Journal of Computer and System Sciences 78 (2012) 970–980

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Synthesis from scenario-based specifications ☆

David Harel *, Itai Segall **

Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel

ARTICLE INFO

Article history:
Received 5 April 2010
Received in revised form 20 December 2010
Accepted 5 August 2011
Available online 17 August 2011

Dedicated to the dear memory of Amir Pnueli: Friend, colleague, and a truly towering figure in computer science

Keywords:
Synthesis
Live sequence charts
LSC
Specification
Scenario-based programming

ABSTRACT

We consider the problem of the automatic generation of reactive systems from specifications given in the scenario-based language of *live sequence charts* (LSCs). We start by extending the language so that it becomes more suitable for synthesis. We then translate a system specification given in the language into a two-player game between the system and the environment. By solving the game, we generate a winning strategy for the system, which corresponds to a correct implementation of the specification. We also define two notions of system correctness, and show how each can be synthesized.

1. Introduction

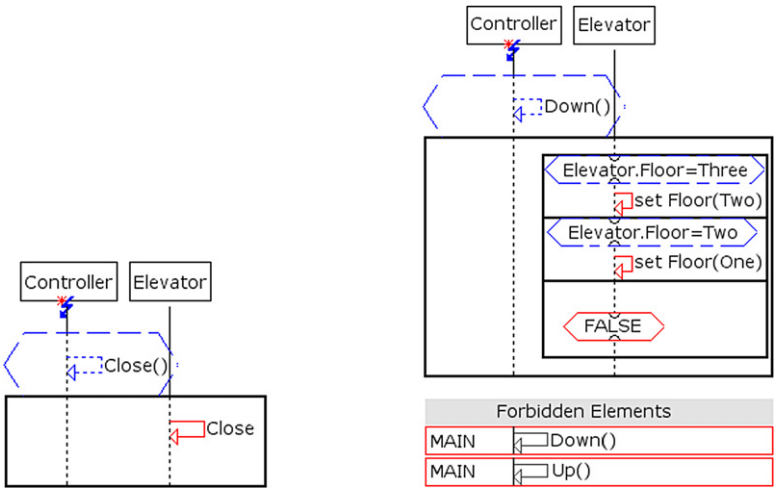
Implementation of systems, especially reactive systems, is an error-prone task. Very strong and sophisticated verification algorithms have been developed over the years for comparing an implementation with its specification, thus verifying correctness. Automatic synthesis of a system directly from its specification would have been even better. Given a specification in some rich and expressive language, a synthesis process would automatically generate a system that adheres to the specification [27].

In this paper, we introduce an approach for synthesis from *live sequence chart* (LSC) specifications. LSCs constitute a visual and intuitive language for scenario-based specification of reactive systems [5]. Syntactically, LSCs extend message sequence charts (MSCs) [14] by adding hot and cold modalities, standing for things that must or may happen, respectively. An operational semantics for the language, termed *play-out*, is defined in [12]. A version thereof that plans its steps ahead, thus reducing violations, is *smart play-out* [9]. However, since its lookahead is limited to a single superstep (a series of events by the system, encapsulated between two consecutive events by the environment), even smart play-out does not guarantee that violations will never be encountered. In order to ensure that regardless of the environment actions the system will indeed satisfy the specification, full synthesis appears to be required.

☆ This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

* Corresponding author.
** Principal corresponding author.
E-mail addresses: dharel@weizmann.ac.il (D. Harel), itai.segall@weizmann.ac.il (I. Segall).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.08.008



(a) The `CloseDoors` LSC (b) The `FloorDown` LSC

Fig. 1. Two charts from the elevator specification. (For interpretation of colors in this figure, the reader is referred to the web version of this article.)

Given a specification in the language of LSCs, we first translate it into a two-player game (specifically, a GR(1) game), solvable in time polynomial in the size of the state space [21]. A winning strategy in the game, if one exists, represents an implementation of a system that adheres to the given specification. If no such strategy exists, the specification is unrealizable; i.e., there is no system that satisfies it.

Following some preliminary material (Section 2), we slightly extend and adapt certain features in the language to be better suited for the synthesis process (Section 3). We introduce an example specification of a three-story elevator (Section 4), and then turn to formally defining the problem of synthesis from LSC specifications (Section 5). In Section 6 we introduce the translation of the specification into a game structure solvable by the GR(1) algorithm, and in Section 7 we give additional examples and review experimental results. Finally, we discuss related and future work (Sections 8 and 9).

2. Preliminaries

2.1. Live sequence charts

Live sequence charts (LSCs) [5,12] are an extension of message sequence charts (MSCs) [14]. An LSC consists of a *prechart* and a *main chart*, with the intended semantics that whenever the prechart is satisfied in a run, the main chart must also be satisfied. The prechart and main chart are denoted by a dashed blue hexagon and a solid black rectangle, respectively. As in MSCs, objects in LSCs are denoted by vertical lines, termed *lifelines*. Messages between objects are denoted by horizontal arrows between two lifelines (or from a lifeline to itself). *Conditions* are denoted by hexagons. Most constructs in the language, including messages and conditions, may be either *hot* or *cold*, denoted by solid red and dashed blue lines, respectively. A hot construct must be satisfied, while a cold one may be satisfied. Failing to satisfy a cold construct is thus a legal part of the execution.

For example, consider the LSC in Fig. 1(a). It has two lifelines, representing the objects `Controller` and `Elevator` (for now, ignore the lightning icon on the Controller object, which will be discussed later). The prechart and main chart consist of a single message each. This LSC states that whenever the Controller sends itself the `Close` message, the Elevator must eventually also send itself a `Close` message (representing the action of the elevator closing its doors). Note that the latter message is hot, thus the elevator doors must eventually be closed. A run in which the controller sends itself `Close` (thus satisfying the prechart), but in which the elevator doors never close, is illegal. Moreover, a run in which the controller sends itself another `Close` message before closing the doors is also illegal, as it violates the *partial order* dictated by the chart. In other words, each chart, in addition to requiring messages to be sent according to the partial order in which they appear, also forbids them from being sent in any different order.

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* (resp. *environment*) *message*, i.e., controlled by the system (resp. environment), if it is sent from an object controlled by the system (resp. environment). We assume here that the main chart does not contain environment messages.

Some of the more advanced constructs in the language can be found in the LSC `FloorDown` in Fig. 1(b). Its prechart contains one message, `Down`, sent from the `Controller` object to itself. Its main chart consists of a *switch-case* construct with cases for the elevator being on floors three and two, and a default case if none of the others hold. The default case consists of a FALSE hot condition, representing a scenario that is not allowed to happen (since the FALSE condition can never be satisfied). The bottom of the chart contains two *forbidden messages*. They are hot, as denoted by the solid red frame, and

state that the messages `Down` and `Up` sent from the controller to itself are forbidden as long as the main chart has not been satisfied. The scope of this prohibition is the main chart, as denoted by the word MAIN.

To summarize, the LSC `FloorDown` in Fig. 1(b) states that whenever the controller sends itself the `Down` message, the current floor of the elevator is checked: if it is three it moves to two, and if it is two it moves to one. Any other case (i.e., in a three-story elevator, the case where it is on the first floor) is considered a violation of the specification. Until the elevator moves, the controller may not send itself a `Down` or `Up` message.

2.2. Play-out and smart play-out

Play-out [12] is an operational semantics for the language, enabling a specification (typically, a set of LSCs) to be executed directly, without generating intermediate code. A *cut* is maintained for each LSC, showing where execution is at the moment along each lifeline. A *configuration* is a snapshot of the state of the specification, including the cuts of all LSCs and the values of all object properties. A chart is considered to be *active* in a configuration if its cut is in the main chart. A message is *enabled* in a cut of a chart if it appears immediately after the cut in the chart. A message is *violating* in a cut of a chart if it appears in the chart, but is not enabled in it. Play-out is carried out in *steps*. At each step, a single message is executed.

Different appearances of the same message are *unified* at runtime. Thus, if a message is executed, all its appearances are considered. If it is enabled in several places, all the corresponding cuts are updated. A message might also be violating in some chart when executed (by appearing in the chart but not being enabled in the current cut of this chart). In some cases (e.g., when this chart is inactive, or if all enabled messages are cold), this is considered a legal step, and the cut will be updated accordingly (for example, the chart may be closed; i.e. its cut reset to the initial location before the prechart). This is termed a *cold violation*. In other cases, however (when a hot message is enabled), sending a violating message is considered a violation of the specification, and must be avoided.

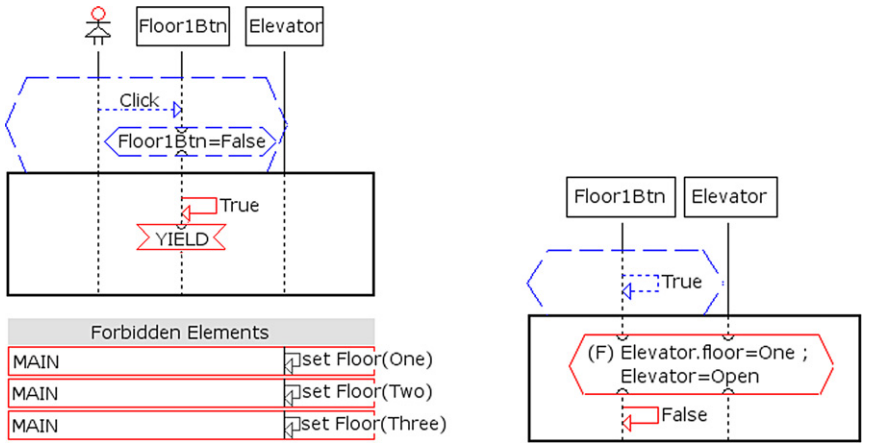
At each step, play-out chooses a single system message that is enabled in some active LSC and that does not violate the specification, and executes it. Play-out does not guarantee that no violations will occur (or rather that at each step there will exist an enabled message that is not violating). Violations might happen since play-out makes its choices arbitrarily, without considering their future consequences. Stronger play-out mechanisms are those of *smart play-out* [9] and *planned play-out* [13]. These are initiated following each environment step, and look for a sequence of system steps to perform in response (termed a *superstep*), that will lead the system to a state where no LSC is active (a *stable state*), in preparation for the next environment step. During this superstep, these algorithms guarantee that no violations will occur. However, looking only one superstep, or any finite number of supersteps, ahead is not sufficient either, as shown in [7]. This leads to the synthesis problem, i.e., given an LSC specification, find a reactive system that always adheres to the specification, or prove that one does not exist.

2.3. Game structures

A *two-player 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 (for a set of variables T, or for a single variable t, we denote a valuation of T or t by $\vec{t}$). Θ is the initial condition, and ρ_s and ρ_e represent the transition relations of the system and the environment, respectively. Since in this formalism the system is the first player, its transition relation may depend only on the current state, whereas that of the environment may also depend on the system's transition. More formally, ρ_s is a relation $\rho_s \subseteq X \times Y \times X$, where $(\vec{x}, \vec{y}, \vec{x}') \in \rho_s$ represents the fact that from the state $(\vec{x}, \vec{y})$, the system may set its variables to $\vec{x}'$ in the next state. Similarly, $\rho_e \subseteq X \times Y \times X \times Y$, where $(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \in \rho_e$ represents the fact that from the state $(\vec{x}, \vec{y})$, if the system sets its variables to $\vec{x}'$, the environment may set its variables to $\vec{y}'$. Finally, φ is the winning condition.

A *strategy* is a partial function mapping a series of states to the next system action, or a possible set thereof. A run is *compliant* with a strategy if each step taken by the system is 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 its transition relation) and is compliant with the strategy, is winning for the system, i.e., it 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 the initial condition Θ).

A GR(1) game is defined in [21] as a two-player game with a winning condition of the form $\varphi = \bigwedge_{i=1}^{n} \Box\Diamond p_i \rightarrow \bigwedge_{j=1}^{m} \Box\Diamond q_j$. Here, the p_is represent assumptions on the environment, and the q_is requirements from the system (both are boolean combinations of atomic propositions over the state variables), and the condition states that the system wins if in every run in which each of the assumptions is satisfied infinitely often, each of the requirements is also satisfied infinitely often. An algorithm for solving such games, i.e., for finding the set of winning states for the system, and for synthesizing a winning strategy for it (or proving that none exists), is given in [21] as well. The algorithm runs in time polynomial in the size of the state space. In this paper, we adopt this algorithm, with a slight modification for games in which the system plays first (this modification is for convenience of notation, and the games are equivalent; implementing the modification is quite straightforward and is not detailed here). In fact, in our case, the generalized Büchi winning conditions of $\bigwedge_{j=1}^{m} \Box\Diamond q_j$ suffice (they are easily reducible to GR(1) conditions by taking empty assumptions).



(a) The `Btn1on` LSC (b) The `Btn1off` LSC

Fig. 2. Two LSCs from the elevator specification, using the Yield construct and both types of conditions.

3. Adapting LSCs for synthesis

In order to adapt the LSC language to be better suited for synthesis, we introduce some modifications to the syntax and semantics of the language, as follows.

3.1. The yield construct

We adopt the superstep approach, first introduced in [9]: following an environment step, the system is allowed to perform any finite number of steps (a *superstep*) before yielding control back to the environment. In order to ease this, we add a *Yield* construct to the language, denoted by a concave hexagon. A Yield may be synchronized on any number of lifelines, and is advanced only when the system yields control to the environment. Using this construct, the specifier can force the system to stop at a certain point and listen to an environment event, rather than continuing its superstep for as long as it wishes. For example, consider Fig. 2(a). Whenever the prechart is satisfied in this chart, the object `Floor1Btn` will send itself the message `True`, and then the system must yield control.

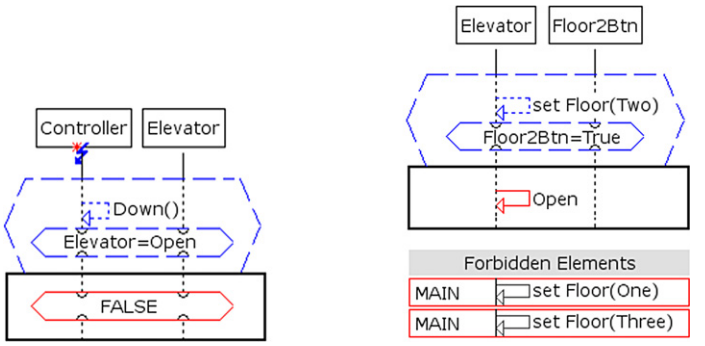
Note that when yielding control, the environment may send any message it wishes, including ones that appear elsewhere in the chart. This is not considered a violation of the partial order of the chart. In other words, a Yield may be considered as a wildcard for all environment messages: it allows any one of them to be sent at this point without violating the chart. For example, consider again the LSC in Fig. 2(a). When the system yields control due to the main chart Yield construct, the environment may send any message, including the message `Click` from the user to `Floor1Btn`. Since this message appears elsewhere in the chart, this could have been considered a violation of the partial order, but since the Yield also acts as a wildcard for all environment messages, this is not the case, and this message is considered enabled at this point in time.

3.2. Spontaneous objects

In the operational semantics of [12], the system may send a message only when it is enabled in some main chart. In other words, in order for a message to be sent, the specifier must explicitly state so in some chart. When treating LSCs as a language for synthesis, the specifier often does not wish to explicitly state that a message must be sent, but rather leaves this choice to the synthesized system.

For this purpose, we introduce *spontaneous objects*. These are objects with the property that messages sent from them may be sent by the synthesized system whenever it is deemed necessary. For non-spontaneous objects, the standard play-out semantics holds; i.e., a message from them may be sent only if it is enabled in some main chart. A lifeline representing a spontaneous object is denoted by a small lightning icon below the object name. We say that a message is spontaneous iff its sender is a spontaneous object.

For example, consider Fig. 1(a), whose LSC has two lifelines – one for the `Controller` object, which is spontaneous, and the other for the `Elevator` object, which is not spontaneous. The message `Close` from `Elevator` to itself may be sent only when it is enabled in some main chart (since its sender, the `Elevator`, is not spontaneous). The message `Close` from the `Controller` to itself, on the other hand, may be sent spontaneously by the synthesized system.



(a) The `DownDoorsClosed` LSC (b) The `DontSkipFloor2` LSC

Fig. 3. The safety and user-experience charts of the elevator specification.

3.3. Types of conditions

In the original play-out mechanism of [12], the decision as to when a condition is evaluated is rather complicated. In order to ease our translation into a formal model, we propose two kinds of conditions: *immediate* and *eventual* ones.

An immediate condition is evaluated as soon as it becomes enabled. If it is hot, then it *must* hold at that moment, and if it is cold, it *may* hold (i.e., if it does not hold, then the rules of a cold violation are followed). For example, consider the prechart of the LSC in Fig. 2(a). The condition in it is an immediate one (it does not contain an "(F)" symbol), and thus will be evaluated immediately upon being enabled. Therefore, whenever the user sends `Click` to `Floor1Btn`, the state of `Floor1Btn` is checked (representing the button's light). If it is off (represented by False), the prechart is satisfied and the main chart becomes enabled. Otherwise, the prechart is closed.

The evaluation of an eventual condition, on the other hand, is deferred until the condition holds. As soon as it does hold, it will be evaluated (i.e., the cut will progress beyond it). Note that an eventual condition will never cause violations (either cold or hot ones), since it is simply not evaluated unless it holds. It may prevent the chart from progressing, however. Therefore a synthesis algorithm that wishes to cause the chart to terminate must eventually satisfy it. For example, the condition in the main chart of the LSC in Fig. 2(b) is eventual (indicated by the "(F)" symbol), and it states that eventually the elevator must be on the first floor with its doors open.

4. Completing the example

We now complete the introduction of the example specification for the three-story elevator. It will serve as a running example throughout the paper.

The example considers a single environment object (the `User`), and five system objects: the `Elevator`, a `Controller`, and three floor buttons, `Floor1Btn`, `Floor2Btn` and `Floor3Btn`, one for each floor. The `Controller` is the only spontaneous object. Messages from the `Controller` represent decisions that the system should be able to make spontaneously; e.g., to move the elevator one floor up. The other objects respond to the decisions, and to user events, and are thus not spontaneous.

The LSCs in Fig. 2(a) and (b) represent three charts each, in which the `Floor1Btn` lifeline is replaced by `Floor1Btn`, `Floor2Btn` and `Floor3Btn`. These charts are the essence of the specification. LSC `Btn1on` (and its copies) in Fig. 2(a) handles turning on the button light whenever the button is pressed, and LSC `Btn1off` in Fig. 2(b) (and its copies) makes sure that the elevator reaches all requested floors.

The LSCs in Fig. 1(a) and (b) specify how the system may close the elevator doors, and move a floor down. Similar charts exist for opening the doors and moving a floor up, respectively.

Finally, the charts in Fig. 3(a) and (b) capture safety and user-experience requirements. LSC `DownDoorsClosed` in Fig. 3(a) states that the `Controller` may not send itself the message `Down` while the elevator doors are open (thus forcing the synthesized system to close its doors before moving the elevator down). A similar chart exists for the message `Up`. LSC `DontSkipFloor2` states that if the elevator gets to the second floor while the light for that floor is on, it must open its doors, and must not move before doing so. Similar charts exist for the first and third floors.

Note that in most previous work on execution and synthesis from LSC specifications, the specifier would have had to explicitly specify, for example, that if a user presses the first floor button, the elevator closes its doors, gradually moves to the first floor, and then opens its doors. Here, especially thanks to the notion of spontaneous objects, the specification can be made much more declarative. Ours merely states that the elevator must get to that floor with its doors open, states how an elevator moves and handles its doors, and adds some safety rules (e.g., that it is not allowed to move while the doors are open). The rest is taken care of automatically by the synthesis algorithm. For example, if the system wishes to close the doors, it may send the controller's close message spontaneously, which will eventually cause the elevator doors to close.

5. The problem definition

Synthesis is about the automatic generation of a system that satisfies a given specification. In order to formalize this notion, we have to define what it means to satisfy a specification. We give two slightly different definitions for this.

We start by defining a run, which corresponds to a trace of messages, and which captures the series of configurations that the specification takes when monitoring the trace. The definition of legal traces captures the notion of non-spontaneous messages, and ensures that they are sent only when allowed.

Definition 1 *(Specification).* A *specification* is a set of LSCs, and an initial configuration.

Definition 2 *(Trace).* Given a specification S, let Σ_E (resp. Σ_S) be the set of environment (resp. system) messages in S. A *trace* is an infinite sequence of the form $(\Sigma_E(\Sigma_S)^*)^\omega$.

Note that we adopt the superstep approach of [9]: following each environment step, the system may perform a finite series of steps before yielding control back to the environment.

Definition 3 *(Run).* Given a specification S and a trace $\alpha_0\alpha_1\cdots$, where $\forall i\ \alpha_i \in \Sigma_E \cup \Sigma_S$, the corresponding *run* is a sequence of configurations $c_0, c_1, c_2, \ldots$, where c_0 is the initial configuration of the specification and for all i, c_{i+1} is the state of the specification after sending α_i from configuration c_i.

Definition 4 *(Legal trace).* Given a specification S, a trace $\alpha_0\alpha_1\cdots$, and its corresponding run $c_0, c_1, c_2, \ldots$, the trace is *legal* in S if for all i, at least one of the following holds:

1. α_i is an environment message,
2. α_i is a spontaneous system message,
3. α_i is enabled in some main chart in configuration c_{i-1}.

We want a run to satisfy a specification if it satisfies all of its charts infinitely often, where a chart is satisfied when it is inactive. By requiring that each chart is satisfied infinitely often, we make sure that whenever a chart becomes active it also eventually becomes inactive. The difference between the two definitions of satisfaction below is in whether all charts need to be satisfied simultaneously or not.

Definition 5 *(Global justice satisfaction).* Given a specification S, consider the set C of all possible configurations of S. Let $GJ(S) \subseteq C$ be the set of configurations s.t. all LSCs of S are inactive: $GJ(S) = \{c \in C\colon\ \forall l \in LSCs(S)\ l \text{ is inactive in } c\}$. A run ρ *satisfies the specification* S in the *global justice* sense if $\inf(\rho) \cap GJ(S) \neq \emptyset$, where $\inf(\rho)$ is the set of states visited infinitely often in ρ.

Thus, in order for a run to satisfy a specification in the global justice sense, it must infinitely often reach a state in which all charts are inactive. This definition is a direct extension of the smart play-out approach [9]: following each environment step, the system strives to reach a stable state, in which all charts are inactive and the system has no more obligations.

A weaker definition is that of local justice, where the LSCs have to be infinitely often satisfied (i.e., inactive), but not necessarily simultaneously.

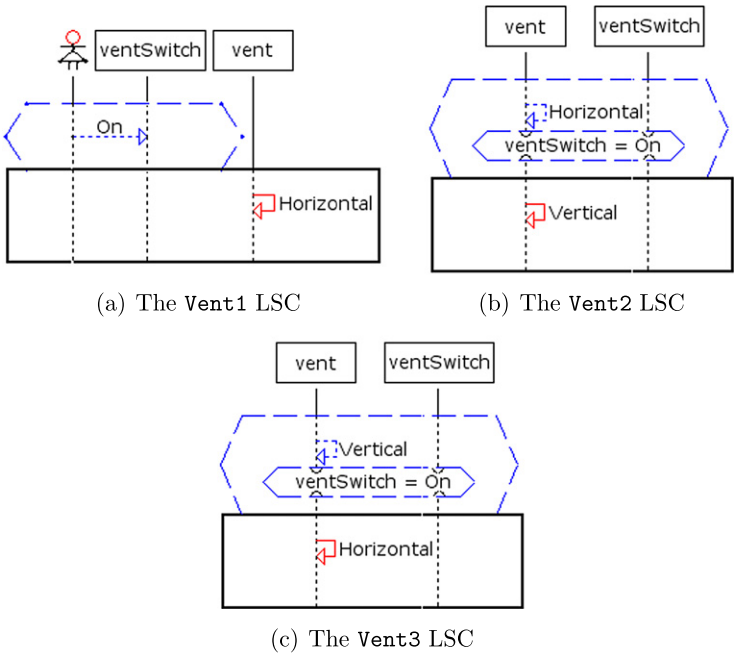
Definition 6 *(Local justice satisfaction).* Given a specification S, consider the set C of all possible configurations of S. Let $l \in LSCs(S)$, and $LJ_l(S) \subseteq C$ be the set of configurations s.t. l is inactive: $LJ_l(S) = \{c \in C\colon\ l \text{ is inactive in } c\}$. A run ρ *satisfies the specification* S in the *local justice* sense if $\forall l \in LSCs(S)\ \inf(\rho) \cap LJ_l(S) \neq \emptyset$, where $\inf(\rho)$ is the set of states visited infinitely often in ρ.

For both satisfaction definitions, we say that a *trace* satisfies a specification iff its corresponding run does.

In order to define a system that satisfies a specification, we first define an environment and a system. An environment is merely an infinite sequence of environment messages, and a system maps finite prefixes of the environment to finite sequences of system messages (supersteps). The trace generated by a system in an environment is an alternation between environment steps and corresponding system supersteps.

Definition 7 *(Environment).* Given a specification S, let Σ_E be the set of environment messages in S. An *environment env* is an infinite sequence $env \in \Sigma_E^\omega$.

Definition 8 *(System).* Given a specification S, let Σ_E and Σ_S be the sets of environment and system messages in S, respectively. A *system sys* is a function $sys\colon \Sigma_E^* \to \Sigma_S^*$.



(a) The `Vent1` LSC (b) The `Vent2` LSC

(c) The `Vent3` LSC

Fig. 4. Charts for the vent system, unrealizable in the global justice sense.

Definition 9 *(Generated trace).* Let *sys* and *env* be a system and an environment, respectively. Assume $env = \eta_0\eta_1\cdots$, where for all i, $\eta_i \in \Sigma_E$. The *trace generated by system sys in environment env*, denoted $\mathrm{tr}(sys, env)$ is: $\mathrm{tr}(sys, env) = sys(\varepsilon)\cdot\eta_0\cdot sys(\eta_0)\cdot\eta_1\cdot sys(\eta_0\eta_1)\cdots$.

We now define a system satisfying a specification and a realizable specification.

Definition 10 *(System satisfaction).* Given a system *sys*, we say that *sys satisfies a specification S* in the global (resp. local) justice sense, if for every environment, *env*, the trace $\mathrm{tr}(sys, env)$ is legal and satisfies *S* in the global (resp. local) sense.

Definition 11 *(Realizability).* A specification *S* is *realizable* in the global (resp. local) justice sense if there exists a system that satisfies *S* in the global (resp. local) justice sense. Otherwise, it is *unrealizable*.

The local justice definition is strictly weaker. If a specification is realizable in the global justice sense, then it is also realizable in the local justice sense. The other way around does not hold, i.e., there are specifications that are unrealizable in the global justice sense but realizable in the local justice one. For example, consider adding the three LSCs from Fig. 4 to the elevator example. These LSCs add a vent to the elevator, that must keep spinning (by switching between horizontal and vertical) as long as the switch is on. In an environment that turns the switch on and leaves it that way forever, there is no system that can satisfy the two LSCs from Fig. 4(a) and (b) simultaneously, thus this specification is unrealizable in the global justice sense. Obviously, it is realizable in the local justice sense, since the system may rotate the vent continuously, thus satisfying each of the charts infinitely often.

Note that even if a specification is realizable in both senses, there may be systems that satisfy it in the local justice sense but not in the global justice one. Moreover, our synthesized systems work by setting intermediate goals (in general, a requirement from the set of requirements in the winning condition of the game, or in our case, making a specific chart inactive). Whenever such a goal is set, our systems try to satisfy it as early as possible. In each step they choose an event that necessarily gets them closer to their current goal (where the distance from a goal is the minimum number of steps required in order to achieve it). However, once a goal is reached, for one step they are less strict. In the step immediately following the goal, more events are allowed, even ones that get the system further away from its next goal (as long as the next goal will still be achievable). Therefore, a system synthesized in the local justice setting may be more responsive to the environment, since whenever it satisfies some LSC it may consider a goal achieved and for a single step it will be less eager to continue, thus allowing the environment to step in. An example for this is given in Section 7.

6. The LSC game structure

In this section, we represent a given LSC specification *S* as a GR(1) game structure $G(S) = \langle V, X, Y, \Theta, \rho_s, \rho_e, \varphi\rangle$. The variables in the structure should capture the configuration of the specification (including all its cuts and the values of

all relevant object properties). Their transition relation updates them according to the LSC semantics, given the choice of messages sent at each time step. The transition relation should also capture the superstep approach. The winning condition is derived from the definition of a system satisfying a specification, as discussed in Section 5.

6.1. The state variables

We adopt the general framework of the state variables from [16]. In this framework, the system controls a single variable, m_s, which represents the message sent by a system object in the current step. The environment controls a variable m_e, which represents the message sent by an environment object in the step, and a set of variables C that represent the current configuration. At every turn, the system may do one of three things: send a single system message, do nothing (by using the special no-op symbol, $\bot$) or yield control for this step (by using the special symbol $\top$). In a step in which the system chooses to yield control, the environment may choose a message of its own to send, or use its own no-op symbol, $\bot$, for doing nothing. Given the choices of m_s and m_e for the next step, updating the variables of C is deterministic, and follows the operational semantics of [12] directly (with the additions and modifications introduced in Section 3).

In [16], as in [9], C (denoted L there) captures cuts by introducing a variable for each lifeline, ranging over its locations. Here, we take a slightly different approach. For each chart, we build an automaton (termed a *cut-automaton*), in which a state represents a possible cut. The current state of the automaton can then be represented by a single variable, ranging over its states. By building the automaton in a BFS-like manner, starting from the minimal cut, we may consider only the reachable cuts in each chart. This representation is therefore much more efficient for charts that are highly "ordered", but much less efficient for very unordered charts (since in such charts the number of reachable cuts is relatively large, hence explicitly enumerating them becomes inefficient). The exact analysis of these two possibilities, and heuristics for choosing between them, are out of scope of this paper.

More formally, the sets V, X and Y of $G(S)$ are as follows:

- The variables are $V = \{m_s, m_e\} \cup C$, where:
  - m_s ranges over the set of system messages appearing in S, plus the symbols $\bot$ and $\top$.
  - m_e ranges over the set of environment messages appearing in S, plus the symbol $\bot$.
  - $C = \{c_l\colon\ l \in \mathit{LSCs}(S)\} \cup \{c_p\colon\ p \text{ object property appearing in } S\}$, where:
    - for an LSC l, c_l ranges over the states in the cut-automaton of l;
    - for an object property p, c_p ranges over the possible values of p.
- There is only one system variable, $X = \{m_s\}$.
- The environment variables are $Y = \{m_e\} \cup C$.

6.2. The transition relations

The transition relations ρ_s and ρ_e should capture the following: (1) the restrictions on when messages may be sent; (2) the superstep approach; (3) the semantics of updating a configuration in LSCs.

Intuitively, message restrictions are captured by requiring non-spontaneous messages to be enabled in order to be sent. The superstep approach is represented by the fact that in any step in which the system takes a step other than yield ($\top$), the environment must take a no-op step ($\bot$).

The LSC semantics is captured by the transitions of the cut-automaton that update its state (representing the cut of the chart) according to the taken steps. Each automaton also has a sink hot-violated state capturing the fact that a hot violation has occurred and the chart can no longer be satisfied. Note that this reduces a safety condition to a liveness one, since violations are allowed by the transition relation, but will cause the liveness conditions to be unsatisfiable. To ensure this, the chart is considered active in this sink state.

More formally, the transition relations ρ_s and ρ_e of $G(S)$ are defined as follows:

- $(\vec{m}_s, \vec{m}_e, \vec{c}, \vec{m}'_s) \in \rho_s$ iff $\vec{m}'_s$ is a spontaneous message or $\vec{m}'_s$ is enabled in the configuration C.
- $(\vec{m}_s, \vec{m}_e, \vec{c}, \vec{m}'_s, \vec{m}'_e, \vec{c}') \in \rho_e$ iff the following conditions hold:
  1. $\vec{m}'_s \neq \top \rightarrow \vec{m}'_e = \bot$.
  2. $\forall c_l \in C\ \ \delta_l(\vec{c}_l, \vec{m}'_s, \vec{m}'_e) = \vec{c}'_l$, where δ_l is the transition function of the cut-automaton of l, mapping a state, a system step and an environment step to the next state.
  3. $\forall c_p \in C\ \vec{c}'_p$ is the value of the property p after performing $\vec{m}'_s$ and $\vec{m}'_e$ from a state in which the value of p is $\vec{c}_p$.

6.3. The winning condition

The winning condition formulates the condition for the system to be winning in a run. It is thus influenced directly by the definition of satisfaction of a specification. For global justice, we require the system to infinitely often yield control from a state in which all LSCs are inactive. For local justice, we require for each LSC that infinitely often a state in which it is inactive is visited, and also that infinitely often the system yields control (thus ensuring finite supersteps).

More formally, the winning condition φ in $G(S)$ for global justice synthesis is defined as $\varphi = \Box\Diamond[\bigwedge_{l \in LSCs(S)} (l \text{ is inactive}) \wedge \bigcirc m_s = \top]$. Note that this is not strictly a GR(1) formula, due to the usage of the "next" operator (denoted by a circle), but can clearly be transformed into a GR(1) formula by introducing a boolean variable *PrevInactive* s.t. $PrevInactive' = 1 \leftrightarrow \bigwedge_{l \in LSCs(S)} (l \text{ is inactive})$ and setting $\varphi = \Box\Diamond[PrevInactive \wedge m_s = \top]$.

For local justice synthesis, we define the winning condition as $\varphi = \bigwedge_{l \in LSCs(S)} \Box\Diamond (l \text{ is inactive}) \wedge \Box\Diamond m_s = \top$.

6.4. Synthesis and correctness

Once a specification is translated into a game structure, the only step left in order to generate a satisfying system is to find a winning strategy in the game structure. The GR(1) synthesis algorithm from [21] can be used for this. The resulting finite-memory strategy actually stands for a correct implementation of a system.

Roughly, given an environment, the sequence of messages produced by playing the winning strategy against this environment represents the generated trace. The trace is legal since the winning strategy necessarily follows the system transition relation. The trace satisfies the specification since it is generated by a winning strategy, and the winning condition is a direct translation of the satisfaction definition.

Note that the synthesis algorithm of [21] runs in time polynomial in the size of the state space. In our case, the state space is of size exponential in the size of the specification.

7. Results and further examples

We have implemented the language modifications in the Play-Engine tool [12]. The cut-automata generation and formal algorithms were implemented in JTLV, a Java framework for developing formal algorithms [23]. The experiments were performed on a 2.2 GHz Linux PC with 32 GB memory. Screencasts of all the results mentioned here can be viewed online at [26].

Consider the three-story elevator example given in Figs. 1, 2 and 3, and described above. The example consists of 18 LSCs. The state machine of the system synthesized for this specification, for the global justice setting, consists of 81 states, and was synthesized in 54 seconds. The resulting system acts as follows. Following a floor button press, it turns the light in the button on and waits (this is due to the Yield constructs). Once the environment performs a no-op (represented by the user clicking a no-op button), the elevator visits all floors for which the buttons have been pressed. When moving between floors it closes its doors, and it opens them at each required floor before turning off the corresponding light. Only once it has visited all the floors and has turned off all the button lights, will it listen again for environment events. This is due to the eagerness of the synthesized system to reach its global justice goal as early as possible.

A more responsive system may be synthesized by using local justice. The same example, but in the local justice setting, results in a state machine with 4071 states, synthesized in 62 seconds. This system is more responsive, and often upon reaching a floor and turning a light off it yields control and lets the user press more buttons before continuing to other floors (even if more floor lights are on). This is due to the fact that each local justice condition is considered a different goal, and the system plays by setting itself one of the goals at a time. Upon reaching it, for one step, it is less eager and may let the environment play, even if that will cause the system to distance itself from the next goal (as long as the next goal is still reachable).

An extension of this example, which is unrealizable in global justice but realizable in local justice, is given in Fig. 4 and is described in Section 5. This example adds three LSCs for the vent of the elevator, which under certain circumstances (when the user leaves the vent switch on) cannot be simultaneously inactive. This specification, which consists of 21 LSCs, is proven to be unrealizable in the global justice sense in 10 seconds. In local justice, a state machine with 29 890 states is generated in 147 seconds. This system is similar to the local justice system described above, with the addition that as long as the vent switch is on, the vent occasionally moves from horizontal to vertical or vice versa (often more than once in a superstep).

One advantage of synthesis over smart play-out, which is already evident from this example, is the notion of local justice. Smart play-out can support only global justice, since it foresees only one superstep into the future. However, in global justice too, smart play-out might cause violations that synthesis avoids. For example, consider adding the LSC of Fig. 5 to the original specification (without the vent charts). It states that if the environment makes a step (represented by the Yield construct in the prechart) in a state where all button lights are off, then necessarily at that point in time the elevator is on the second floor (both conditions are immediate). In other words, the chart sets the second floor to be a default floor at which the elevator must be before yielding control when all floor lights are off. However, in order to "understand" this, the system must plan more than one superstep ahead, and see that it needs to go to the second floor in order to avoid a violation in the next superstep.

Indeed, for this specification, smart play-out fails (it does not go to the second floor at the end of its superstep, thus reaching a violation in the following superstep), while the synthesized system avoids it by going to the second floor before yielding control back to the environment. This specification, consisting of 19 LSCs, is synthesized in global justice in 95 seconds and uses 120 states. In global justice, the state machine consists of 10 709 states and is synthesized in 83 seconds.

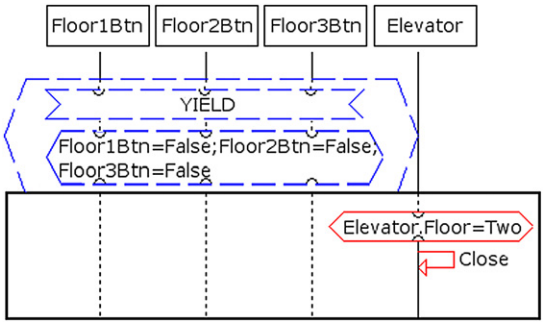


Fig. 5. The DefaultFloor2 LSC. When this chart is added to the specification, smart play-out fails.

Table 1
Summary of experimental results.

Vent LSCs	Default floor LSCs	# LSCs	Global/local justice	Synthesis time	# States
–	–	18	Global Local	54 62	81 4071
+	–	21	Global Local	10 147	unrealizable 29 890
–	+	19	Global Local	95 83	120 10 709
+	+	22	Global Local	12 188	unrealizable 75 742

Finally, combining all of the above, we get a specification consisting of 22 LSCs, which is unrealizable in the global justice sense, a fact that is proven in 12 seconds. In local justice, a state machine consisting of 75 743 states is synthesized in 188 seconds. Table 1 summarizes these experimental results.

8. Related work

The problem of automatic synthesis of a system from its requirements goes back to Church [4], and was first tackled in [3,24]. Synthesis from a temporal logic specification has also been studied heavily, both for closed systems (ones that do not interact with the environment) [19,6], and for open ones [22,1,28]. Synthesis from scenario-based specifications using variants of message sequence charts has also been studied. For a survey of these, see [18].

Synthesis from live sequence charts was first studied in [8], and is tackled there by defining consistency, showing equivalence of consistency and satisfiability (realizability, in this paper), and then synthesizing a satisfying system from the proof of consistency. In [8] a subset of the language is considered, consisting only of messages, but the algorithm is not implemented. A game theoretic approach to synthesis from LSCs, involving a reduction to parity games is described in [2], but the experimental results are described there as negative. Synthesis from LSCs using a reduction to CSP is described in [25].

The basis for the work described here is [16], which uses GR(1) games in a way similar to ours. It focuses on compositional synthesis, and supports only fully spontaneous specifications with only messages, and also only global justice synthesis. The problem is similarly tackled in [15], which supports global justice for a larger subset of the language constructs. Finally, in [17], a variant of the language, inspired by timed automata, is defined, and a system is synthesized from a specification in this variant using timed game automata and the UPPAAL-TIGA tool. The superstep approach is not taken in [17], but rather a totally uncontrollable environment is considered (thus many specifications become unrealizable). Also, only fully spontaneous specifications with only messages and time constraints are considered in [17].

9. Future work

While the experimental results given here are promising, we are still far from being able to handle large and complicated specifications. Finding ways to improve efficiency and scale of the algorithms is crucial in order to render our work practical and usable.

We currently support many of the constructs in the language, e.g. conditions and forbidden messages, but some important features of the language are still missing. Two prominent ones are symbolic instances and time. In symbolic instances, a lifeline may represent a class rather than a concrete object [20]. For example, the three charts stating that each floor button should turn on after being pressed could have been specified in a single chart with a symbolic lifeline for the “floor button” class. A naïve approach to symbolic instances would be to create their concrete instantiations before synthesizing,

but a more efficient approach, taking advantage of the succinctness of symbolic charts would be a great step forward in usability.

Time is supported in play-out and smart play-out by introducing a global clock, with discrete clock tick events for advancing it [11,10]. Support of time in the synthesis algorithm, either in this form or in some other more advanced form (e.g., using the timed automata approach introduced in [17]) is also a major open issue.

The composition algorithm in the recent compositional synthesis work [16] considers fully spontaneous specifications, and also only global justice winning conditions. Extending it to the constructs introduced here is left as future work. Also, the algorithm of [16] is not sound and complete; i.e. for some specifications it might fail to deliver a correct system, or to prove none exists. Finding an efficient sound and complete compositional synthesis algorithm is another important open question.

Acknowledgments

We would like to thank Hillel Kugler, Shahar Maoz, Assaf Marron and Moshe Vardi for their very helpful comments on this work, and on early drafts of the paper. We would also like to thank the referees of this paper, for their very helpful remarks.

References

[1] M. Abadi, L. Lamport, P. Wolper, Realizable and unrealizable specifications of reactive systems, in: Proc. 16th Int. Colloq. on Automata, Languages and Programming (ICALP'89), in: Lecture Notes in Comput. Sci., vol. 372, Springer-Verlag, 1989, pp. 1–17.
[2] Y. Bontemps, P. Heymans, P.Y. Schobbens, From live sequence charts to state machines and back: A guided tour, IEEE Trans. Softw. Eng. 31 (12) (2005) 999–1014.
[3] J. Büchi, L. Landweber, Solving sequential conditions by finite-state strategies, Trans. Amer. Math. Soc. 138 (1969) 295–311.
[4] A. Church, Logic, arithmetic and automata, in: Proc. 1962 Int. Congr. Math., Uppsala, 1963, pp. 23–25.
[5] W. Damm, D. Harel, LSCs: Breathing life into message sequence charts, J. Formal Methods Syst. Des. 19 (1) (2001) 45–80; Preliminary version in: Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Kluwer Academic Publishers, 1999, pp. 293–312.
[6] E. Emerson, E. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, Sci. Comput. Programming 2 (1982) 241–266.
[7] D. Harel, A. Kantor, S. Maoz, On the power of play-out for scenario-based programs, in: D. Dams, V. Hanneman, M. Steffen (Eds.), Concurrency, Compositionality and Correctness: Essays in Honor of Willem-Paul de Roever, in: Lecture Notes in Comput. Sci., vol. 5930, Springer-Verlag, 2010, pp. 207–220.
[8] D. Harel, H. Kugler, Synthesizing state-based object systems from LSC specifications, Internat. J. Found. Comput. Sci. 13 (1) (February 2002) 5–51.
[9] D. Harel, H. Kugler, R. Marelly, A. Pnueli, Smart play-out of behavioral requirements, in: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02), in: Lecture Notes in Comput. Sci., vol. 2517, Springer-Verlag, 2002, pp. 378–398.
[10] D. Harel, H. Kugler, A. Pnueli, Smart play-out extended: Time and forbidden elements, in: Proc. 4th Int. Conf. on Quality Software (QSIC 2004), IEEE Computer Society, 2004, pp. 2–10.
[11] D. Harel, R. Marelly, Playing with time: On the specification and execution of time-enriched LSCs, in: Proc. 10th Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2002), IEEE Computer Society, 2002, pp. 193–202.
[12] D. Harel, R. Marelly, Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine, Springer-Verlag, 2003.
[13] D. Harel, I. Segall, Planned and traversable play-out: A flexible method for executing scenario-based programs, in: Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07), in: Lecture Notes in Comput. Sci., vol. 4424, Springer-Verlag, 2007, pp. 485–499.
[14] ITU, International Telecommunication Union Recommendation Z.120: Message Sequence Charts, Technical report, 1996.
[15] H. Kugler, C. Plock, A. Pnueli, Controller synthesis from LSC requirements, in: Proc. 12th Fundamental Approaches to Software Engineering (FASE'09), in: Lecture Notes in Comput. Sci., vol. 5503, Springer-Verlag, 2009, pp. 79–93.
[16] H. Kugler, I. Segall, Compositional synthesis of reactive systems from live sequence chart specifications, in: Proc. 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09), in: Lecture Notes in Comput. Sci., vol. 5505, Springer-Verlag, 2009, pp. 77–91.
[17] K.G. Larsen, S. Li, B. Nielsen, S. Pusinskas, Scenario-based analysis and synthesis of real-time systems using Uppaal, in: Proc. 13th Conf. on Design, Automation, and Test in Europe (DATE'10), IEEE, 2010, pp. 447–452.
[18] H. Liang, J. Dingel, Z. Diskin, A comparative survey of scenario-based to state-based model synthesis approaches, in: Proc. 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06), ACM, 2006, pp. 5–12.
[19] Z. Manna, R. Waldinger, A deductive approach to program synthesis, ACM Trans. Prog. Lang. Syst. 2 (1980) 90–121.
[20] R. Marelly, D. Harel, H. Kugler, Multiple instances and symbolic variables in executable sequence charts, in: Proc. 17th Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), ACM, 2002, pp. 83–100.
[21] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, in: Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06), in: Lecture Notes in Comput. Sci., vol. 3855, Springer-Verlag, 2006, pp. 364–380.
[22] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proc. 16th Symp. on Principles of Programming Languages (POPL'89), ACM, 1989, pp. 179–190.
[23] A. Pnueli, Y. Sa'ar, L.D. Zuck, JTLV: A framework for developing verification algorithms, in: 22nd Intl. Conf. on Computer Aided Verification (CAV'10), in: Lecture Notes in Comput. Sci., vol. 6174, Springer-Verlag, 2010, pp. 171–174.
[24] M. Rabin, Decidability of second order theories and automata on infinite trees, Trans. Amer. Math. Soc. 141 (1969) 1–35.
[25] J. Sun, J.S. Dong, Synthesis of distributed processes from scenario-based specifications, in: Proc. 13th Int. Symp. on Formal Methods Europe (FM'05), in: Lecture Notes in Comput. Sci., vol. 3582, Springer-Verlag, 2005, pp. 415–431.
[26] http://www.wisdom.weizmann.ac.il/~itais/Synthesis/.
[27] M.Y. Vardi, From verification to synthesis, in: Proc. 2nd Int. Conf. on Verified Software: Theories, Tools, Experiments (VSTTE'08), in: Lecture Notes in Comput. Sci., vol. 5295, Springer-Verlag, 2008, p. 2.
[28] H. Wong-Toi, D. Dill, Synthesizing processes and schedulers from temporal specifications, in: Proc. 2nd Int. Workshop on Computer Aided Verification (CAV'90), 1990, pp. 272–281.