. Without loss of generality we assume that a message $m \in \Sigma$ being exchanged from O_1 to O_2 is unique to these two objects. In other words, $\Sigma_{O_1,O_2} \cap \Sigma_{O'_1,O'_2} = \emptyset$, for all $O_1, O_2, O'_1, O'_2 \in \mathcal{O}$, when $O_1 \neq O'_1$ and $O_2 \neq O'_2$. Furthermore, we require that all symbols in Σ are part of some alphabet Σ_{O_1,O_2}, i.e. $\Sigma = \bigcup_{(O_1,O_2) \in 2^{\mathit{Inst}}} \Sigma_{O_1,O_2}$.

Example 1 The LSC from Fig. 1 has been modified in Fig. 2 to highlight the locations on each instance line. Both basic charts are defined over the alphabet $\Sigma = \{\text{GiveMoney}, \text{AskCoffee}, \text{PrepCoffee}, \text{ServeCoffee}, \text{MoneyBack}, \text{NOP}\}$. The formal definition of the prechart is given by the basic chart $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$, where

$$\begin{aligned}
\mathit{Inst} =& \{\text{Machine}, \text{Costumer}\} \\
\mathit{End} =& \{\text{Machine} \mapsto 2, \text{Costumer} \mapsto 2\} \\
\mathit{Msg} =& \{(\text{Costumer}, 1, \text{GiveMoney}, \text{Machine}, 1), \\
& (\text{Costumer}, 2, \text{AskCoffee}, \text{Machine}, 2)\}
\end{aligned}$$

The main chart can be defined similarly. Note that we have included a message in the alphabet, NOP, which is irrelevant to this chart. This demonstrates how the alphabet may contain more messages than actually used, which may become relevant if there are other charts in the same system using different messages. We shall revisit this when discussing the language defined by an LSC.

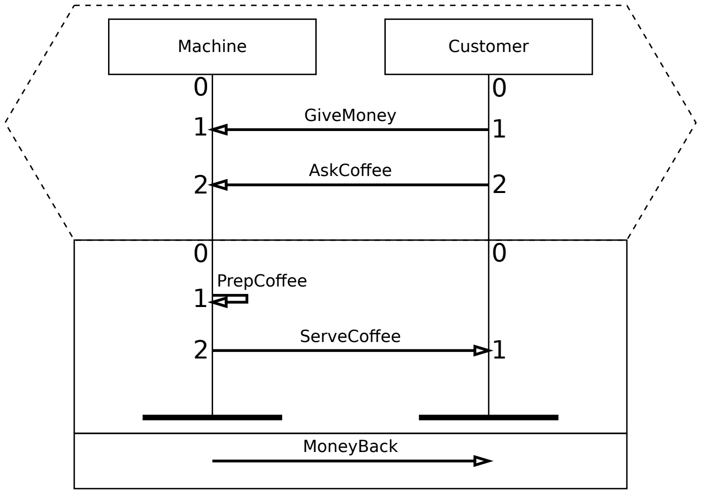


Fig. 2: A chart defining the behavior of a coffee machine and a costumer. Locations on instance lines have been highlighted.

Let $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$ be a basic chart over the alphabet Σ. For convenience let $\mathit{msgs}(B) = \{m \in \Sigma \mid (O_1, a, m, O_2, b) \in \mathit{Msg}\}$ denote the set of messages present in B. Given these basic elements we can now formally define LSCs.

Definition 2 (Live Sequence Chart). *A LSC is a tuple $L = (\mathit{Pre}, \mathit{Main}, F)$ over an alphabet Σ and finite set of objects $\mathcal{O}$, where*

- *Pre and Main are basic charts over Σ and $\mathcal{O}$, and*
- *$F \subseteq \Sigma$ is a finite set of forbidden messages, where $F \cap (\mathit{msgs}(\mathit{Pre}) \cup \mathit{msgs}(\mathit{Main})) = \emptyset$.*

Given a LSC $L = (\mathit{Pre}, \mathit{Main}, F)$, we denote the set of messages present in L by $\mathit{msgs}(L) = \mathit{msgs}(\mathit{Pre}) \cup \mathit{msgs}(\mathit{Main}) \cup F$.

Definition 3 (Specification). *A specification $\mathcal{S}$ is a finite set of LSCs over an alphabet Σ and a finite set of objects $\mathcal{O}$.*

2.2 Semantics

The semantics of an LSC is defined by an unwinding structure unfolding executing the chart step by step. Overall we use the notion of a *cut*, which can be visualized graphically as a jagged horizontal line across a basic chart, stating for each instance line how many of its messages have been exchanged so far (see Fig. 3). Progress in the chart consists of exchanging messages tied to locations, meaning that every cut can be defined as the maximum location reached on each instance line.

Definition 4 (Cut). *Let $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$ be a basic chart. A cut C is a mapping $C : \mathit{Inst} \to \mathbb{N}_0$, such that $C(O) \leq \mathit{End}(O)$ for all $O \in \mathit{Inst}$. Also, for any message exchange $e = (O_1, a, m, O_2, b) \in \mathit{Msg}$, either $C(O_1) < a \wedge C(O_2) < b$ or $C(O_1) \geq a \wedge C(O_2) \geq b$.*

The initial cut of a basic chart $B = (\mathit{Inst}, \mathit{End}, \mathit{Msg})$, denoted C_0, maps each object to the location 0. The maximal cut of B, denoted C_{max}, maps each object $O \in \mathit{Inst}$ to the maximum location $\mathit{End}(O)$ on its instance line. Formally, the initial cut C_0 and maximal cut C_{max} are defined as follows.

$$\begin{aligned} C_0(O) &= 0 && \forall O \in \mathit{Inst} \\ C_{max}(O) &= \mathit{End}(O) && \forall O \in \mathit{Inst} \end{aligned}$$

Example 2 The chart in Fig. 3 is an extended version of the previous example. Now we see that the coffee machine has been split into two objects, which also have a few new messages showing how the coffee is prepared and how a light is turned on, indicating to the costumer that the machine is working. The cut C shown over the main chart is defined as follows.

$$\begin{aligned} C(\text{WaterTank}) &= 2 \\ C(\text{Interface}) &= 1 \\ C(\text{Customer}) &= 0 \end{aligned}$$

This indicates that the request for coffee and water in the main chart has been made, but also that the light on the coffee machine has not yet been turned on.

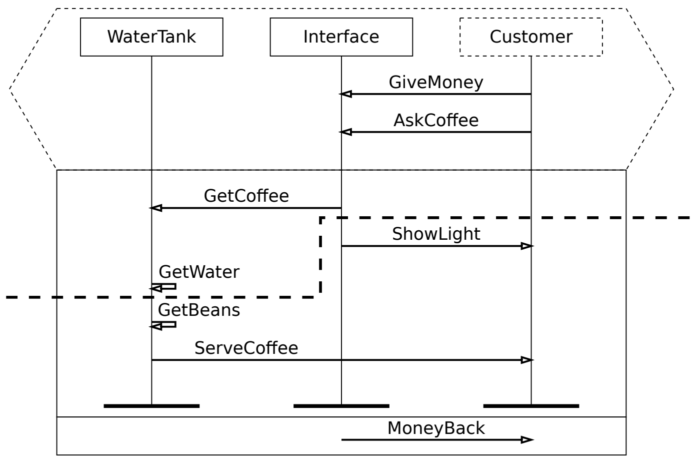


Fig. 3: A chart with a cut symbolized by a line. All messages above the line are part of the cut and the messages below are not.

Definition 5 (Cut Transition). *Given a basic chart $B = (Inst, End, Msg)$ over the alphabet Σ, a cut C and a message $m \in \Sigma$, we say that m is enabled at C if there exists $O_1, O_2 \in Inst$ such that $(O_1, C(O_1)+1, m, O_2, C(O_2)+1) \in Msg$. If m is enabled at C, the cut C can transition to C', denoted $C \xrightarrow{m} C'$, where*

$$C'(O) = \begin{cases} C(O_1) + 1 & \textit{if } O = O_1 \\ C(O_2) + 1 & \textit{if } O = O_2 \\ C(O) & \textit{otherwise.} \end{cases}$$

By advancing the execution we may eventually reach the maximal cut C_{max}, where all messages in the chart have been exchanged.

Definition 6 (Run).
Given a basic chart B, a run $\gamma = (C_0, m_1, C_1, m_2, \ldots, m_n, C_n)$ of B is a sequence of cuts and messages, where $C_n = C_{max}$ and $C_i \xrightarrow{m_{i+1}} C_{i+1}$ for all i such that $0 \leq i < n$.

We define the projection of a word w onto the alphabet Σ, denoted $w|_{\Sigma}$, to be the word w' where all symbols not in Σ have been removed. Let B be a basic chart over the alphabet Σ. We say that a word $w \in \Sigma^*$ is in the language $\mathcal{L}(B)$ of B, if there exists a run γ of B such that $\gamma|_{msgs(B)} = w$.

Example 3 Here we give an example of a run γ of the main chart shown in Fig. 3.

$$\gamma = \Big(C_0, \text{GetCoffee}, C_1, \text{GetWater}, C_2, \text{GetBeans}, \\ C_3, \text{ShowLight}, C_4, \text{ServeCoffee}, C_{max} \Big)$$

The initial cut C_0 maps every object to the location 0. The next cut C_1 maps both WaterTank and Interface to the location 1 and Customer to the location 0. The cut C_2 is the one shown in Fig. 3 (See Example 2). C_3 is the same as C_2 except that WaterTank maps to the location 3. The cut C_4 can be written as $C_4 = \{\text{WaterTank} \mapsto 3, \text{Interface} \mapsto 2, \text{Customer} \mapsto 1\}$. Below we show the word defined by the run γ. Note that spaces are used in between symbols to enhance readability.

$$\gamma|_{msgs(B)} = \text{GetCoffee GetWater GetBeans ShowLight ServeCoffee}$$

The language of the main chart in Fig. 3 is as follows.

$$\mathcal{L}(Main) = \{\text{GetCoffee GetWater GetBeans ShowLight ServeCoffee}, \\ \text{GetCoffee ShowLight GetWater GetBeans ServeCoffee}, \\ \text{GetCoffee GetWater ShowLight GetBeans ServeCoffee}\}$$

Similarly, a LSC $L = (Pre, Main, F)$ over the alphabet Σ defines a language $\mathcal{L}(L)$.

Definition 7 (Language of a LSC). *Let $L = (Pre, Main, F)$ be a LSC over the alphabet Σ.*

The language $\mathcal{L}(L)$ is a set containing all the words $w \in \Sigma^*$, such that for any decomposition xyz of w, if $y|_{msgs(L)} \in \mathcal{L}(Pre)$ then there is a decomposition uv of z such that $u|_{msgs(L)} \in \mathcal{L}(Main)$.

Recall that $msgs(L)$ describes exactly the messages used in the LSC. Moreover, note that the substrings y and u do not contain any strings from F, since the messages in F also are contained in $msgs(L)$ and as such are kept in the projected word, yet these messages are never part of any word in $\mathcal{L}(Pre)$ nor $\mathcal{L}(Main)$.

Example 4 A part of the language of the LSC L in Fig. 3 is shown below. Note that for brevity all messages are denoted by their initials (GetWater becomes GW etc.):

$$\begin{aligned}\mathcal{L}(L) = \{&\text{GM AC GC SL GW GB SC},\\ &\text{GM AC GC SL GW GB SC GM GM GM},\\ &\text{GM AC GC GW SL GB SC GM AC GC SL GW GB SC},\\ &\text{NOP NOP GM NOP AC GC GW SL NOP GB NOP SC NOP}, \ldots\}\end{aligned}$$

Since the message NOP is not part of the messages of the chart, ie. NOP $\notin msgs(L)$, it can in theory occur any number of times at any place in any of the words of the language.

Finally, we define the language of a system specification as the intersection of the language of every chart in the specification.

Definition 8 (Language of a Specification). *Let $\mathcal{S} = \{L_1, L_2, \ldots, L_n\}$ be a specification. The language of $\mathcal{S}$ is defined as follows.*

$$\mathcal{L}(\mathcal{S}) = \bigcap_{i=1}^{n} \mathcal{L}(L_i)$$

To be able to synthesize a controller for the system that adheres to a given specification, it is helpful to consider the problem as a two-player game, where the system and environment play against each other. In this setting the objects of the LSCs are partitioned into two sets: a set of objects SYS which the system has control over, and a set ENV which the environment controls (ie. uncontrollable).

$$SYS \subseteq \mathcal{O} \qquad\qquad ENV = \mathcal{O} \setminus SYS$$

If we recall that Σ_{O_1,O_2} is the set of messages that is sent by O_1 and received by O_2, we can, given a partitioning of objects $\mathcal{O}$ into ENV and SYS, define the messages that can be sent (controlled) by environment objects ENV and system objects SYS, as Σ_{ENV} and Σ_{SYS}, respectively. The two alphabets are defined as follows.

$$\Sigma_{ENV} = \bigcup_{O_1 \in ENV, O_2 \in \mathcal{O}} \Sigma_{O_1,O_2} \qquad \Sigma_{SYS} = \bigcup_{O_1 \in SYS, O_2 \in \mathcal{O}} \Sigma_{O_1,O_2}$$

Definition 9 (LSC Realizability Game). *Let $\mathcal{S}$ be a system specification over the alphabet Σ and the sets of controllable and uncontrollable objects be SYS and ENV, respectively.*

A LSC realizability game is a two-player game. In round i of the game, player one (the environment) plays $a_i \in \Sigma_{ENV}$, after which player two (the system) plays $w_i \in \Sigma^*_{SYS}$.

We say that player two wins if the iteratively constructed infinite word $a_1 w_1 a_2 w_2 \ldots$ is in the language of $\mathcal{S}$.

If there is a universal winning strategy for player two (the system), the specification $\mathcal{S}$ is said to be *realizable.*

3 Repeated Reachability Game

In this section we shall introduce a game structure, previously presented in [18], which in this paper is referred to as a repeated reachability game. Later we will translate the LSC realizability game into a repeated reachability game. Hence by solving a repeated reachability game, we can decide the realizability of the associated LSC specification. Other types of games can also be used for the same purpose [15,17]. However, the analysis of these games is more complicated and the added expressiveness do not offer any advantages for the subset of LSCs considered here.

We introduce a two-player game between the environment and the system, each having control over a finite set of variables. In each turn, the environment first makes a move and then the system is allowed to respond. Each player can make a move according to their transition relation that depends on the current value of all variables, i.e. the configuration of the game. To win, the system has to force the play into some specific configuration infinitely often.

Since this is a two-player game, the system is not able to control all transitions of the game. As such, when searching for a winning strategy, the system must for a given configuration consider both its own possible moves and those of the environment, since it always wants to force the environment to change its variables in a way such that the play ends up in a target configuration or at least a configuration that the system has a strategy for.

Formally the repeated reachability game can be defined as follows.

Definition 10 (Repeated Reachability Game). *A repeated reachability game is a tuple $\mathcal{G} = (X, Y, x_0, y_0, \rho_X, \rho_Y, W)$, where*

- *X and Y are two disjoint and finite sets of variables controlled by the environment and the system, respectively. Every variable ranges over a finite domain. Let $\mathcal{X}$ and $\mathcal{Y}$ denote the sets of all valuations of the variables in X and Y in the same order. A configuration of a game (x, y) is a valuation over both the environment and system variables, i.e. $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.*
- *$x_0 \in \mathcal{X}$ and $y_0 \in \mathcal{Y}$ are the initial valuations of the variables in X and Y, respectively. We say that (x_0, y_0) is the initial configuration.*
- *$\rho_X \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{X}$ is the transition relation for the environment, such that for any configuration (x, y) there exists at least one tuple $(x, y, x') \in \rho_X$. Each tuple (x, y, x') in this relation says that in configuration (x, y) the environment variables can transition to x'.*
- *$\rho_Y \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{Y}$ is the transition relation for the system, such that for any configuration (x, y) there exists at least one tuple $(x, y, y') \in \rho_Y$. Each tuple (x, y, y') in this relation says that in configuration (x, y) the system variables can transition to y'.*
- *$W \subseteq \mathcal{X} \times \mathcal{Y}$ is the set of target configurations.*

A strategy for the system is a function $f : \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \to \mathcal{Y}$, such that for all $x, x' \in \mathcal{X}$ and $y \in \mathcal{Y}$ it holds that $f(x, y, x') = y'$ implies $(x, y, y') \in \rho_Y$.

Given a strategy f, a play σ according to f is an infinite sequence of configurations $\sigma = [(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots]$, s.t. $(x_i, y_i, x_{i+1}) \in \rho_X$ and $f(x_i, y_i, x_{i+1}) = y_{i+1}$ for $i \geq 0$.

We say that a play σ according to strategy f is winning for the system, if some configuration $(x, y) \in W$ appears infinitely often in σ. A strategy is winning for the system if all plays according to the strategy are winning for the system.

Example 5 Consider a faculty playing the role of the environment and a computer scientist being the system. The faculty offers coffee as a compensation to the computer scientist if her results are good. For survival the computer scientist is dependent on a steady stream of coffee. We can now ask if there exists a winning strategy such that the computer scientist stays alive.

We formulate this as a game $\mathcal{G} = (X, Y, x_0, y_0, \rho_X, \rho_Y, W)$, where

$X = \{\mathit{compensation}\}$ The domain of *compensation* is $\{\mathit{coffee}, \mathit{none}\}$

$Y = \{\mathit{results}\}$ The domain of *results* is $\{\mathit{good}, \mathit{bad}\}$

$x_0 = \{\mathit{compensation} \mapsto \mathit{none}\}$

$y_0 = \{\mathit{results} \mapsto \mathit{bad}\}$

$\rho_X = \{(x, y, x') \in \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \mid y(\mathit{results}) = \mathit{good} \Rightarrow x'(\mathit{compensation}) = \mathit{coffee}\}$

For notational convenience we shall write *results* as short hand for $y(\mathit{results})$, and $\mathit{results}'$ as a short hand for $y'(\mathit{results})$. Thus, we can also write ρ_X as

$\rho_X = \{(x, y, x') \in \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \mid \mathit{results} = \mathit{good} \Rightarrow \mathit{compensation}' = \mathit{coffee}\}$

$\rho_Y = \mathcal{X} \times \mathcal{Y} \times \mathcal{Y}$ (No restrictions on the choices of the system)

$W = \{(x, y) \in \mathcal{X} \times \mathcal{Y} \mid \mathit{compensation} = \mathit{coffee}\}$

A winning strategy f for the computer scientist is $f(x, y, x') = \{\mathit{results} \mapsto \mathit{good}\}$. As in any play, where the computer scientists constantly produces good results, the faculty is required to compensate her with life essential coffee. Thus, all plays according to strategy f are winning.

3.1 Controllability Check

The winning condition of a repeated reachability game $\mathcal{G} = (X, Y, x_0, y_0, \rho_X, \rho_Y, W)$ is a recurrence property stating that a configuration in W must be reached infinitely often. This means that it is not enough to simply reach a configuration in W. Instead, for the system to win the game we must ensure that, given the transitions of the system, at any configuration of every play it will eventually lead to another configuration in W.

As presented by Pnueli in [18] we use *controllable predecessors* and fixpoint computation to determine whether or not the system can control the game to an extent where it can force visits to a configuration in W infinitely often. The controllable predecessors of a set of configurations P are denoted as $\diamondsuit P$ and defined as the following.

$$\diamondsuit P = \{(x, y) \in \mathcal{X} \times \mathcal{Y} \mid \forall x' \, [(x, y, x') \in \rho_X] \Rightarrow \exists y' \, [(x, y, y') \in \rho_Y] \wedge (x', y') \in P\}$$

Now we can use the controllable predecessors to find the configurations Z_1 from where the system can force a visit a target configuration, regardless of the choices made by the environment. These configurations can be expressed by $Z_1 = W \vee \textcircled{$\Diamond$} W \vee \textcircled{$\Diamond$}\textcircled{$\Diamond$} W \vee \ldots$. This can also be written as a minimal fixpoint.

$$Z_1 = \mu T. \textcircled{$\Diamond$} T \vee W$$

In Z_1 we now have the configurations where the system can force a visit to a target configuration. This is not enough to win as the system must force a visit to a target configuration infinitely often. For this we now construct Z_2 as the set of configurations from where the system can force a play to visit a target configuration from Z_1.

$$Z_2 = \mu T. \textcircled{$\Diamond$} T \vee (W \wedge \textcircled{$\Diamond$} Z_1)$$

Thus, from any configuration in Z_2 the system can force a visit to a target configuration twice, namely by going to a configuration (x, y) in $W \wedge \textcircled{$\Diamond$} Z_1$ and by the fact that $(x, y) \in \textcircled{$\Diamond$} Z_1$, the system can force another visit to a target configuration. We can now construct Z_3, Z_4 and so on as the sets of configurations where the system can force 3, 4 and more visits to a target configuration, respectively.

$$\begin{aligned} Z_3 &= \mu T. \textcircled{$\Diamond$} T \vee (W \wedge \textcircled{$\Diamond$} Z_2) \\ Z_4 &= \mu T. \textcircled{$\Diamond$} T \vee (W \wedge \textcircled{$\Diamond$} Z_3) \\ &\vdots \end{aligned}$$

From this series of equations we observe that Z_∞ can be expressed as a maximal fixpoint, defined as follows.

$$Z_\infty = \nu Z. \mu T. \textcircled{$\Diamond$} T \vee (W \wedge \textcircled{$\Diamond$} Z)$$

We now have Z_∞ as the set of configurations where the system can force a visit to a target configuration infinitely often. Thus, Z_∞ is the set of winning configurations, and checking for realizability amounts to checking if $(x_0, y_0) \in Z_\infty$ for the initial configuration (x_0, y_0).

A similar approach is presented in [15] and [17]. However, they allow both of the transition relations to range over x' and y', i.e. $\rho_X, \rho_Y \subset \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \times \mathcal{Y}$. This complicates the computation of controllable predecessors and the added expressiveness is not essential in the translation from LSCs.

Variable	Domain	Description
env_{msg}	$\Sigma_{ENV} \cup \{\epsilon\}$	The message being sent by the environment in the current configuration, if any.
$\ell_{O,i,B}$	$\{0,\ldots,End(O)\} \cup \{Reset, Fail\}$	The location of the current cut on the instance line for object $O \in Inst$ in instance i of basic chart $B = (Inst, End, Msg)$.
$gbuchi$	$\{0\} \cup \left\{ (j,i) \middle\| \begin{array}{l} 1 \leq j \leq n+1, \\ 1 \leq i \leq \max_j \end{array} \right\}$	The winning condition that should infinitely often be 0.

Table 1: Environment variables and their domains.

Variable	Domain	Description
sys_{msg}	$\Sigma_{SYS} \cup \{\epsilon\}$	The message being sent by the system in the current configuration, if any.
$CurrentPlayer$	$\{env, sys\}$	The player that can send a message in the next round.

Table 2: System variables and their domains.

4 Encoding Live Sequence Charts as Games

In this section we present an encoding of the LSC realizability game as a repeated reachability game.

Let $\mathcal{S} = (L_1, L_2, \ldots, L_n)$ be a LSC specification over the alphabet Σ and the finite set of objects $\mathcal{O}$ and ENV and SYS denote the sets of objects controlled by the environment and the system, respectively. We encode the LSC realizability game as a repeated reachability game, $\mathcal{G}_{\mathcal{S}} = (X, Y, x_0, y_0, \rho_X, \rho_Y, W)$, which is defined throughout the rest of this section.

Here we shall use the following notation for the sub-components of LSC L_j of the specification $\mathcal{S} = (L_1, \ldots, L_j, \ldots, L_n)$.

$$\begin{aligned} L_j &= (Pre_j, Main_j, F_j) \\ Pre_j &= (Inst_{Pj}, End_{Pj}, Msg_{Pj}) \\ Main_j &= (Inst_{Mj}, End_{Mj}, Msg_{Mj}) \end{aligned}$$

The environment variables are shown in Table 1. The variable env_{msg} decides in each turn which message the environment is sending. It takes the value ϵ whenever the environment is not sending a message. This variable is controlled by the environment. However, it is only allowed to send a message when the environment is the current player.

Recall that in the LSC realizability game we are building an infinite word $w \in \mathcal{L}(\mathcal{S})$. By Definitions 7 and 8 we have that the LSC L_j only considers messages $msgs(L_j)$. Now consider the projection of w on $msgs(L_j)$ as follows.

$$w|_{msgs(L_j)} = a_1, a_2, a_3 \ldots$$

To ensure that every decomposition of w satisfies L_j we initialize a copy of L_j starting from the initial cut for the prechart at every symbol considered by L_j. We then will have one copy starting at a_1, another starting at a_2, etc.

A variable $\ell_{O,i,B}$ is introduced for every object O in the i'th instantiated copy of the basic chart B to keep track of the current cut. The domain of these variables is augmented with two special values: the value *Reset*, which signals to other locations in the same copy of the basic chart to reset, and the value *Fail* that is given if the chart is violated and by which there are no further transitions.

By construction we ensure that a copy resets and becomes available again, once it concludes that the prechart is not satisfied or the main chart has finished. If the main chart is violated, a variable of the cut entered the *Fail* location, and the copy never finishes. Notice that the main chart is activated by the prechart, and starts at its initial cut once the prechart is satisfied.

Since w is infinite, we can also expect $w|_{msgs(L_j)}$ to be infinite, thus, one might conclude that we would need infinitely many copies of L_j. However, from the initial cut to the maximal cut of the prechart and main chart, there are at most $|Msg_{Pj}|$ and $|Msg_{Mj}|$ messages, respectively. Thus, a copy that is not violated (none of its variables have been set to *Fail*) must reset after seeing $|Msg_{Pj}| + |Msg_{Mj}|$ messages from the set $msgs(L_j)$.

Now, if we reuse copies of L_j that were previously reset, we only need a finite number of copies. Specifically, we say that we need $copies_j$ number of copies, where $copies_j$ is defined as follows.

$$copies_j = |Msg_{Pj}| + |Msg_{Mj}|$$

The variable *gbuchi* encodes a Büchi winning condition, such that it is 0 infinitely often if and only if (i) the environment is allowed to send a message infinitely often, and (ii) any instantiated main chart eventually completes. This variable either has the value of 0 or is a tuple (i, j), meaning that it is checking whether copy i of LSC L_j has a non-initial cut in the main chart.

The system variables are shown in Table 2. The sys_{msg} variable is the system's counterpart of env_{msg}, and decides in each turn which message the system is sending.

The variable *CurrentPlayer* states which player is allowed to send a message in the next round of the game. This variable is controlled by the system, enabling it to send multiple messages in response to a single environment message.

The initial configuration (x_0, y_0) of the repeated reachability game $\mathcal{G}_\mathcal{S}$ is given below.

$$\begin{array}{rr}
x_0(\ell_{O,i,Main_j}) = 0 & \forall O \in Inst_{Mj}, 1 \leq j \leq n, 1 \leq i \leq copies_j \\
x_0(\ell_{O,i,Pre_j}) = 0 & \forall O \in Inst_{Pj}, 1 \leq j \leq n, 1 \leq i \leq copies_j \\
x_0(env_{msg}) = \epsilon & x_0(gbuchi) = 0 \\
y_0(sys_{msg}) = \epsilon & y_0(CurrentPlayer) = env
\end{array}$$

We now describe the transition relation for the system, ρ_Y, of the repeated reachability game $\mathcal{G}_{\mathcal{S}}$. For convenience we write *CurrentPlayer* instead of $y(\mathit{CurrentPlayer})$ and $\mathit{CurrentPlayer}'$ instead of $y'(\mathit{CurrentPlayer})$ (and similarly for x).

$$\mathit{CurrentPlayer}' = \{\mathit{sys}, \mathit{env}\} \tag{1}$$

$$\mathit{sys}'_{\mathit{msg}} = \begin{cases} \Sigma_{SYS} \cup \{\epsilon\} & \textbf{if } \mathit{CurrentPlayer} = \mathit{sys} \\ \epsilon & \textbf{otherwise} \end{cases} \tag{2}$$

We define that $(x, y, y') \in \rho_Y$ if Equations (1) and (2) hold. If there are more cases, the variable takes the value of the first case from the top for which the condition holds. If there is a set of values, any of them are possible assignments of the variable in y'. This semantics is rather similar to the one specified for SMV in [16], and indeed it is trivial to express the rules that make up the corresponding transition in SMV.

The rule in Equation (1) says that $y'(\mathit{CurrentPlayer})$ can take on the value of either *env* or *sys*.

The other rule (Equation (2)) says that if $y(\mathit{CurrentPlayer}) = \mathit{sys}$, then $y'(\mathit{sys}_{\mathit{msg}})$ can be any value from the set $\Sigma_{SYS} \cup \{\epsilon\}$. If case one ($y(\mathit{CurrentPlayer}) = \mathit{sys}$) does not hold, $y'(\mathit{sys}_{\mathit{msg}})$ takes on the value ϵ. Notice how these rules do not impose many restrictions on the system, thus, leaving a lot of freedom to the system when choosing a strategy.

Similarly, we shall describe the transition relation for the environment, ρ_X. We assume the same notation and semantics for these equations as introduced for the transition relation for the system.

Before we introduce the equations for ρ_X, we present two auxiliary functions, $\mathit{next}_{i,B}(O, a)$ and $\mathit{see}(m)$, and three auxiliary definitions $\mathit{active}_{i,j}$, $\mathit{started}_{i,j}$ and $\mathit{maymove}_{i,j}$.

$$\mathit{next}_{i,B}(O, a) = (\ell_{O,i,B} = a - 1) \lor (\ell_{O,i,B} = \mathit{Reset} \land a = 1)$$

$$\mathit{see}(m) = \begin{cases} \mathit{env}_{\mathit{msg}} = m & \textbf{if } m \in \Sigma_{ENV} \\ \mathit{sys}_{\mathit{msg}} = m & \textbf{otherwise} \end{cases}$$

$$\mathit{active}_{i,j} = \bigwedge_{O \in \mathit{Inst}_{Pj}} \ell_{O,i,\mathit{Pre}_j} = \mathit{End}_{Pj}(O)$$

$$\mathit{started}_{i,j} = \bigvee_{O \in \mathit{Inst}_{Pj}} \ell_{O,i,\mathit{Pre}_j} \neq \mathit{Reset} \land \ell_{O,i,\mathit{Pre}_j} \neq 0$$

$$\mathit{maymove}_{i,j} = \mathit{started}_{i,j} \lor \bigwedge_{1 \le c < i} \mathit{started}_{c,j}$$

The auxiliary function $next_{i,B}(O,a)$ is true if the next location on the instance line of O in the i'th copy of basic chart B is a. Notice that if $a = 1$, then the location variable for O in the copy can be either 0 or *Reset*, thus, allowing the reset location *Reset* to be treated as the initial location.

The auxiliary function $see(m)$ is true if the message m is sent in the current turn.

The auxiliary definition $active_{i,j}$ is true, if the location variables of the i'th copy of the prechart Pre_j for LSC L_j have advanced to the maximal cut, making the i'th copy of the main chart $Main_j$ of L_j active.

The definition $started_{i,j}$ is true if at least one location on an instance line in copy i of the prechart of LSC L_j is different from *0* and *Reset*, meaning that it has begun matching a word.

Finally, the definition $maymove_{i,j}$ tells us whether copy i of LSC L_j should react on the next message. Since we only allow one copy to start matching a word at the occurrence of any message, copy i should react on the message, if it has already started matching the word or all copies before copy i of L_j have begun matching words, meaning that this one is the next in line.

$$env'_{msg} = \begin{cases} \Sigma_{ENV} \cup \{\epsilon\} & \textbf{if } CurrentPlayer = env \\ \epsilon & \textbf{otherwise} \end{cases} \tag{3}$$

$$\ell_{O_1,i,Pre_j}{}' = \begin{cases} 0 & \textbf{if } \exists O_2 \in Inst_{Pj} \setminus \{O_1\} \\ & \quad \text{s.t. } \ell_{O_2,i,Pre_j} \neq Reset \\ & \quad \text{and } \ell_{O_2,i,Pre_j}{}' = Reset \\ 0 & \textbf{if } \exists O_2 \in Inst_{Mj} \text{ s.t. } \ell_{O_2,i,Main_j} \neq Reset \\ & \quad \text{and } \ell_{O_2,i,Main_j}{}' = Reset \\ \ell_{O_1,i,Pre_j} & \textbf{if } active_{i,j} \\ a & \textbf{if } next_{i,Pre_j}(O_1,a) \wedge next_{i,Pre_j}(O_2,b) \\ & \wedge\ see(m) \wedge maymove_{i,j} \\ & \quad \text{where } (O_1,a,m,O_2,b) \in Msg_{Pj} \\ & \quad \text{or } (O_2,b,m,O_1,a) \in Msg_{Pj} \\ Reset & \textbf{if } see(m) \\ & \quad \text{where } m \in msgs(L_j) \cap \Sigma_{O_1,O_2},\ O_2 \in \mathcal{O} \\ 0 & \textbf{if } \ell_{O_1,i,Pre_j} = Reset \\ \ell_{O_1,i,Pre_j} & \textbf{otherwise} \end{cases} \tag{4}$$

$$
{\ell_{O_1,i,Main_j}}' = \begin{cases}
0 & \textbf{if } \exists O_2 \in \mathit{Inst}_{Mj} \setminus \{O_1\} \\
& \quad \text{s.t. } \ell_{O_2,i,Main_j} \neq \mathit{Reset} \\
& \quad \text{and } {\ell_{O_2,i,Main_j}}' = \mathit{Reset} \\
\ell_{O_1,i,Main_j} & \textbf{if } \neg \mathit{active}_{i,j} \\
\mathit{Reset} & \textbf{if } \mathit{next}_{i,Main_j}(O_1, \mathit{End}_{Mj}(O_1)) \\
& \quad \wedge\ \mathit{next}_{i,Main_j}(O_2, \mathit{End}_{Mj}(O_2)) \wedge \mathit{see}(m) \\
& \quad \wedge \bigwedge_{O \in \mathit{Inst}_{Mj} \setminus \{O_1,O_2\}} \ell_{O,i,Main_j} = \mathit{End}_{Mj}(O), \\
& \quad \text{where } (O_1, a, m, O_2, b) \in \mathit{Msg}_{Mj}, \\
& \quad a = \mathit{End}_{Mj}(O_1),\ b = \mathit{End}_{Mj}(O_2) \\
a & \textbf{if } \mathit{next}_{i,Main_j}(O_1, a) \wedge \mathit{next}_{i,Main_j}(O_2, b) \\
& \wedge\ \mathit{see}(m) \quad \text{where } (O_1, a, m, O_2, b) \in \mathit{Msg}_{Mj} \\
& \quad \text{or } (O_2, b, m, O_1, a) \in \mathit{Msg}_{Mj} \\
\mathit{Fail} & \textbf{if } \mathit{see}(m) \\
& \quad \text{where } m \in \mathit{msgs}(L_j) \cap \Sigma_{O_1,O_2},\ O_2 \in \mathcal{O} \\
\ell_{O_1,i,Main_j} & \textbf{otherwise}
\end{cases} \tag{5}
$$

$$
\mathit{gbuchi}' = \begin{cases}
(1,1) & \textbf{if } \mathit{gbuchi} = 0 \\
0 & \textbf{if } \mathit{CurrentPlayer} = \mathit{env} \\
& \wedge\ \mathit{gbuchi} = (\mathit{copies}_n + 1, n) \\
(1, j+1) & \textbf{if } \neg \mathit{active}_{i,j} \wedge \mathit{gbuchi} = (\mathit{copies}_j, j) \\
(i+1, j) & \textbf{if } \neg \mathit{active}_{i,j} \wedge \mathit{gbuchi} = (i, j) \\
\mathit{gbuchi} & \textbf{otherwise}
\end{cases} \tag{6}
$$

We can now move on with defining the transition relation of the environment. First of all, Equation (3) allows the environment to send messages whenever it is the current player.

All the other environment variables and rules are deterministic as they keep track of the current cuts and how they advance through the copies of the charts, in order to determine if the winning condition of the LSC realizability game is satisfied.

Equation (4) specifies how the cut in the i'th copy of the prechart Pre_j of LSC L_j advances on the instance line of O_1. The first two cases state that the current location variable is set to the value 0 if another location variable in the i'th copy of LSC L_j goes to the reset location *Reset*.

The introduction of the location *Reset* is due to this condition. If 0 were used instead, the location variables of the chart copy would agree on resetting each turn, and thus, deviate from the semantics of LSCs. Hence, the value *Reset* signals to other location variables in the same chart copy that they should go to 0. As such, *Reset* has the same meaning as 0 when considering the next message to react upon.

The third case says that we do not change the value of ℓ_{O_1,i,Pre_j} if the corresponding main chart copy of the LSC is active, unless specified otherwise by case 1 or 2.

The next case says that O_1 advances to location a if this copy is supposed to react on the message and there is a message exchange at location a on the instance line of O_1 in the i'th copy of Pre_j, where both the message and the next locations of sender and receiver match those of the message exchange taking place in the current turn.

If none of the cases above were satisfied, and we see a message used in LSC L_j which should have been handled on O_1, then the fifth case says that this copy of the prechart must be reset.

The sixth case is reached if none of the cases above were fulfilled. It changes the value from *Reset* to 0, ensuring that this instance line can again reset the location of the other instance lines in the same copy of the chart by going to *Reset*. This is relevant when some locations of other instance lines in the same copy of the chart are different from 0 and a message used in the chart that is not allowed at the current location of the instance line of O_1, is seen.

Finally, if no condition was fulfilled, the last case returns the old value of the variable.

Equation (5) specifies how the cut in the i'th copy of the main chart $Main_j$ of LSC L_j advances on the instance line of O_1. The cases are quite similar to the cases for the location variable of the prechart (Equation (4)). However, the case that would reset if we see an unhandled message, now goes to the special location *Fail*, emphasizing that the LSC has been violated.

Finally, we have the winning condition *gbuchi* in Equation (6). This variable will infinitely often be assigned the value 0 if and only if every copy of every chart is inactive (either completed or never activated) infinitely often, and the environment is the current player infinitely often. Therefore the variable iterates through all copies of all charts to check if they are inactive and finally checks if the environment is the current player.

Summing all this up, we say that $(x, y, x') \in \rho_X$ if Equations (3), (4), (5) and (6) hold.

Using the variable *gbuchi*, we can define the set of target configurations W for the repeated reachability game, $\mathcal{G}_{\mathcal{S}}$, as follows.

$$W = \{(x, y) \mid x(gbuchi) = 0\}$$

We now make the following claim of correctness.

Claim. A specification $\mathcal{S}$ is realizable if and only if there is a winning strategy f for the repeated reachability game $\mathcal{G}_{\mathcal{S}}$.

Let $\mathcal{S} = \{L_1, \ldots, L_n\}$ be a specification. From the definition of the language of a specification (Definition 8) we know that a word w is in the language $\mathcal{L}(\mathcal{S})$ if $w \in \mathcal{L}(L)$ for all $L \in \mathcal{S}$.

Let $L_j = \{Pre_j, Main_j, F_j\}$ be an LSC, where $L_j \in \mathcal{S}$. The location variables for LSC L_j in the game keep track of the word $v = w|_{msgs(L_j)} = a_1a_2\ldots$ given so far, and each possible substring $\hat{v}$ of the word is considered as there is a copy of the chart ready to match from the start symbol of $\hat{v}$.

Now we are interested in determining if any substring $\hat{v}$ of v contains a sequence of symbols that forms a word $v_{Pre_j} \in \mathcal{L}(Pre_j)$. If there is a word v_{Pre}, such that there it not a following sequence of relevant symbols that constitutes a word $v_{Main_j} \in \mathcal{L}(Main_j)$, at least one location will change to the location *Fail* in some copy i of $Main_j$. As such, this copy will never be able to become inactive again. Thus, it is not the case that $gbuchi = 0$ infinitely often, because it will be stuck at the value (i, j), and the system will lose the game.

Conversely, if the word w is in the language of the specification $\mathcal{L}(\mathcal{S})$, then the location variables for all copies of the LSCs will do either of the following. They may never enter the main chart or each time they do, they will reach the maximal cut of the chart, thus allowing $gbuchi$ to be 0 infinitely often.

5 LiveSC - A Live Sequence Chart Editor

To complement the subset of LSCs formally described in this report, we have implemented a web application called LiveSC for drawing and checking realizability of LSCs. This web applications runs in most modern browsers[1]. LiveSC is loaded with examples and you can test it at `http://jopsen.github.com/LiveSC`, no installation required (except Java-plugin).

LiveSC is implemented in HTML5 using CoffeeScript and a number of Javascript libraries. For more information visit `https://github.com/jopsen/LiveSC`.

The realizability and synthesis engine is implemented as a Java applet using the JTLV library [20]. This library enables us to load SMV transition system as BDDs (Binary Decision Diagrams). Then using BDD operations we can compute controllable predecessors and the winning configurations Z_∞ using maximum and minimum fixpoints as described in Section 3.1.

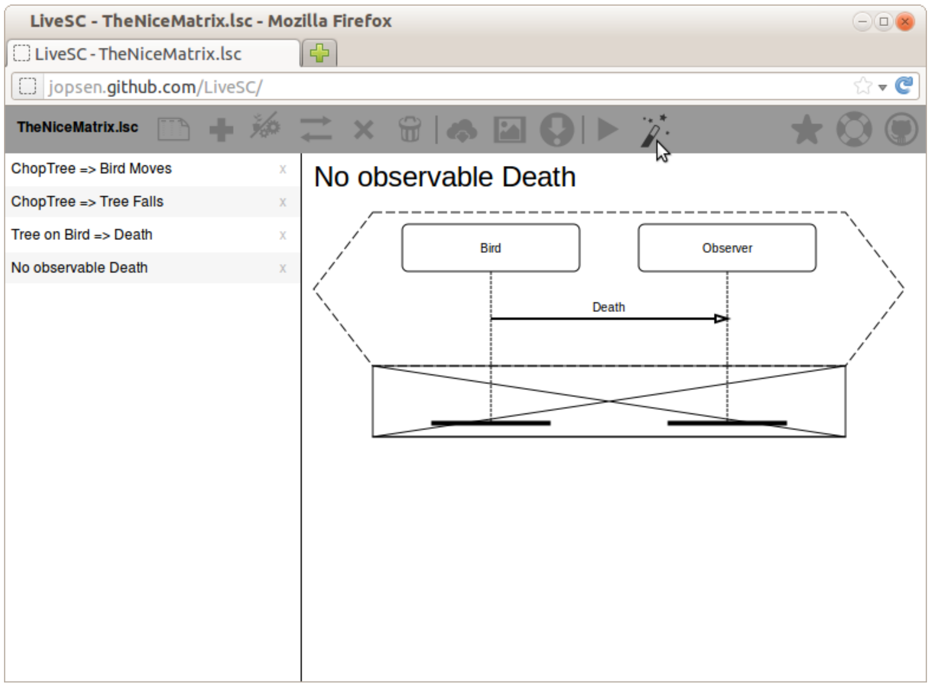


Fig. 4: A screenshot of the LiveSC tool.

Fig. 4 shows a screenshot of LiveSC working on one of the built-in examples. The tool has three main graphical elements: a toolbar in the top, a menu list on the left and a workspace on the right. The menu list contains the names of the LSCs that make up the specification currently open in the editor, while the LSC

[1] We recommend Firefox 12, Google Chrome 20 or later.

the user is currently working on is shown in the workspace area. Here the user can add, rename, move or delete messages and instance lines.

The toolbar allows the user to create, load or save specifications. Moreover, it allows for the creation of new elements such as charts, messages and objects in a drag-and-drop fashion.

It is possible to specify that the whole main chart as false, i.e. its language is empty. In this way the sequence of messages seen in the prechart is forbidden, as any run satisfying the prechart of the LSC will not be able to satisfy the main chart.

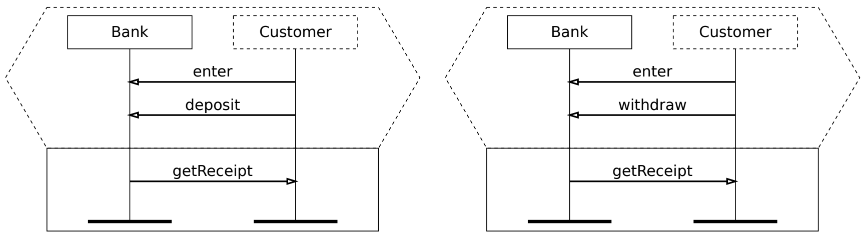
According to the semantics of LSCs given in Section 2 a chart with empty language can be created by crossing two synchronous messages, a setup that is not strictly prohibited by the definition of basic charts in order to facilitate this feature.

For the sake of synthesis the toolbar also has a button allowing to change the objects in a specification from system to environment and vice versa. The highlighted button allows the user to synthesize a strategy. However, at this point LiveSC does not have any method for exporting the synthesized strategy, so this will just synthesize a strategy and report the number of nodes in the BDD representation of the transition system.

5.1 Bank Account Example

In this section we shall consider a simple example involving a bank that demonstrates how the tool works.

Fig. 5 displays two scenarios for our bank example. This specification consists of a bank controlled by the system and a customer controlled by the environment. Scenario 5a involves the customer entering the bank, making a deposit and getting a receipt. Essentially, the scenario shows that if the customer enters the bank and makes a deposit he gets a receipt. Scenario 5b involves a withdrawal instead.

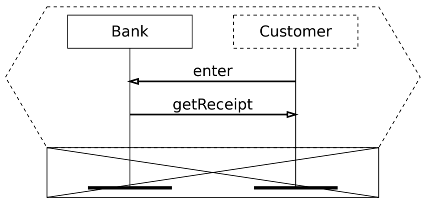


(a) Scenario describing a deposit. (b) Scenario describing a withdrawal.

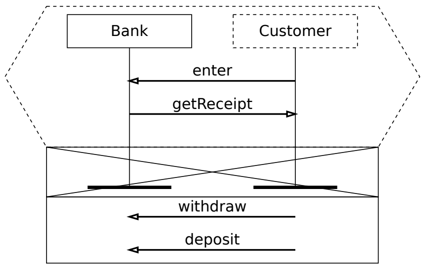
Fig. 5: Two common bank scenarios.

Now if the customer were to enter the bank, and the bank decided to issue a receipt immediately, the bank would not be forced to issue a receipt when the customer makes a deposit. This is due to the fact that the occurrence of the message "getReceipt" in the prechart of scenario 5a resets the prechart.

To further regulate the behavior of the bank, we introduce scenario 6a, which states that a customer cannot enter the bank and get a receipt. The cross drawn onto the main chart signifies that it is false. Hence, if the prechart ever completes, the main chart will cause the system to fail.



(a) Forbids receipts to customers.



(b) Forbids receipts to customers with any other actions.

Fig. 6: Two different attempts to regulate the bank.

However, with scenario 6a under consideration the system becomes unrealizable. As a customer who decides to enter and then makes a deposit will by scenario 5a be granted a receipt in violation of scenario 6a.

This is because scenario 6a does not consider the message ”deposit”. If we extend 6a to become scenario 6b the specification will again be realizable. This is because the prechart of 6b is reset, whenever a deposit or withdrawal is made.

In the example gallery for LiveSC[2] you will find this example in both its unrealizable version with scenario 6a and in its realizable version with scenario 6b. The realizability check for these examples can be done in a Java enabled browser.

The synthesized transition system, for the bank example with scenario 6b, has 127 422 BDD nodes in its representation. If for instance the *withdraw* scenario is excluded the BDD will have 34 340 nodes. In comparison the synthesized transition system for a simple system with only one chart, two objects and two messages back and forth (one in the prechart and the other in the main chart) contains 3 318 BDD nodes. An example of such a system could simply be the *deposit* scenario in Fig. 5, without the deposit message.

These preliminary numbers indicate that the size of the transition system quickly blows up as more charts and messages are added to a system.

[2] Try it at `http://jopsen.github.com/LiveSC`

6 Conclusion

We have given a syntax and semantics for a subset of Live Sequence Charts (LSCs). This allowed us to define the problem of LSC realizability from a game-theoretic perspective, where the LSC message exchanges and their ordering may be expressed through transition rules.

We have described this general problem as a LSC realizability game, which in turn is translated into a repeated reachability game, also described here. Using this we can now algorithmically determine whether a LSC specification is realizable by computing a winning strategy for the repeated reachability game.

Finally, the algorithm has been implemented in our tool called LiveSC, which is publicly available. This allows the user to create a system specification consisting of LSCs and determine whether or not the specification is realizable.

6.1 Future Work

The subset of LSCs presented here, also encoded in the tool, only permits restricted kinds of constructs. Other constructs can be introduced while still keeping the problem decidable. These include, but are not limited to, existential charts, finite domain variables, conditions and invariants on messages or variables, scopes etc.

It is of interest to extend the tool so it is possible to provide a counterexample that tells the user which charts and the sequence of messages that cause a specification to be unrealizable. Hence, making it easier to locate and understand the problem, or fix the specification if possible.

The algorithms used for translating the LSC realizability game and solving the repeated reachability game have not yet been analyzed in terms of complexity. Moreover, it is worth investigating how to optimize the algorithm used by the tool and reduce the size of the transition system of the synthesized controller, which in turn may yield faster execution and cause more complex specifications to become tractable.

It is also interesting to attempt reordering the variables of the underlying BDDs and optimize the translation of the LSC realizability game to the repeated reachability game, such that the state space and transition systems are smaller and easier to analyze.

Our current translation has a high focus on simplicity as opposed to size. This is reflected by the fact that each location variable for an instance has a fail state, whilst with complicated conditions it would be sufficient to have one global variable with a fail state for all charts.

Finally, a formal proof of correctness for the translation is required to show that the solution to the repeated reachability game matches the defined LSC semantics.

References

1. Yves Bontemps. *Relating inter-agent and intra-agent specifications : the case of life sequence charts*. PhD thesis, 2005, `dial.academielouvain.be/vital/access/services/Download/boreal:4222/PDF_01`.
2. Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Eng.*, 31(12):999–1014, 2005.
3. Yves Bontemps and Pierre-Yves Schobbens. The complexity of live sequence charts. In *FoSSaCS*, pages 364–378, 2005.
4. Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inform.*, 62(2):139–169, 2004.
5. Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. In Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri, editors, *FMOODS*, volume 139 of *IFIP Conference Proceedings*. Kluwer, 1999.
6. Werner Damm, Tobe Toben, and Bernd Westphal. On the expressive power of live sequence charts. In *Program Analysis and Compilation*, pages 225–246, 2006.
7. David Harel, Amir Kantor, and Shahar Maoz. On the power of play-out for scenario-based programs. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Concurrency, Compositionality, and Correctness*, pages 207–220. Springer-Verlag, Berlin, Heidelberg, 2010.
8. David Harel and Hillel Kugler. Synthesizing state-based object systems from LSC specifications. In *Revised Papers from the 5th International Conference on Implementation and Application of Automata*, CIAA '00, pages 1–33, London, UK, UK, 2001. Springer-Verlag.
9. David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart play-out of behavioral requirements. In Mark Aagaard and John W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, pages 378–398. Springer, 2002.
10. David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: generating statechart models from scenario-based requirements. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, pages 309–324. Springer-Verlag, Berlin, Heidelberg, 2005.
11. David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
12. David Harel, Assaf Marron, and Gera Weiss. Programming coordinated behavior in java. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 250–274, Berlin, Heidelberg, 2010. Springer-Verlag.
13. Jochen Klose, Tobe Toben, Bernd Westphal, and Hartmut Wittke. Check it out: on the efficient formal verification of live sequence charts. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 219–233, Berlin, Heidelberg, 2006. Springer-Verlag.
14. Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 512–527, London, UK, UK, 2001. Springer-Verlag.

15. Hillel Kugler, Cory Plock, and Amir Pnueli. Controller synthesis from LSC requirements. In *FASE*, pages 79–93, 2009.
16. K. L. McMillan. The SMV System, 1992-2000, `http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps`.
17. Cory Plock. *Synthesizing executable programs from requirements*. PhD thesis, New York, NY, USA, 2008.
18. Amir Pnueli. Extracting controllers for timed automata. Technical report, 2005, `http://laser.inf.ethz.ch/2005/reports/extracting.pdf`.
19. Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757, 1990.
20. Amir Pnueli, Yaniv Sa'ar, and Lenore Zuck. JTLV - a framework for developing verification algorithms. 2010.
21. Guy Wiener, Gera Weiss, and Assaf Marron. Coordinating and visualizing independent behaviors in erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 13–22, New York, NY, USA, 2010. ACM.