Controller Synthesis from LSC Requirements[⋆]

Hillel Kugler[1], Cory Plock[1], and Amir Pnueli[2]

[1] Computational Biology Group, Microsoft Research, Cambridge, UK
{hkugler,v-coploc}@microsoft.com
[2] Computer Science Department, New York University, New York, NY, USA
amir@cs.nyu.edu

Abstract. Live Sequence Charts (LSCs) is a visual requirements language for specifying reactive system behavior. When modeling and designing open reactive systems, it is often essential to have a guarantee that the requirements can be satisfied under all possible circumstances. We apply results in the area of controller synthesis to a subset of the LSC language to decide the realizability of LSC requirements. If realizable, we show how to generate system responses that are guaranteed to satisfy the requirements. We discuss one particular implementation of this result which is formulated as an extension of smart play-out, a method for direct execution of scenario-based requirements.

1 Introduction

Going directly from requirements to a correct implementation has long been a "holy grail" for system and software development. According to this vision, instead of implementing a system and then working hard to apply testing and verification methods to prove system correctness, a system is rather built correctly by construction. Synthesis is particularly challenging for reactive systems, in which the synthesized system must satisfy the requirements for any possible behavior of an external environment [2,25].

One formal specification language for reactive systems is Live Sequence Charts (LSCs) [4]. LSCs is a visual language, extending the classical message sequence charts with the ability to specify both safety and liveness properties. A methodology called the play-in/play-out approach was described in [11] as part of a tool called the Play-Engine. Play-in provides an intuitive means of capturing requirements by interacting with a graphical representation of the system, while play-out executes the scenarios in a way that gives a feeling of running an implementation of the system.

An improvement to play-out called *smart play-out* is introduced in [9]. This approach uses verification methods—in particular, model-checking—to run LSC specifications and avoid certain violations that may occur in the original version of play-out. Unfortunately, smart play-out cannot avoid all possible violations [7]. This paper addresses an improvement to smart play-out which guarantees non-violation over all computations, provided that the requirements are realizable. To accomplish this, we reformulate the previous model checking problem instead as a synthesis problem.

[⋆] This research was supported in part by NIH grant R24-GM066969 and a donation by Robert B. K. Dewar and Edmond Schonberg.

We view the problem as a two-player open game between the *system* and the *environment*. The system refers to the components of an executable program we wish to construct; the environment represents external entities which produce system inputs. The system attempts to *win* the game by satisfying the LSC requirements, whereas the environment's goal is to foil the system by steering the game into a violating state. The game is carried out using a special transition system called a *game structure* that encodes the logic of user-supplied LSC requirements.

Given a game structure, the work of [24,23] provides a means of deciding realizability, which amounts to determining if a reactive system is capable of avoiding violation over all inputs and across all runs. If so, a transition system called a *controller* is extracted. The controller encodes the so-called *winning strategy* as the original input transition system with non-winning transitions removed to avoid violating the safety properties, and guards (possibly) added to certain edges to ensure satisfaction of the liveness properties. By following the transitions of the resulting controller, satisfaction of the complete requirements is guaranteed.

In this paper, we describe how to construct a game structure that expresses the behavior of a subset of the LSC language. We apply the results of [24], with certain modifications that allow us to deal with some advanced LSC constructs in a natural way, to determine realizability and extract the controller, provided it exists. If so, we use the winning strategy to choose correct system responses to every environment input. Responses are guaranteed to exist, provided the requirements are realizable.

The paper is organized as follows. Related work is discussed in section 2. We discuss the contributions and shortcomings of smart play-out in greater detail in section 3 and motivate the need for this work in section 4. We provide definitions in section 5 and a description of our synthesis methodology in section 6. Our main result is discussed in section 7 and conclusion in section 8.

2 Related Work

In recent years there have been considerable research efforts on synthesizing executable systems from scenario-based requirements [15,16,20,17,31,30,28,21,12]. In many of these papers, the requirements are given using a variant of classical message sequence charts and the synthesized system is state-based. Although there are many common aspects to our work and these papers, the main distinguishing feature of LSCs is that they are more expressive than most of the classical MSC variants.

Synthesis from LSCs was first studied in [8], and is tackled there by defining consistency, showing that LSC requirements are consistent iff they are satisfiable by a state-based object system. A satisfying system can then be synthesized. This line of work was continued in [10], which includes an implementation of a sound but not complete algorithm for Statechart synthesis. A game theoretic approach to synthesis from LSCs involving a reduction to parity games is described in [3]. Synthesis from LSCs using a reduction to CSP appears in [27]. All the above papers were either theoretical and did not include an implementation, or the synthesis approach was sound but not complete, or the synthesis time complexity was not encouraging. An alternative strategy for synthesis from LSCs is to use a translation from LSCs to temporal logic [18,5] or automata [14] and then apply existing synthesis algorithms, e.g., [25,29].

In [24] a controller synthesis implementation for generalized Büchi winning conditions in the language of TLV-BASIC [26] is presented. The work is later extended in [23] to include Reactive(1) designs, or generalized Streett winning conditions. Neither of the results are specific to LSC requirements, but we utilize a modified version of the former implementation for our present work. In recent work [19], a compositional synthesis approach for a core subset of LSCs containing only messages is presented. The main contribution of [19] is the compositional approach, whereas this paper focuses on the basic synthesis algorithm for a wider LSC subset.

3 Smart Play-Out

Smart play-out [9] is a method for direct execution of scenario-based requirements, which allows a user to interact with an executing reactive system whose behavior was specified using Live Sequence Charts [4]. The user first creates the LSC requirements using play-in by manipulating the user interface of the target application (e.g., by pressing buttons, rotating knobs, etc.)

Once the requirements have been specified, smart play-out allows the user to play the role of the environment by injecting input events and then observe system responses that follow according to the requirements. More specifically, smart play-out accepts input events only when the Play-Engine is in a so-called *stable state*. Whenever an input (environment) event is injected within a stable state, smart play-out formulates a response— a sequence of outputs events called a *superstep*—which leads the computation to another stable state, provided a superstep exists. The main contribution of smart play-out is the means through which supersteps are identified and executed.

Smart play-out finds supersteps by first encoding the logic of LSCs into a transition system and then formulating a model checking problem for the specified environment input. Roughly speaking, smart play-out tries to verify the property "no superstep leading to a stable state exists" with the hope that the property is false. If it is indeed false, the model checker produces a counterexample as a witness to the existence of the superstep. Smart play-out then feeds the counterexample to the Play-Engine so that the user may witness the superstep being carried out graphically.

One limitation of smart play-out is that the model checking procedure explores the state space only to the extent necessary to identify a superstep leading to some successor stable state. The procedure disregards whether any supersteps exist from the successor stable state, or any stable state thereafter. Therefore, smart play-out may blindly lead the system into a state from which no superstep exists—a violation of the requirements, since reactive systems must supply a (correct) response to every environment input.

When the user sees the violation, they may arrive at an inaccurate conclusion that something is wrong with the requirements, when in fact the violation was due to the Play-Engine's poor choice of supersteps. Better selection of supersteps could have yielded non-violation instead. The main problem is that the supersteps (i.e., counterexamples) seem to be chosen arbitrarily by the model checker. By choosing supersteps more wisely, it is possible to identify a priori whether supersteps exist for *all* possible

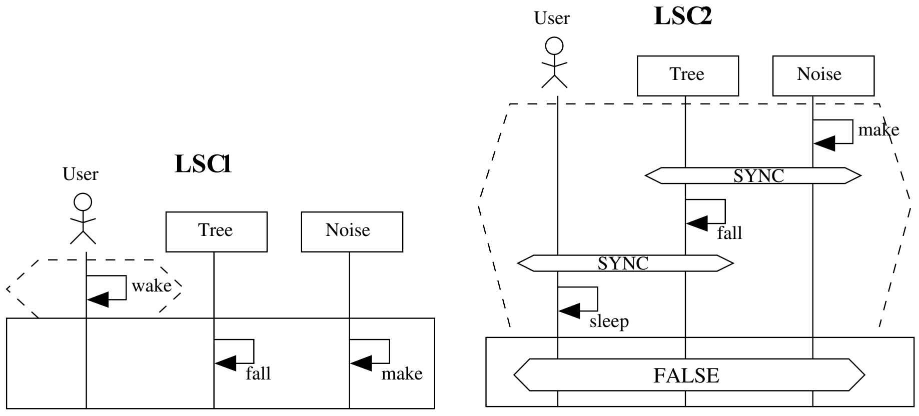

Fig. 1. LSC Requirements

sequences of inputs. To achieve this, we use synthesis techniques to perform a complete analysis of the state space. This allows for forward-looking decisions and complete avoidance of violations, provided the requirements are realizable.

4 Example

To solidify the above discussion with an example, consider LSC requirements consisting of the two LSCs shown in Fig. 1. Both scenarios include USER as an environment instance, and both TREE, NOISE as system instances. Accordingly, the behaviors of TREE and NOISE are within the control of the system we intend to construct, whereas the behaviors of the USER are assumed to be external.

According to LSC1, whenever USER sends the *wake* message, the controller must respond with a non-deterministic ordering of message *fall* and *make* in order to satisfy the main chart. Therefore, the traces *wake*, *fall*, *make* or *wake*, *make*, *fall* are both acceptable for satisfying LSC1. On the other hand, LSC2 is an *anti-scenario* that specifies the sequence *make*, *fall*, *sleep* cannot ever occur. The synchronizing conditions remove the otherwise non-deterministic ordering of *make*, *fall*, *sleep* to ensure that only traces with this precise ordering will satisfy the prechart.

We now consider how smart play-out might respond to an input message *wake* executed by the USER. After formulating a model-checking problem that checks for the non-existence of a satisfying trace, the resulting counter-example yields one of the two possible event sequences above. Supposing that smart play-out executes the sequence: *make*, *fall*, LSC1 would be satisfied, but the prechart of LSC2 would advance such that the next enabled message is *sleep*. The USER could then execute *sleep* and violate the requirements. This illustrates the inability of smart play-out to look ahead into the future by more than one superstep.

Using synthesis, it is possible for the controller to decisively choose an alternative sequence that would not allow the environment to violate the requirements. In response to the message *wake*, the synthesis algorithm would have removed the transitions that permit the sequence *make*, *fall* to occur, leaving *fall*, *make* as the only existing path.

5 Game Structures

A *game structure* (GS) is defined by $G : \langle V, X, Y, \Theta, \rho_e, \rho_s, \varphi \rangle$ consisting of:

- V, a finite set of typed *state variables*. We define s to be an interpretation of V, assigning to each variable $v \in V$ a value $s[v] \in D_v$ within its respective domain. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to expressions over V in the usual way. An *assertion* is a Boolean formula over V. A state s *satisfies* an assertion φ, denoted $s \models \varphi$, if $s[\varphi] = \mathrm{T}$. We say that s is a φ-state if $s \models \varphi$.
- $X \subseteq V$ is a set of *input variables* controlled by the environment. Let $\bar{X}$ denote the set of all input variable valuations.
- $Y = V \setminus X$ is a set of *output variables* controlled by the system. Let $\bar{Y}$ denote the set of all output variable valuations.
- Θ is the initial condition characterizing all initial states of G.
- $\rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the environment. This is an assertion relating state $s \in \Sigma$ to a possible input value $\vec{x'} \in \bar{X}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_e identifies valuation $\vec{x'}$ as a possible *input* in state s, if for some output $\vec{y'}$, $(s, \vec{x'}, \vec{y'}) \models \rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$. where $(s, \vec{x'}, \vec{y'})$ denotes a transition from state s to state $(\vec{x'}, \vec{y'})$.
- $\rho_s(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the system. This is an assertion relating state $s \in \Sigma$ to a possible output value $\vec{y'} \in \bar{Y}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_s identifies valuation $\vec{y'}$ as a possible *output* in state s, if for some input $\vec{x'}$, $(s, \vec{x'}, \vec{y'}) \models \rho_s(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$.
- φ is the winning condition, given by an LTL (linear temporal logic) formula.

As can be seen, changes in state are characterized by changes in the variable valuations. We partition *variables* into those controlled by the environment (input variables) and those by the system (output variables). Each player may then observe and modify the valuations of its own variables, but can only observe the valuations of the opponent's.

5.1 Dependent vs. Independent Moves

We say that a player *moves* from a state whenever it modifies the variable valuations according to its transition relation. In most game settings, including [24], players strictly alternate between moving: a predesignated player moves first according to the current valuation of the input and output variables. The second player then observes the same valuations as the first and also the first player's move and then moves herself.

A different approach is presented in [6], whereby both players move simultaneously and independently. That is, they move at the same instant and both moves are a function of the current variable valuations only—a player can't observe the opponent's move. According to our approach, the players move simultaneously as before, but both players are permitted to move dependently or independently. A player's move is *dependent* if it is a function of the current variable valuations *and* the opponent's move. If it is a function of only the current variable valuations, then it is an *independent* move, as above.

We adopt this approach because LSCs inherently require the system and environment to synchronize during certain points during the execution. This happens, for example, when system and environment instances arrive on an LSC condition. Our definition lends itself to modeling this type of behavior quite naturally. Although it is possible to simulate this type of synchronous behavior using the alternating-turn approach, extra memory and logic seems to be required.

To illustrate the above concepts, consider the following SMV [22] code:

```
 1  next(env) := case
 2    sys=0 & env=0 & next(sys)= 1 : 2;
 3    sys=0 & env=0 : 3;
 4    1 : env;
 5  esac;
 6
 7  next(sys) := case
 8    sys=0 & env=0 : {1,2};
 9    1 : sys;
10  esac;
```

Listing 1.1. Example SMV Code

The example depicts the transition relation for environment input variable `env` between lines 1-5, and system output variable `sys` on lines 7-10. Elsewhere, variable `sys` is defined to range over $0,\ldots,2$ and `env` over $0,\ldots,3$.

The transition relation for each variable is expressed by a case statement. Each line of the case statement takes the form *expr : val* where *expr* is an expression over the variables and *val* is a legal next value (or set of values) when *expr* is true. Each line is evaluated in the order appearing in the input. If *expr* does not hold, then the next line is evaluated and so on, until one of the expressions holds. Expression "1" is a catch-all expression referring to all cases not covered by the expressions appearing above it.

For example, according to line 8, if `sys` and `env` are both 0 in the current state, then `sys` can nondeterministically choose between 1 or 2 in the next state. Line 9 states that the value of `sys` remains unchanged for all other cases. Lines 8 and 9 are examples of independent moves, since neither relies on the environment's move (the value of `env` in the next state.)

As for the transition relation of `env`, line 2 states that if `sys` and `env` are 0 and, furthermore, `sys` is 1 in the next state, then `env` is 2 in the next state. Line 2 is an example of a dependent move, since the case only holds with the cooperation of the system. Line 3 is also dependent since it implies that `next(sys)` is not equal to 1. However, line 4 is independent because it does not depend on any particular value of `next(env)`.

5.2 Deadlock

Conceptually, each round of play proceeds as follows: from a given state, both players each choose among any available move which is legal according to their own transition relation. If both moves are independent then neither player risks interference from their opponent. If a player chooses a dependent move, then the set of allowable moves becomes restricted according to the opponent's move. Both players may also choose dependent moves. However, when at least one of the moves is dependent, there exists a possibility that moves which were legal according to each player's own transition relation may no longer be legal once combined. Such moves are said to be *deadlocked*.

To illustrate deadlock in this context, consider the following contrived example:

```
1  next(sys) := case
2    sys=0 & env=0 & next(env) = 1 : 1;
3    1 : sys;
4  esac;
5
6  next(env) := case
7    sys=0 & env=0 & next(sys)= 1 : 0;
8    1 : env;
9  esac;
```

Listing 1.2. Deadlock Example

First note that lines 2 and 7 both refer to dependent transitions, since the transition relation for each player's variable depends on the opponent's move. Now consider the state where `sys=env=0` holds. According to line 2, if `env` is 1 in the next state, then `sys` must be 1 in the next state. However, according to line 7, if `sys` is 1 in the next state, then `env` must be 0 in the next state. The system's move on line 2 and the environment's move on line 7 are deadlocked because there will never be a way to proceed using this combination. Note that there could exist other moves that do work. For instance, both players may move from `sys=env=0` to state `sys=env=0`.

Although not shown in this example, there could generally exist states from which all moves are deadlocked, leaving no possible next move. We refer to these states as *fully deadlocked*.

The presence of deadlocks, or even fully deadlocked states, in a transition system is not necessarily forbidden. For example, one may intentionally introduce deadlocks into a transition system to model some kind of real life dead-end situation, with the idea of having synthesis generate the strategy to avoid the deadlocks. In contrast with previous synthesis work based on the turn-based approach, such as [24], additional consideration is required for handling (fully) deadlocked states in the case of games with simultaneous transitions.

6 Synthesis

Let $\mathcal{G}$ be a game structure and s and s' be states of $\mathcal{G}$. We say s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We freely switch between $(s, \vec{x'}, \vec{y'}) \models \rho_e$ and $\rho_e(s, \vec{x'}, \vec{y'}) = 1$ and similarly for ρ_s.

A *play* σ of $\mathcal{G}$ is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* ($s_0 \models \Theta$) and *consecution* (for each $i \geq 0, s_{i+1}$ is a successor of s_i). Let σ be a play of $\mathcal{G}$. From state s, the environment chooses an input $\vec{x}' \in X$ and system chooses an output $\vec{y}'$ such that $\rho_e(s, \vec{x}', \vec{y}') = 1$ and $\rho_s(s, \vec{x}', \vec{y}') = 1$.

We say that play σ is *winning for the system* if it is infinite and satisfies the winning condition φ. Otherwise, σ is *winning for the environment*.

Let $\sigma = s_0, \ldots, s_n$. A *strategy* for the system is a function $f : \Sigma^+ \times \bar{X} \mapsto \bar{Y}$ where for every $\vec{x}' \in \bar{X}$ such that $\rho_e(s_n, \vec{x}', f(\sigma, \vec{x}')) = 1$, we have $\rho_s(s_n, \vec{x}', f(\sigma, \vec{x}')) = 1$. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \ldots, s_i, s_{i+1}[\bar{X}]) = s_{i+1}[\bar{Y}]$, where $s_{i+1}[\bar{X}]$ and $s_{i+1}[\bar{Y}]$ are the restrictions of s_{i+1} to variable sets X and Y, respectively.

Strategy f is winning for the system from state $s \in \Sigma$ if all s-plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_c the set of states from which there is a winning strategy for the system. $\mathcal{G}$ is said to be winning for the system if all initial states of $\mathcal{G}$ are winning for the system. In this case, we say $\mathcal{G}$ is *realizable* and we *synthesize* a winning strategy which is a working implementation for the system. Otherwise $\mathcal{G}$ is *unrealizable*.

6.1 Controllable Predecessors

States from which the system can force the game into p are referred to as *controllable predecessors* of p, denoted $\bigcirc\!\!\!\!\diamond\, p$, where p is an assertion over the state space $(\bar{X}, \bar{Y})$. The main idea is that the system, from a controllable predecessor of p, can choose a move for which all remaining environment moves lead to p—or—for each possible environment move, can choose at least one move leading to p. That is, the system can take either an independent or dependent move. Our controllable predecessor formula is a disjunction of two parts, Φ_1 and Φ_2. We have:

$$\Phi_1 = \exists \vec{y}' [[\exists \vec{x}' \rho] \wedge [\forall \vec{x}' \rho_e \rightarrow [\rho_s \wedge (\vec{x}', \vec{y}') \in \|p\|]]]$$

where $\|p\|$ denotes the set of states characterized by assertion p and $\rho = \rho_e \wedge \rho_s$ is the set of joint moves. Formula Φ_1 states that for some system move $\vec{y}'$, any legal environment move $\vec{x}'$ must lead to p. The left side of the conjunction assures the absence of fully deadlocked predecessors. Next we have:

$$\Phi_2 = [\exists \vec{x}' \exists \vec{y}' \rho] \wedge \forall \vec{x}' [[\forall \vec{y}' \neg \rho_e] \ \vee \ [\exists \vec{y}' \rho \wedge (\vec{x}', \vec{y}') \in \|p\|]]$$

The right side requires that for every environment input $\vec{x}'$, either there are no environment moves available or there must exist some system move $\vec{y}'$ leading to p. The left side of the conjunction assures the absence of fully deadlocked predecessors.

Putting it all together, we compute the set of controllable predecessors of p with:

$$\|\bigcirc\!\!\!\!\diamond\, p\| = \{s \mid \Phi_1 \vee \Phi_2\}$$

6.2 Realizability and Winning Strategy

Once the notion of controllable predecessor is in place, the decision procedure for realizability and the extraction of the winning strategy proceeds according to [24], which focuses on winning conditions which are recurrence properties, i.e., LTL formulas of the form $\square \diamond q$ for an assertion q. We restrict our attention to formulas of this form for the purposes of this paper.

A state satisfies $\bigcirc\!\!\!\!\diamond\, p$ (for some assertion p) if the system can force the environment to reach a p-state in a single step. Based on this pre-image operator, a set of *winning states* is computed according to the following fix-point equation:

$$W_c = \nu Z \mu Y. \bigcirc\!\!\!\!\diamond\, Y \vee q \wedge \bigcirc\!\!\!\!\diamond\, Z \tag{1}$$

Given a game structure $\mathcal{G}$, we can check realizability of G by testing $\overline{W_c} \cap \Theta = \emptyset$. If $\mathcal{G}$ is winning for the system, a winning strategy is extracted by removing controllable transitions which lead to states outside of W_c.

7 Main Result

We now present a method for constructing a game structure from LSC requirements. Some of the LSC logic necessary for this result is already incorporated into smart play-out: whenever the user injects an input event, smart play-out constructs an LSC transition system. We avoid redundancy here by focusing most of our attention on the extensions necessary for synthesis. The interested reader can consult [9] for the specifics of the smart play-out construction.

On a high level, smart play-out defines one SMV module for every object in the requirements and composes the modules asynchronously for program executions and model-checking. In contrast, the synthesis algorithm of [24] requires precisely two transition systems—one for the system and one for the environment. One of our goals is therefore to express the collection of asynchronous transition systems as a game structure. Secondly, we add additional logic over and above that supplied by smart play-out which is necessary for synthesis. We begin by first introducing the variables used in our construction and then describe the transition relation for each.

7.1 Variables

Let $\mathcal{O}$ be an object system and let $\mathcal{LR} = L_1, \ldots, L_n$ over $\mathcal{O}$ be a set of LSC requirements. We construct a game structure $\mathcal{G}$ with a set of *input variables* belonging to the environment and *output variables* belonging to the system. We now specify the set of variables V by defining the input variables X and output variables Y. The input variables are as follows:

1. act_{L_i} is 1 when the main chart of LSC L_i is active, and 0 otherwise.
2. $msg^s_{O_j \to O_k}$ denotes the sending of a message from object O_j to object O_k in which $O_j.own = env$ (O_j belongs to the environment.) The value is set to 1 at the occurrence of the send and is changed to 0 at the next state.
3. $msg^r_{O_j \to O_k}$ denotes the receipt of a message by object O_k from object O_j in which $O_k.own = env$. As in the case of sending, the value is 1 at the instant the message is received and changes to 0 in the next state.

4. l_{L_i,O_j} is the location of object O_j in the main chart of LSC L_i where $O_j.own = env$. The location number ranges over $0,\ldots,l^{max}$ where l^{max} is the last location of O_j in the main chart of LSC L_i. This variable is meaningful only when act_{L_i} is 1.
5. $l_{pch(L_i),O_j}$ is the location of object O_j in the prechart of LSC L_i where $O_j.own = env$. Its value ranges over $0,\ldots,l^{max}$ where l^{max} is the last location of O_j in the prechart of LSC L_i. This variable is meaningful only when act_{L_i} is 0.
6. $gbuchi$ is an auxiliary variable used to reduce a Generalized Büchi winning condition to a Büchi winning condition.
7. $envreq$ is a variable that determines which of the environment's objects has control in the next step.

The output variables belonging to the system are given by:

1. $msg^s_{O_j \rightarrow O_k}$ denoting the sending of a message from object O_j to object O_k in which $O_j.own = sys$ (O_j belongs to the system.)
2. $msg^r_{O_j \rightarrow O_k}$ denoting the receipt of a message by object O_k from object O_j in which $O_k.own = sys$.
3. l_{L_i,O_j} is the location of object O_j in the main chart of LSC L_i such that $O_j.own = sys$.
4. $l_{pch(L_i),O_j}$ is the location of object O_j in the prechart of LSC L_i such that $O_j.own = sys$.
5. $currobj$ is a number ranging over $1,\ldots,|\mathcal{O}|$, referring to the object $O_{currobj}$ that currently has control of the execution.

The active flags, (act_{L_i}, for all i) and the auxiliary variable $gbuchi$ are not properties of the environment specifically, although they are environment variables. These are examples of *bookkeeping variables*, whose values are a function of the variables of both players. The choice of ownership could therefore be arbitrary. However, we assign ownership of these variables to the environment in order to be conservatively safe.

For example, if there exists a subtle error in the transition relation of any of these variables, the environment would find a way to utilize the error to its advantage in order to win the game and deem the requirements unrealizable. This is positive because we are forced to deal with the error in such a case. We could have alternatively chosen the system as the owner instead, in which case an error in the definitions could lead to false realizability—a more dangerous situation, particularly in the case of safety critical systems.

The purpose of the remaining variables, $envreq$ and $currobj$ are explained below.

7.2 Transitions

Smart play-out constructs a transition system comprised of an asynchronous composition of SMV modules. Generally speaking, each module defines the behaviors of one object in the LSC requirements, consisting of a set of variables and a transition relation. When generating traces, the TLV-BASIC [26] model-checking routine arbitrarily selects modules for execution one at a time. The corresponding variables are then updated according to the transition relation of the selected module. Intuitively, each module's (i.e., object's) transition relation permits the object to carry out the next behavior on the object's instance line, with respect to the object's present LSC location.

On the other hand, the current synthesis implementation requires a game structure in which all objects (and associated transition relations) belonging to the system are grouped into a single system module, and likewise for the environment. This raises the question of how to deal with the multiple definitions for each variable. We now describe the solution.

Let φ_i be any variable belonging to object O_i in the smart play-out construction. The transition relation, according to [9], for φ_i takes on the form:

$$\varphi_i' = \begin{cases} c_1^i & \text{if } \Omega_1^i \\ \vdots & \vdots \\ c_n^i & \text{if } \Omega_n^i \end{cases}$$

where c_j^i is a constant, Ω_j^i is a conditional expression over the variables of all objects in $\mathcal{O}$, and n is the number of SMV transition relation cases produced by smart play-out for φ_i. In our synthesis construction, we have:

$$\varphi' = \begin{cases} \varphi_1' & \text{if } \mathit{currobj} = 1 \\ \vdots & \vdots \\ \varphi_k' & \text{if } \mathit{currobj} = k \end{cases}$$

where k is the number of objects. Therefore, we may simulate the asynchronous behavior of the smart play-out transition system by manipulating the variable $\mathit{currobj}$. This variable is responsible for determining which object, among the system and environment objects, move in the next step.

Variable $\mathit{currobj}$ must necessarily be owned by either the system or the environment. It would seem that permitting just one player to determine the current objects for both itself and its opponent could result in an unfair advantage. To level the playing field according to our result, the system and environment choose among their respective objects, but the decision of when each player gets their turn to decide goes to the system. To prevent the system from starving the environment of any opportunity to move, we will require the system to yield control to the environment infinitely often.

Formally, let $O_1, \ldots, O_j$ be the set of objects belonging to the environment and $O_{j+1}, \ldots, O_k$ be those of the system. We let:

$$\mathit{envreq}' \in \{1, \ldots j\}$$

The environment uses envreq to non-deterministically choose which of its objects will be the next to move once given a turn. With this in place, the system selects the current object in the following way:

$$\mathit{currobj}' \in \{\mathit{envreq}', j+1, \ldots, k\}$$

Note that this permits the system to execute arbitrarily long supersteps, since it can just keep selecting values between $j+1, \ldots, k$. However, the winning condition discussed in the next section will require that $\mathit{currobj}' \leq j$ infinitely often, causing all supersteps to be finite.

7.3 Initial and Winning Conditions

The initial condition, Θ, of our game structure is the set of states in which $gbuchi = 0$, $act_{L_i} = 0$ for all i, all message variables are set to 0, and all location variables are set to 0. The initial value of $envreq$ is not specified in Θ, so the choice is therefore non-deterministic. The winning condition φ is the generalized Büchi LTL formula:

$$\Box \Diamond \bigwedge_{i=1}^{n} act_{L_i} = 0 \wedge \Box \Diamond \; currobj \leq j$$

which is equivalent to:

$$\Box \Diamond \; gbuchi = 0$$

where:

$$gbuchi' = \begin{cases} 1 \text{ if } gbuchi = 0 \\ 2 \text{ if } gbuchi = 1 \wedge \bigwedge_{i=1}^{n} act_{L_i} = 0 \\ 0 \text{ if } gbuchi = 2 \;\wedge\; currobj \leq j \end{cases}$$

The above winning condition ensures that a stable state—where all main charts are simultaneously inactive—is visited infinitely often and that all supersteps are finite. It assumes that no environment messages appear in a main chart. For this, a more expressive winning condition beyond the scope of this paper is necessary.

7.4 Synthesis in the Play-Engine

When a Play-Engine user creates LSC requirements and wishes to perform synthesis, the following steps occur: first, the LSC requirements are translated into a game structure according to the techniques of this section. Next, the synthesis algorithm described in subsection 6.2 is executed. If the algorithm yields an unrealizable outcome, the process terminates at this point and the user is notified. Otherwise, a single, synchronous, transition system is constructed. We refer the interested reader to [24] for more details on this construction, which we do not describe in this paper.

At this point, the Play-Engine user may act in the role of the environment by injecting environment inputs and observing system responses, in a manner nearly identical to smart play-out. Upon each input event, a model checking routine is executed on the above output transition system. Since the winning condition guarantees that all LSCs will infinitely often be simultaneously inactive for any realizable LSC requirements, it is therefore also guaranteed that a valid super-step will exist for every reachable stable state in the output transition system.

Note that while the model-checking procedure is executed each time the user injects an input, the synthesis need only run once. Moreover, LSC requirements and the synthesis algorithm need not exist on the same computer or platform as the application to be deployed, since the only deliverable is the output transition system.

8 Conclusion

In this paper, we introduced a method for overcoming the limitations of smart play-out by performing a complete analysis of the state space. We first described a modification to the previous turn-based approaches for synthesis which permits players to transition simultaneously in a dependent or independent fashion. We then showed how to construct a game structure that expresses the behaviors of LSC requirements as a two-player game between the reactive system and its environment. After invoking the synthesis routine, the end result is a controller—a transition system—which consists only of transitions that collectively satisfy the LSC requirements, provided a satisfying system exists. The controller, which encodes the winning strategy, can be used for executing supersteps that satisfy the requirements.

We describe an implementation of the foregoing synthesis procedure as an extension to the Play-Engine's smart play-out feature. With this implementation, the user first plays in behavioral requirements, as before. Then the synthesis procedure may be invoked from the Play-Engine's user interface, which constructs the game structure, checks realizability, and extracts a controller if the requirements are realizable. The synthesis algorithm executes once, yielding a controller, from which supersteps may be extracted using a superstep extraction process similar to that already present in smart play-out.

We are currently implementing a new Scenario-Based Tool [1] with a special focus on scenario-based modeling of biological systems [13]. Consistency checking and synthesis are important capabilities required for biological modeling, thus we are implementing extensions and variants of the work described here. An experimental implementation of a new compositional synthesis algorithm was already implemented using this new tool [19]. Independently of any specific tool or application domain, however, we wish to place our current focus on a broader solution of synthesizing executable programs from scenario-based requirements, whereby the controller generated by the synthesis routine can be used to directly execute a general reactive system.

References

1. Microsoft Research Cambridge, Scenario-Based Tool for Biological Modeling (2009), `http://research.microsoft.com/SBT/`
2. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable concurrent program specifications. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
3. Bontemps, Y., Heymans, P., Schobbens, P.Y.: From live sequence charts to state machines and back: A guided tour. IEEE Trans. Software Eng. 31(12), 999–1014 (2005)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001); preliminary version appeared in: Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1999)
5. Damm, W., Toben, T., Westphal, B.: On the Expressive Power of Live Sequence Charts. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 225–246. Springer, Heidelberg (2007)
6. de Alfaro, L., Henzinger, T., Majumdar, R.: From verification to control: dynamic programs for omega-regular objectives. In: Proc. 16th IEEE Symp. Logic in Comp. Sci., pp. 279–290. IEEE Computer Society Press, Los Alamitos (2001)
7. Harel, D., Kantor, A., Maoz, S.: On the Power of Play-Out for Scenario-Based Programs. Technical report, Weizmann Institute (2009)
8. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Int. J. of Foundations of Computer Science (IJFCS) 13(1), 5–51 (2002); also in: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 1–51. Springer, Heidelberg (2001)

9. 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); also available as Tech. Report MCS02-08, The Weizmann Institute of Science
10. Harel, D., Kugler, H., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenarios-Based Requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
11. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
12. Hennicker, R., Knapp, A.: Activity-Driven Synthesis of State Machines. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 87–101. Springer, Heidelberg (2007)
13. Kam, N., Kugler, H., Marelly, R., Appleby, L., Fisher, J., Pnueli, A., Harel, D., Stern, M., Hubbard, E.: A scenario-based approach to modeling development: A prototype model of C. elegans vulval fate specification. Developmental Biology 323(1), 1–5 (2008)
14. Klose, J., Wittke, H.: An automata based interpretation of live sequence chart. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, p. 512. Springer, Heidelberg (2001)
15. Koskimies, K., Makinen, E.: Automatic synthesis of state machines from trace diagrams. Software — Practice and Experience 24(7), 643–658 (1994)
16. Koskimies, K., Mannisto, T., Systa, T., Tuomi, J.: SCED: A Tool for Dynamic Modeling of Object Systems. Tech. Report A-1996-4, University of Tampere (July 1996)
17. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: Proc. Int. Workshop on Distributed and Parallel Embedded Systems (DIPES 1998), pp. 61–71. Kluwer Academic Publishers, Dordrecht (1999)
18. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-Based Specifications. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 445–460. Springer, Heidelberg (2005)
19. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Proc. 15^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009). LNCS. Springer, Heidelberg (2009)
20. Leue, S., Mehrmann, L., Rezai, M.: Synthesizing ROOM models from message sequence chart specifications. Tech. Report 98-06, University of Waterloo (April 1998)
21. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2006), pp. 5–12 (2006)
22. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers, Boston (1993)
23. 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)
24. Pnueli, A.: Extracting controllers for timed automata. Technical report, New York University (2005)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. Princ. of Prog. Lang., pp. 179–190 (1989)
26. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 184–195. Springer, Heidelberg (1996)
27. Sun, J., Dong, J.S.: Synthesis of distributed processes from scenario-based specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 415–431. Springer, Heidelberg (2005)

28. Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM Trans. Software Engin. Methods 13(1), 37–85 (2004)
29. Vardi, M.: An automata-theoretic approach to fair realizability and synthesis. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 267–278. Springer, Heidelberg (1995)
30. Whittle, J., Saboo, J., Kwan, R.: From scenarios to code: an air traffic control case study. In: 25th International Conference on Software Engineering (ICSE 2003), pp. 490–495. IEEE Computer Society, Los Alamitos (2003)
31. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: 22nd International Conference on Software Engineering (ICSE 2000), pp. 314–323. ACM Press, New York (2000)