

Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications ^{*}

Hillel Kugler¹ and Itai Segall^{2**}

¹ Computational Biology Group, Microsoft Research, Cambridge, UK
hkugler@microsoft.com

² Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
itai.segall@weizmann.ac.il

Abstract. Synthesis is the process of automatically generating a correct running system from its specification. In this paper, we suggest a translation of a Live Sequence Chart specification into a two-player game for the purpose of synthesis. We use this representation for synthesizing a reactive system, and introduce a novel algorithm for composing two such systems for two subsets of a specification. Even though this algorithm may fail to compose the systems, or to prove the joint specification to be inconsistent, we present some promising results for which the composition algorithm does succeed and saves significant running time. We also discuss options for extending the algorithm into a sound and complete one.

1 Introduction

Automatic synthesis of systems directly from their specification has been a dream for many researchers. In the dream, a specifier is faced with an expressive yet intuitive specification language, in which she specifies the requirements from her system. All the rest will then happen automatically – by clicking a button, the specification will automatically be checked for consistency, and if found consistent, a system that is correct-by-construction will be generated, i.e., a system that is guaranteed to satisfy the specification. On the way towards realizing this dream, one must first choose a specification language that is both expressive and intuitive, and then build strong and fast algorithms for synthesizing systems from this language. Synthesis raises major challenges in terms of the inherent complexity of the problem and the required methodological development approach. Compositional synthesis, in which two synthesized systems may easily be composed into one large system, may help in addressing these challenges. By synthesizing small parts of the specification separately, and composing the intermediate results, one may save significant running time. Moreover, specifications

^{*} The research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

^{**} This work was carried out during this author's internship at Microsoft Research, Cambridge, UK.

are usually constructed by a team and evolve over time, by introducing more requirements and modifying existing ones. A compositional approach to synthesis supports such evolution by reusing results for existing parts of the specification, rather than having to synthesize a complete system whenever the specification is modified. Another common way of tackling the high complexity of synthesis is by introducing semi-automatic algorithms. Such algorithms require some interaction with the user, but may perform much better by exploiting the user’s understanding of the system. In a compositional synthesis algorithm this can be done by leaving the choice of the parts that should be separately synthesized to the user.

Live Sequence Charts (LSCs) [8] have been introduced as a highly expressive extension of Message Sequence Charts [18]. LSCs are multi-modal charts that distinguish between behaviors that may happen (existential, cold) and those that must happen (universal, hot). LSCs are highly expressive, and different translations of LSCs into temporal logic have been suggested (see, e.g., [9, 21]). On the other side, being visual in nature, we believe the language is highly intuitive. Thus, LSCs were suggested as an expressive and intuitive specification language to use for synthesis in [13]. Despite research efforts on synthesizing systems from LSCs, e.g., [15, 5], practical application to real-world systems has not yet been achieved.

In this paper, we propose a representation of LSC specifications as two-player game structures, in which a winning strategy for the system is equivalent to a reactive system satisfying the requirements. This representation is then synthesized into a reactive system using an approach similar to that proposed in [28, 29]. We further propose a method for composing two synthesized systems. This method consists of an algorithm that is sound but not complete, and an algorithm that is complete but not sound. Therefore, it might fail to compose systems, or to prove their specifications to be inconsistent. However, we do provide several test cases for which it does succeed in creating a system for the entire specification, or prove the entire specification to be inconsistent, in running time significantly faster than that of non-compositional synthesis. We also briefly describe an extension of the approach that is sound and complete. This extension may be problematic in terms of running time and implementation, therefore it is given in this paper mainly for completeness of the approach, rather than as a full replacement for synthesis of composite specifications.

This work focuses on a subset of LSCs that includes only messages, and assumes that main charts include only messages controlled by the system. We also assume that no LSC has multiple copies simultaneously open during runtime. Finally, all messages in the specification are assumed to be synchronous, i.e., the event of sending a message and receiving it are simultaneous.

We implemented the approach introduced here as part of the new Scenario-Based Tool [33] developed at Microsoft Research Cambridge, using TLV [31] for the symbolic computations.

Some details of implementation, proofs, and notations are omitted from this version of the paper due to lack of space. See [23] for more details.

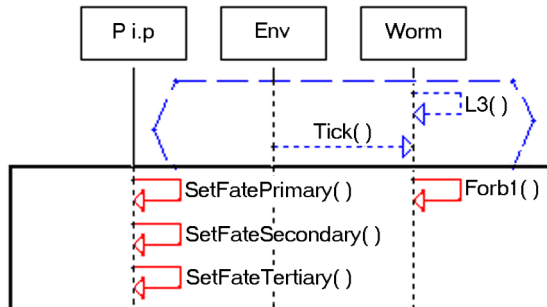


Fig. 1. An example LSC.

2 Preliminaries

2.1 Live sequence charts

Live sequence charts (LSCs) [8] are an extension of message sequence charts (MSCs) [18]. LSCs, like MSCs, contain vertical lines, termed *lifelines*, which denote objects, and *events*, involving one or more lifelines. The most basic construct of the language are *messages*: a message is denoted by an arrow between two lifelines (or from a lifeline to itself), representing the event of the source object sending a message to the target object. A typical LSC consists of a prechart (denoted by a blue dashed hexagon), and a main chart (denoted by a solid frame). The intended semantics is that whenever the prechart is satisfied in a run of the system, eventually the main chart must also be satisfied. The synthesis method presented here focuses on messages, and does not currently support any of the more advanced constructs of the language, such as conditions, loops, etc.

Any object taking part in the specification is either controlled by the system, or by the environment. A message is said to be a *system (environment) message*, i.e., controlled by the system (environment), if it is sent from an object controlled by the system (environment).

An example of an LSC appears in Fig. 1. The LSC refers to three objects, `Env` representing the environment, and `Worm` and `Pi.p` representing two system objects. The LSC states that whenever `Worm` sends the message `L3` to itself, and then `Env` sends `Tick` to `Worm` (in this order), then `Pi.p` should send itself the messages `SetFatePrimary`, `SetFateSecondary` and `SetFateTertiary` (in this order), and `Worm` should send itself the message `Forb1`. Note that in the main chart there is no explicit order between the `Pi.p` messages and that of the `Worm`. Also note that if one of the main chart messages occurs before the prechart ends (e.g., after `L3` is sent, but before `Tick`), then the prechart is cold-violated and gracefully closed. This is considered legal behavior. If, however, the prechart completes, then the main chart messages must be sent in the correct order. Otherwise, this is considered a violation of the specification.

An operational semantics and an execution technique termed *play-out* was defined for the LSC language in [16]. Play-out remembers at each point in time

for each LSC the current *cut* (intuitively, a marker of what has already happened, and what not). It also maintains the set of *active* LSCs (those for which the prechart has been satisfied, but not the main chart). The set of all LSCs and their cuts is termed a *configuration*. At each step, the play-out mechanism chooses one message that is *enabled* in some active LSC (i.e., appears right after the current cut), and not violating in any others (a message is violating if it appears in an active chart but is not enabled), and executes it. Stronger mechanisms, termed *smart play-out* and *planned play-out* are introduced in [14, 17]. These are initiated following each environment step, and look for a sequence of system events to perform in response (termed *superstep*), in order to drive the system to a stable state (one in which no LSC is active). However, looking only one superstep, or a finite number of supersteps, ahead, is not sufficient either. An example for this is given in [12]. This leads to the synthesis problem, i.e., given an LSC specification, finding a reactive system that adheres to the specification, or proving one does not exist.

2.2 Game structures and strategies

We view the synthesis problem as a two-player game between the system and environment, as formulated in a game structure. We modify the game structure and strategy definitions from [29] to reflect games in which the system is the first player. Intuitively, a game structure is a tuple $G : \langle V, X, Y, \Theta, \rho_s, \rho_e, \varphi \rangle$, where V represents the set of *state variables*, X is the set of system-controlled variables, and Y is the set of environment-controlled variables. Θ is the initial condition. ρ_s and ρ_e represent the transition relations of the system and environment, resp. The transition relation of the system depends only on the current state, whereas that of the environment may depend also on the system's transition. Finally, φ is the winning condition, of the form $\varphi = \Box \Diamond q$, where q is a state formula.

A *strategy* is a partial function mapping a series of states to a set of possible system actions. A run is *compliant* with a strategy if each step taken by the system is one allowed by the strategy. A strategy is *winning for the system* if any run, in which the environment takes only legal steps (i.e., ones allowed by the game definition), and is compliant with the strategy, is winning for the system, i.e., satisfies φ . Finally, a game structure is *realizable* if there exists a strategy that is winning for the system from any initial state (one satisfying Θ).

3 The Synthesis Problem

The synthesis problem is defined as follows. Given an LSC specification, determine whether there exists a reactive system that satisfies the specification, and generate one if so. Such a system will be called a *synthesized system*. This synthesized system must fulfill two requirements: infinitely often it must listen for environment events, and, it must never violate the specification. Note that violation here refers both to explicit violations of the requirements (safety), and to cases in which a step that must happen never does (liveness).

Two major distinct views can be taken when considering synthesis from LSCs. According to the first view, inspired by [16], the synthesized system is a direct execution engine for the specification, that never violates it. According to the second view, adopted in this work, the system need not execute the specification directly – it may take any action, as long as the two requirements above hold. We will refer to the problem addressed in this paper as *synthesis*, and to the other interpretation as *non-violating execution*.

In practice, the difference between the two translates to the choice of steps from a given state. In synthesis, the system may perform any step it deems necessary, while in non-violating execution, a message may be sent only if it is enabled in some active main chart, as defined in the operational semantics [16].

Our interpretation of the synthesis problem treats the specification as more under-specified – the specifier states things that may and must happen in the system, but anything unconstrained may also happen. In non-violating execution, an event may happen only if explicitly specified so. Note that in non-violating execution, a specification may be unrealizable, but become realizable by adding another LSC (or set of LSCs). In synthesis, however, specifications are monotonic. If a specification is unrealizable, then so is every extension thereof, and vice versa, a synthesized system for a specification may also serve as a synthesized system for any subset of it. This monotonicity gives rise to the issue of composition – given synthesized systems for two specifications, find a system for the unified specification.

Non-violating execution may seem more appropriate for finalized specifications. However, for intermediate stages the specification is usually more under-specified, and the specifier does not want to restrict execution to those steps explicitly appearing in it. Therefore, for such specifications, synthesis is more appropriate. Moreover, synthesis of intermediate specifications may aid the specifier in identifying under-specified parts in the specification, and extending it accordingly. For the final specification, the choice between non-violating execution and synthesis depends on the amount of detail the specifier has introduced, and the level of under-specification in it. Even for the final specification the specifier may choose to leave certain parts under-specified and decide to use synthesis.

4 The Representation

4.1 The LSC game structure

As mentioned above, this work focuses on a subset of LSCs that includes only messages, and assumes that main charts include only system messages. We also assume that no LSC has multiple copies simultaneously open during runtime. Finally, all messages in the specification are assumed to be synchronous, i.e., the event of sending a message and receiving it are simultaneous.

Given an LSC specification, we construct a game structure G . Intuitively, the system controls a single variable, m_s , that represents the message sent by a system object in this step. The environment controls a variable m_e representing

the message sent by an environment object in this step, and a set of variables L that represent the current LSC configuration. In every turn, the system sends a single message or chooses not to (by using the special symbol \perp for m'_s). The environment may choose an environment message to send only if the system did not send one of its own (i.e., if m'_s is \perp). The environment may also choose not to send any message (by using its own \perp symbol for m'_e). The domain of m_e includes one additional special symbol, ∞ (may also be used only when m'_s is \perp). By using this symbol, the environment may force the game to stay forever in the current state. This forces the system to pass control back to the environment only at the end of a superstep, and is crucial for the correctness of the composition. Updating the configuration is deterministic, given m'_s and m'_e , and follows the semantics defined in [16] directly.

We adopt the superstep approach from [14]. Following a single environment step, the system may perform as many steps as it wishes (finitely many) in order to reach a state in which all LSCs are not active. Only then will the environment be allowed to play again.

More formally, given an LSC specification, we construct a game structure $G : \langle V, X, Y, \Theta, \rho_s, \rho_e, \varphi \rangle$ as follows:

- The set of variables $V = \{m_s, m_e\} \cup L$, as follows:
 - m_s represents the system message sent in this step. Its domain is the set of all messages sent by system objects in the specification, plus the symbol \perp , representing a no-op.
 - m_e represents the environment message sent in this step. Its domain is the set of all messages sent by environment objects in the specification, plus the symbols \perp and ∞ , both representing no-ops.
 - L contains the following:
 - * For each lifeline i , a variable loc_i representing the location of the cut on lifeline i . Each location variable ranges over $0, \dots, l^{max}$, where l^{max} is the last location of lifeline i .
 - * For each LSC l , boolean variables $active_l$ and $hotViolated_l$ representing whether the LSC is active, and whether it was ever hot violated, respectively. We also introduce another boolean variable $prev_l$ for each LSC l , that represents the fact that in the previous timestep $active_l$ and $hotViolated_l$ were both false.
- $X = \{m_s\}$ is the only system variable.
- $Y = V \setminus X = \{m_e\} \cup L$ are the environment variables.
- The system transition relation is defined such that

$$\rho_s(m_s, m_e, L, m'_s) = 1 \iff [(m_s = \perp \wedge m_e = \infty \rightarrow m'_s = \perp)]$$
- The environment transition relation is defined such that

$$\rho_e(m_s, m_e, L, m'_s, m'_e, L') = 1 \iff [(m'_s \neq \perp \rightarrow m'_e = \perp) \wedge \text{the } L' \text{ variables represent the state of the specification after sending } m'_s \text{ and } m'_e \text{ from state } L].$$
 We omit from this version of the paper the details of updating the L variables, as they are somewhat similar to those of [14], and are a direct translation of the operational semantics defined in [16].

- The winning condition is $\varphi = \Box\Diamond(m_s = \perp \wedge \text{prev}_{l_1} \wedge \dots \wedge \text{prev}_{l_k})$, where l_1, \dots, l_k are the LSCs in the specification. This represents the requirements from the synthesized system, i.e., infinitely often the system must listen to environment events (this happens when $m_s = \perp$), and it must never violate the specification (represented by the requirements on the prev variables in φ , similarly to the requirement for ending a superstep in [14]). We denote by q the state formula in φ .
- The initial condition is $\Theta = [(m_s = m_e = \perp) \wedge \text{values for } L \text{ that represent all LSCs being closed}]$.

For a variable u , we denote by \bar{u} a valuation of u , and similarly for sets of variables.

4.2 Monotonicity

Given a realizable LSC game structure, the game structure corresponding to any subset of the LSCs is also realizable. Moreover, the restriction of a winning state in the composite structure to the subset one is a winning state in it. Intuitively, given a winning strategy for the composite specification, the same strategy can be used for the subset one. Since the strategy is winning for the composite specification, it satisfies the safety and liveness requirements of all LSCs in the entire specification, therefore it satisfies them for the LSCs in the subset one, and is a winning strategy for the subset specification.

Similarly, any extension of an unrealizable LSC game structure (by adding more LSCs) is also unrealizable.

5 The Synthesis Algorithm

We adapt the algorithm from [29] for games in which the system (controller) plays first in each turn. For lack of space, the details of this modification are omitted from this version of the paper. The result of the algorithm is a transition system $S = \langle V, \rho, \Theta \rangle$.

Definition 1. *Given a transition system $S = \langle V, \rho, \Theta \rangle$, the strategy induced by S is defined as: $f(s_0, s_1, \dots, s_t) = \{m'_s \mid \exists m'_e, L' : (s_t, m'_s, m'_e, L') \models \rho\}$, i.e., the strategy allows any system message that appears in transitions from s_t .*

Since the induced strategy considers only the current state (state-strategy), we will use the short notation of $f(V)$.

6 Strategy Composition

Consider LSC game structures for two subset specifications G_1, G_2 . The variables m_s and m_e are the only ones appearing in both. For now, assume the sets of messages appearing in the two specifications are equal, thus the domains of m_s and m_e are also equal. The case where some messages appear only in one of the

specifications is discussed in Section 6.3. Clearly, $\rho_s^1 = \rho_s^2$ since they depend only on m_s, m_e and m'_s . ρ_e is the conjunction of ρ_e^1 and ρ_e^2 (each restricted to the variables relevant to it). q , the state formula in φ , is the conjunction of q_1 and q_2 . Finally, the initial condition is $\Theta = \Theta_1 \wedge \Theta_2$. Define $G = \langle V, X, Y, \rho_s, \rho_e, \varphi, \Theta \rangle$ to be the LSC game structure for the composite specification.

6.1 The composition algorithm

We present an algorithm for the composition of transition systems that induce strategies. The algorithm has two main steps. It first computes the synchronous parallel composition of the transition systems, and then removes bad states from the result. A state is considered bad if it is a dead end, or will necessarily lead to one.

Given a transition system S inducing a strategy for the LSC game structure G , the following assertions, $Rlvt(V, m'_s)$ and $Self(V, m'_s)$, represent whether the message m'_s is relevant from a given state, and whether it leaves the LSC configuration unchanged from it, resp.

$$(s, \bar{m}'_s) \models Rlvt \Leftrightarrow \exists \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho$$

$$(s, \bar{m}'_s) \models Self \Leftrightarrow \bar{m}'_s \neq \perp \wedge [\forall \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho \rightarrow (s[L] = \bar{L}')]]$$

Where $s[L]$ stands for the restriction of s to the variables of L .

Using these assertions, we define the operator *bad predecessor*, denoted $\ominus p$, as follows (where the notation $\|q\|$ stands for the set of states satisfying q):

$$\| \ominus p \| = \{ s \mid \forall \bar{m}'_s [(s, \bar{m}'_s) \models Rlvt] \rightarrow [(s, \bar{m}'_s) \models Self \vee \exists \bar{m}'_e, \bar{L}' ((s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho \wedge (\bar{m}'_s, \bar{m}'_e, \bar{L}') \in \|p\|)] \}$$

Thus, a state satisfies $\ominus p$ if any system message relevant from it is either a self message (i.e., it leaves the configuration unchanged), or opens an opportunity for the environment to get to a state satisfying p . By applying $\ominus p$ iteratively until a fixpoint is reached, we mark all bad states. By initially setting the set of bad states to \emptyset , dead-end states are marked as bad in the first iteration, and in following iterations, all states necessarily leading to them. We therefore define the set *Bad* as: $Bad = \mu B. \ominus B$, i.e., the minimal fixpoint of the bad predecessor predicate.

One can improve the performance of the fixpoint computation by first computing the set of reachable states in the transition system, and considering only those in the fixpoint iterations. Since the size of synthesized systems is typically significantly smaller than that of the whole specification model, the set of reachable states in them may be computed relatively easily.

The pseudo-code of an algorithm for the composition of synthesized systems is given in Fig. 2. The algorithm gets as input two transition systems, and if successful returns a new transition system. It computes the synchronous parallel composition of the two input systems, and removes bad states from it. If an initial state is found to be bad, then the algorithm terminates with no returned system. Otherwise, it constructs a transition relation that makes sure no bad states are ever reached. In the following sections we explore how this algorithm is either sound or complete, depending on the inputs.

```

1: procedure COMPOSE(System S1, System S2)
2:    $S := S_1 || S_2$ 
3:    $bad \leftarrow calc\_bad(S, reachable(S))$ 
4:   if  $\exists s_0 \models \Theta, s_0 \in bad$  then return “Failed”
5:   else  $\rho(s, m'_s, m'_e, L') \leftarrow$ 
6:      $[\rho(s, m'_s, m'_e, L') \wedge (\forall \tilde{m}'_e, \tilde{L}' : \rho(s, m'_s, \tilde{m}'_e, \tilde{L}') \rightarrow (s, m'_s, \tilde{m}'_e, \tilde{L}') \notin bad)]$ 
7:     Return  $S$ 
8:   end if
9: end procedure

```

Fig. 2. Pseudo-code for the composition algorithm

6.2 Sound composition

For the sound composition, we use the algorithm from Fig. 2 with synthesized systems as input. Intuitively, the algorithm tries to weave the two strategies in a way that does not violate either. If the strategies “agree” on the steps to be taken and their order, then the algorithm will succeed. Otherwise, the algorithm will fail. Note that the input strategies are not necessarily maximal, therefore the fact that the algorithm did not succeed in weaving them together does not mean there are no other winning strategies that can be successfully composed.

When the algorithm is given synthesized systems as input, it is sound, i.e., if it finds a system, then it is a synthesized system for the composite specification that induces a system-winning strategy. The formal proof of this claim is omitted from this version of the paper. Intuitively, we observe that if $m_s = \perp$ in a given state (i.e., the system decides to let the environment play), then the environment may use the ∞ symbol to force the system to stay in this state forever. Therefore, if $m_s = \perp$ in a system-winning state, then this state necessarily satisfies q . The soundness proof relies on this observation, along with the fact that the only variables shared between the subset systems are m_s and m_e . The proof considers the system found by the algorithm, and the strategy induced by it. It shows that any run compliant with it is necessarily compliant with the strategies for the subset specifications, and therefore winning in them. Then, following the observation above, it is also winning for the composite specification.

6.3 Augmented Strategies

Often when one considers composition of two specifications, there are messages that appear in one specification and not in the other. The synthesis algorithm as presented here will not allow steps not appearing explicitly in the specification. For composition, however, each part should be allowed to advance as much as it wishes, while using messages appearing only in it.

In this section, we show how a synthesized system may be augmented to allow steps that appear only in another (given) specification. Although the augmented system might not be a winning one anymore (it may now choose infinitely many

steps from the other system without advancing), the composition algorithm is still sound when given these augmented systems as input.

Definition 2. *Given an LSC game structure, G , we define the set of system messages irrelevant to G to be the set of values for m'_s s.t. in any state, sending them changes nothing in the configuration, as follows:*

$$\text{irrel}(G) = \{\bar{m}'_s \mid \forall s, \bar{m}'_e, \bar{L}' : (s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho_e \rightarrow (\bar{L} = \bar{L}')\} \setminus \{\perp\}$$

Given two synthesized systems, S_1, S_2 , for game structures G_1, G_2 resp., we create augmented systems \tilde{S}_1, \tilde{S}_2 , by augmenting their transition relations with transitions in which the LSC configuration does not change, the system sends a message relevant only to the other system, and the environment sends no message, as follows (for $i, j \in \{1, 2\}, i \neq j$):

$$\tilde{\rho}_i(s, s') = \rho_i(s, s') \vee (s[L_i] = s'[L_i] \wedge s'[m_e] = \perp \wedge s'[m_s] \in \text{irrel}(G_i) \setminus \text{irrel}(G_j))$$

The algorithm, given augmented synthesized systems, is still sound, i.e., if it finds a system then it is a synthesized system for the composite specification that induces a winning strategy. The proof of the soundness is similar with augmented strategies, with the addition that steps resulting from augmenting one transition relation do not change the state of that system, and are always “real” steps in the other system, therefore there are finitely many such consecutive steps.

The strategy synthesized by the synthesis algorithm of [29] is one that allows only steps that strictly get it closer to a stable state (one that satisfies q). Following a stable state, any step leading to a winning state is allowed, and then again only steps that strictly get it closer to a stable state. One may further augment the system by allowing any step leading to a winning state from states that are not stable, but for which no “real” step has been taken since a stable state (but only ones resulting from augmenting the strategy).

6.4 Complete composition

For the complete part, we use the same algorithm from Fig. 2 on inputs that represent an over-approximation of the maximal winning strategies. These will be termed *optimistic strategies*. We show that if the algorithm is given optimistic strategies as input, then it returns an optimistic strategy. Therefore, if no system is returned, then the composite specification is unrealizable.

Definition 3. *Given an LSC game structure, G , a strategy is optimistic if $\forall s$, and \bar{m}'_s : if $(\forall \bar{m}'_e, \bar{L}'[(s, \bar{m}'_s, \bar{m}'_e, \bar{L}') \models \rho_e] \rightarrow (\bar{m}'_s, \bar{m}'_e, \bar{L}') \in \text{win}(G))$, then $\bar{m}'_s \in f(s)$. i.e., an optimistic strategy allows any system step that will necessarily lead to a winning state. A system inducing an optimistic strategy will be termed an optimistic system.*

In other words, an optimistic strategy must allow any transition to a state from which the system can win, and is therefore an over-approximation of the maximal winning strategy.

One can construct the minimal optimistic strategy by allowing any system step from a winning state after which any environment step reaches another

winning state. Such a strategy will never violate any safety constraint (since it will not lead to a non-winning state), but it might violate liveness constraints.

If the composition algorithm from Fig. 2 is given optimistic systems as input, then it returns an optimistic system for the composite specification. Intuitively, the proof relies on the fact that a restriction of a winning state to a subset game structure is a winning state in it, therefore the synchronous parallel composition step induces an optimistic strategy. It then shows that no winning state is ever added to *Bad*, therefore no transitions between winning states are removed.

As a corollary, the algorithm is complete, i.e., if it finds no system, then the composite specification is unrealizable.

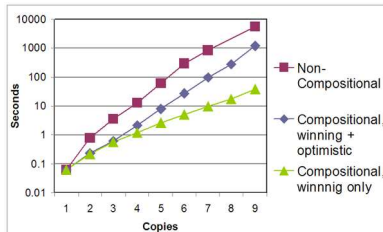
Note that even though the computed strategy is not the minimal optimistic strategy, it also will never violate a safety requirement, assuming the two input strategies never do so. This ensures that if we start from minimal optimistic strategies, and keep composing them (and the results of composing them), we will get a system that never causes a violation to any LSC. The reason the algorithm is not sound is that the resulting system, by being optimistic, does not guarantee to pass control back to the environment infinitely often. It might enter an infinite loop of system events, even though neither violates any chart.

6.5 Towards a sound and complete algorithm

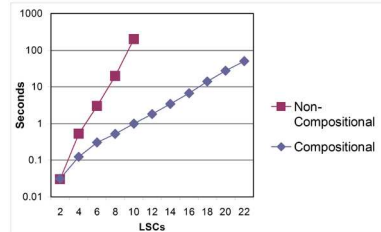
The composition algorithms described here are not meant to completely replace the full synthesis algorithm, but rather as fast alternatives that for some cases may work, and for others might fail. In such cases, there may be a need for applying full synthesis on the composite specification.

We now briefly describe possible extensions for these algorithms that together form a sound and complete algorithm. These are described in very general guidelines, and further implementation details are left as future work. The main reason for this is that the resulting algorithm may be too slow to be of any practical usage. Moreover, one part of the extension (the sound part) must be applied at the bottom-most level, i.e., when synthesizing a system. If one performs several composition steps, and now realizes he needs to further extend the system, he needs to start from the beginning, strengthen the synthesized systems and compose them together again. For the other part (the complete part), there does not seem to be a symbolic implementation.

The sound part of the algorithm can be extended as follows. The synthesis algorithm, as described above, finds strategies that at each step strictly move towards a stable state (one that satisfies q). By introducing an extra variable, *counter*, one can allow a given number of steps to move away from stable states (in each superstep). The intended usage is as follows: one sets *counter* to some low value, synthesizes his basic systems, and composes them. If some composition step fails, the basic systems can be resynthesized with larger initial *counter*, thus improving the chances for the compositions to succeed (yet extending its complexity and running time). If the composite system is realizable, then there exists a large enough initial *counter* for which the compositions will succeed.



(a) Running time (log-scale) as a function of the number of disjoint copies of a 6-LSC specification.



(b) Running time (log-scale) as a function of the number of LSCs, for inconsistent chain specifications.

Fig. 3. Running times for two parameterized examples

The complete part can be extended by strengthening the computation of bad states. Currently, states are marked bad if they may lead only to themselves or to other bad states. However, the strategy might allow infinite loops of system moves. Since the strategy does not violate any LSC (assuming initially minimal optimistic strategies were used), if such loops were avoided altogether, the resulting system would have been a winning one. Thus, by identifying such loops of increasing size, one improves his composed optimistic strategy, and if the system is unrealizable, eventually it will be proven as such.

An algorithm that alternately increases the counter and the loop size will be sound and complete. Details of these extensions are left as future work.

7 Results

We implemented our approach as part of the new Scenario-Based Tool [33] developed at Microsoft Research Cambridge, using TLV [31] for the symbolic computations. We now describe some experimental results from these tools.

The first example is a simple specification consisting of 6 LSCs, that refer to two objects. Following an initial environment message, the system must send the messages m_1 , m_2 , m_2 in this order before passing control back to the environment. This specification was replicated, using disjoint sets of messages, with the number of replicas parameterized, and ranging from 1 to 10. Each copy was synthesized separately and the results were composed. Figure 3(a) compares the running time (log-scale) as a function of the number of copies, for: (a) the compositional approach, when only winning systems are composed, (b) the compositional approach, when both winning and optimistic systems are composed, and (c) non-compositional synthesis of the entire specification. The compositional approach, when only winning strategies are composed, is clearly significantly faster, but if a composition step would have generated an inconsistent specification, it could not have been proved without optimistic strategies.

Another example is adopted from [12], where it is shown that synthesis is strictly stronger than smart play-out. We modify the example to form a series

of inconsistent specifications of growing lengths, where specification i requires considering i supersteps ahead in order to prove its inconsistency. Figure 3(b) shows the running time (log-scale) as a function of the specification size, for compositional synthesis (in which each LSC is synthesized separately and composed into the system) as opposed to full synthesis of the specification. Clearly, the compositional approach saves significant running time. It is worth mentioning that inconsistency is proven when the composition is performed in a specific order. Different choices of the order did not manage to prove inconsistency.

Two more test cases were generated, both inspired by a biological model describing the process of vulval precursor cell fate determination in the development of the *C. elegans* nematode [20]. In one, (a simplification of) the different developmental steps were each synthesized separately, and the results were composed. This specification consists of 22 LSCs. Without composition, the synthesis of the entire system did not finish within 5 days, whereas the compositional approach obtains a running system in less than 3 minutes. The second specification focuses on the last developmental stage, and demonstrates the incremental nature of the specification process, while using compositional synthesis. This system was composed in 9.85 seconds, while the full non-compositional synthesis did not finish within 3 days. The latter example also acts as an example in which smart play-out may choose a superstep that is correct, but may lead to violations in future supersteps. The synthesized system, on the other hand, avoids such violations.

8 Related Work

In recent years there have been considerable research efforts on synthesizing executable systems from scenario-based requirements [26]. In many of these papers the requirements are given using a variant of classical Message Sequence Charts while the synthesized system is state-based. The main distinguishing feature of our work is that we consider synthesis from Live Sequence Charts, which are more expressive than most of the classical MSC variants.

One should realize that constructing a program from a specification is a long-known general and fundamental problem, dating back to work by Church [7] and tackled by [6, 32]. There has also been much research on synthesis from a specification given in temporal logic, starting with closed systems, that do not interact with the environment [27, 10], and later [30, 1, 36] dealing with the synthesis of open systems from Linear Temporal Logic specifications. The problems of realizability checking and synthesis from LTL are shown to be 2EXPTIME-complete. Despite this high complexity, progress has been made in the development and application of synthesis algorithms, by proposing new algorithms [25], using heuristic approaches [11], considering subsets of temporal logic [2, 28], smart implementation and application [19, 3, 34]. A compositional method for synthesis is presented in [24] building upon basic results first described in [25].

Synthesis from LSCs was first studied in [13], and is tackled there by defining consistency, showing that an entire LSC specification is consistent if and only if it

is satisfiable by a state-based object system, and then synthesizing a satisfying system. The work in [13] considers a core LSC subset consisting of messages only, similar to this paper, but does not implement the algorithms or study the practical questions related to implementation. A game theoretic approach to synthesis from LSCs involving a reduction to parity games is described in [4], the authors summarize the experimental results as negative, partially due to a poor prototype implementation. Synthesis from LSCs using a reduction to CSP is described in [35]. In [22] synthesis from LSC is tackled somewhat similarly to this paper, however compositional synthesis is not considered at all.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and Unrealizable Concurrent Program Specifications. In *Proc. 16th Int. Colloq. Automata, Languages and Programming, LNCS vol 372*, pp. 1–17, 1989.
2. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller Synthesis for Timed Automata. In *IFAC Symp. on System Structure and Control*, pp. 469–474, 1998.
3. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic Hardware Synthesis from Specifications: A Case Study. In *Proceedings of the Design, Automation and Test in Europe*, pp. 1188–1193, 2007.
4. Y. Bontemps, P. Heymans, and P. Y. Schobbens. From Live Sequence Charts to State Machines and Back: A Guided Tour. *IEEE Trans. Software Eng.*, 31(12):999–1014, 2005.
5. Y. Bontemps and P. Schobbens. Synthesizing Open Reactive Systems from Scenario-Based Specifications. In *Proc. of the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03)*, 2003.
6. J. Büchi and L. Landweber. Solving Sequential Conditions by Finite-State Strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.
7. A. Church. Logic, Arithmetic and Automata. In *Proc. 1962 Int. Congr. Math.*, pp. 23–25, Upsala, 1963.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Form. Meth. in Sys. Design*, 19(1):45–80, 2001.
9. W. Damm, T. Toben, and B. Westphal. On the Expressive Power of Live Sequence Charts. In *Program Analysis and Compilation, LNCS vol. 4444*, pp. 225–246, 2006.
10. E. Emerson and E. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.
11. A. Harding, M. Ryan, and P. Schobbens. A New Algorithm for Strategy Synthesis in LTL Games. In *Proc. 11th Intl. Conf. on Tools and Alg. for the Construction and Analysis of Systems (TACAS'05), LNCS vol. 3440*, pp. 477–492, 2005.
12. D. Harel, A. Kantor, and S. Maoz. On the Power of Play-Out for Scenario-Based Programs. 2009. To appear.
13. D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. of Found. of Comp. Sci. (IJFCS)*, 13(1):5–51, Feb. 2002.
14. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *FMCAD'02*, pp. 378–398, 2002.
15. D. Harel, H. Kugler, and A. Pnueli. *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements, LNCS vol. 3393*, pp. 309–324, 2005.
16. D. Harel and R. Marelly. *Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. 2003.

17. D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, LNCS vol. 4424, pp. 485–499, 2007.
18. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.
19. B. Jobstmann and R. Bloem. Optimizations for LTL Synthesis. In *6th Conf. Formal Methods in Computer Aided Design (FMCAD '06)*, 2006.
20. N. Kam, H. Kugler, R. Marelly, L. Appleby, J. Fisher, A. Pnueli, D. Harel, M. Stern, and E. Hubbard. A Scenario-Based Approach to Modeling Development: A Prototype Model of *C. Elegans* Vulval Fate Specification. *Developmental Biology*, 323(1):1–5, 2008.
21. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *Proc. 11th Int. Conf. on Tools and Alg. for the Const. and Anal. of Systems (TACAS '05)*, LNCS vol. 3440, pp. 445–460, 2005.
22. H. Kugler, C. Plock, and A. Pnueli. Controller Synthesis from LSC Requirements. In *12th International Conference on Fundamental Approaches to Software Engineering, (FASE 2009)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2009.
23. H. Kugler and I. Segall. Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. Technical report, Microsoft Research, 2009.
24. O. Kupferman, N. Piterman, and M. Vardi. Safrless Compositional Synthesis. In *Proc. 18th International Conference on Computer Aided Verification*, LNCS, 2006.
25. O. Kupferman and M. Vardi. Safrless Decision Procedures. In *Proc. 46th IEEE Symp. on Found. of Computer Science*, pp. 531–540, Pittsburgh, October 2005.
26. H. Liang, J. Dingel, and Z. Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In *Proc. of the Intl. Work. on Scenarios and State Machines: Models, Algs., and Tools (SCESM'06)*, pp. 5–12, 2006.
27. Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Programming Languages and Systems*, 2:90–121, 1980.
28. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, LNCS vol. 3855, pp. 364–380, 2006.
29. A. Pnueli. Extracting Controllers for Timed Automata. Technical report, NYU, 2005.
30. A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pp. 179–190, 1989.
31. A. Pnueli and E. Shahar. A Platform for Combining Deductive with Algorithmic Verification In *Proc. 8th Intl. Conf. Computer Aided Verification (CAV'96)*, LNCS vol. 1102, pp. 184–195, 1996.
32. M. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
33. Microsoft Research Cambridge, Scenario-Based Tool for Biological Modeling, 2009. <http://research.microsoft.com/SBT/>.
34. S. Sohail, F. Somenzi, and K. Ravi. A Hybrid Algorithm for LTL Games. In *Proc. of the 9th Int. Conference on Verification, Model Checking, and Abstract Interpretation*, LNCS vol. 4905, pp. 309–323, 2008.
35. J. Sun and J. S. Dong. Synthesis of Distributed Processes from Scenario-Based Specifications. In *Intl. Symp. Formal Methods Europe (FM'05)*, pp. 415–431, 2005.
36. H. Wong-Toi and D. Dill. Synthesizing Processes and Schedulers from Temporal Specifications. In *2nd Intl. Work. on Comp. Aided Verification (CAV '90)*, pp. 272–281, 1990.