# Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs[⋆,⋆⋆]

David Harel and Itai Segall

The Weizmann Institute of Science, Rehovot, Israel
{david.harel,itai.segall}@weizmann.ac.il

**Abstract.** We introduce a novel approach to the smart execution of scenario-based models of reactive systems, such as those resulting from the multi-modal inter-object language of live sequence charts (LSCs). Our approach finds multiple execution paths from a given state of the system, and allows the user to interactively traverse them. The method is based on translating the problem of finding a superstep of execution into a problem in the AI planning domain, and issuing a known planning algorithm, which we have had to modify and strengthen for our purposes.

## 1 Introduction

Scenario-based modeling appears to be a promising approach to system and software design and development, and has resulted in intensive research efforts in the last few years. One of the most widely used languages for capturing inter-object scenario-based specifications is that of *message sequence charts* (MSCs) proposed by the ITU [21], or its UML variant, *sequence diagrams* [28]. Recently, an extension of MSCs has been proposed, called *live sequence charts* (LSCs) [4]. LSCs are multi-modal charts that distinguish between behaviors that may happen (existential, cold) and those that must happen (universal, hot).[1] The language is highly expressive and can also specify negative behavior, and more, and it has been extended to include, among other things, time, forbidden elements, and symbolic instances (i.e., the ability to talk also about classes, rather than only object instances) [14]. An LSC is divided into two parts, a prechart and a main chart. A prechart is a precondition for the main chart, i.e., if the prechart of an LSC is satisfied, then its main chart must be satisfied as well.

In [15,14], the *play-in/play-out approach* is introduced, in which the user specifies scenarios by playing them in directly from a graphical user interface of the system to be developed. When a scenario is played in, the *Play-Engine* tool

⋆ This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

⋆⋆ This is a somewhat shortened conference version of the paper. The full version [16] can be obtained by emailing one of the authors.

[1] A variant of LSCs has also been defined, called *modal sequence diagrams*, or MSDs, which adheres to the UML 2.0 standard; see [12].

translates it on the fly into an LSC. Play-out is the complementary idea, in which the Play-Engine uses the operational semantics of the language in order to execute a set of LSCs. In this stage, the user again interacts directly with the GUI, and the system responds to each user action with a *superstep*, which is a set of actions, *steps*, that are dictated by the LSC specification as the result of the action.

Here now is a simple example of an LSC specification for a three-story elevator, as shown in Figure 1. LSC Check1, shown in Figure 1(a), states that if the user presses the `Close-Doors` button then the elevator sends the message `Check1` to itself. The specification includes three such LSCs, with messages Check1, Check2, Check3. LSC Goto1, shown in Figure 1(b), states that if the elevator sends `Check1` to itself, then the system checks whether the elevator is not on floor 1 and the `Floor1` button is pressed. If so, the elevator closes its doors, moves to floors 2 and 1 and then opens its doors. The `Floor1` button is turned off as well. According to the LSC semantics, partial order is defined only among locations in the same lifeline (denoted by vertical lines). Therefore, in this example, there is no explicit order between turning off `Floor1` and the other four actions. At the bottom of the figure, two forbidden elements are specified, which state that the elevator may *not* send the message `Check2` or `Check3` to itself, as long as the main chart is active. Similar charts exist in the system specifying movement to floors 2 and 3. Finally, the LSC TestCase of Figure 1(c) is a test case, stating that if the elevator visits floors 3, 2 and 1, in that order, then the `Floor2` button is enabled.

The play-out mechanism described in [14] is naïve, in the sense that at each given point the system selects a single action that is enabled at that point and executes it. This approach might lead to violations in the future, which could have been avoided by selecting the action more wisely from the set of enabled actions. In our example, assume the elevator has visited floors 3 and 2, and is now moving to floor 1 according to the LSC Goto1 of Figure 1(b). Once the `set Floor(1)` message is sent from the elevator to itself, the TestCase LSC in Figure 1(c) becomes active too. Note that the elevator doors are closed at that point in time. Now the system has two options: either open the doors as specified in the Goto1 chart, or enable the `Floor2` button as specified in the TestCase chart. If the system chooses to open the doors first, and only then enables the button, a violation will occur, since TestCase states that after enabling the button the doors must be closed, but the doors are already open and should not be closed again. Had the system chosen to enable the button first, this problem would have been avoided.

One way to tackle this problem is by using *smart play-out* [10,11], in which play-out is translated into a model-checking problem. A model-checker is then handed the claim "no legal superstep exists", and if it delivers a counter-example, which is really a legal superstep, it is then fed into the Play-Engine for execution.

Often it is useful to know more than a single legal superstep, but model checkers are usually unable to provide more than one counter-example. In [14], this issue is addressed in a rather crude way. The first-found superstep is turned by the Play-Engine into a negative (forbidden) scenario. The resulting LSC is then added to the specification and smart play-out is rerun. A different superstep will then be found, if one exists. In this approach, the model-checker must be employed repeatedly for each new superstep to be found, and the specification keeps growing with another chart at each such run.[2]

In this paper we describe a new approach to the play-out problem, termed *planned play-out*, which uses AI planning algorithms and finds many legal supersteps in a single run. As we show, this approach can also be used to support interactive play-out, where the user is allowed to backtrack, and to choose between possible steps, in the quest for an acceptable superstep.

Technically, finding a legal superstep is translated into a planning problem, and a planner is employed in order to solve it. The resulting plans are then translated back into supersteps. We have chosen to use the IPP planner [25,20], an iterative Graphplan-like algorithm. Graphplan planners use a data structure called a *planning graph*, which is a polynomial-sized graph that represents some of the constraints in the planning problem, and which is used to reduce the search for a legal plan. The resulting plan is a partial-order, in the sense that it is divided into timesteps such that two actions in the same timestep are unordered in time. Planning problems, Graphplan and IPP are all discussed in Section 3.

## 2 LSCs

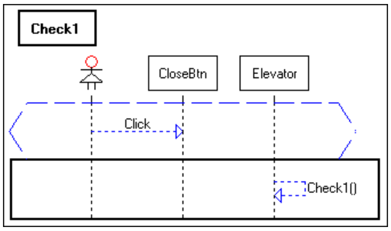
A more detailed description of the LSC language is omitted from this version of the paper; see [16].

## 3 Planning

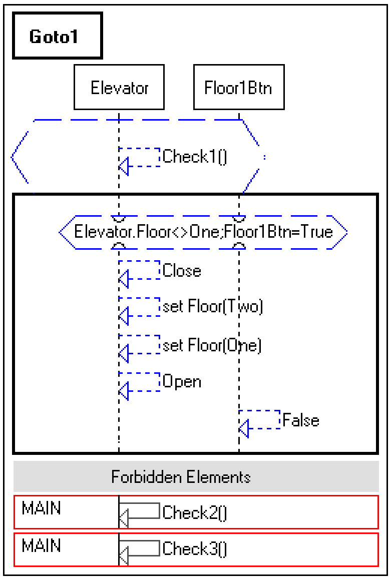
Planning is a field of research central to AI, in which algorithms are designed to generate a list of actions that lead to a predefined goal. Planning is appropriate whenever a number of actions must be performed in a coherent manner to achieve a goal — for example, a robot trying to reach a destination without bumping into walls or getting into dead ends. The algorithms usually consider in advance the consequences of their actions, and decide on the entire plan before performing it.

A *planning problem* typically consists of three inputs: (1) a description of the current state of the world — the *initial conditions*, (2) a description of the desired state after performing the plan — the *goals*, and (3) a set of possible actions — the *domain theory*. An action typically has a *precondition*, describing when it is allowed, and an *effect*, describing the consequence of performing it. An *output plan* is then a multiset of actions from the domain theory, with a partial (or total) order, such that if performed in a manner consistent with the order, starting in a state consistent with the initial conditions, the goals will be achieved.

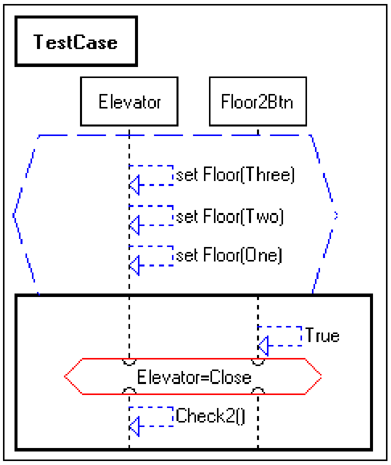
[2] In addition, the new superstep might be very similar to those already found, as the only requirement is for it not to be identical to them.



(a) The Check1 LSC, stating that if the user presses the `Close-Doors` button, the elevator sends the message `Check1` to itself.



(b) The Goto1 LSC, describing how the elevator moves to floor 1.



(c) The TestCase LSC, stating that if the elevator goes to floors 3, 2 and 1, in that order, then the button for floor 2 is pressed.

**Fig. 1.** Sample LSCs from the three-story elevator example

There is a wide range of languages for representing the initial conditions, the goals and the possible actions. We shall focus on the classic STRIPS representation [7], and its extension, ADL [27]. The propositional STRIPS representation describes initial conditions and goals as a conjunction of positive boolean predicates. Actions consist of conjunctive preconditions, add effects (predicates that are true after performing the action) and delete effects (predicates that become false).

An example is given in Figure 2. It is similar to the rocket domain introduced in [29], and involves two objects, A and B, and a bag. The purpose is to move the objects from room R1 to R2, but only the bag can be moved directly between

```
Initial conditions: At(A, R1) ∧ At(B, R1) ∧ At(Bag, R1)
Goal: At(A, R2) ∧ At(B, R2)
Actions:
   Insert(object, room):
         Precondition: At(object, room) ∧ At(Bag, room)
         Effects: In(object) ∧ ¬At(object, room)
   Remove(object, room):
         Precondition: In(object) ∧ At(Bag, room)
         Effects: ¬In(object) ∧ At(object, room)
   Move(from, to):
         Precondition: At(Bag, from)
         effects: ¬At(Bag, from) ∧ At(Bag, to)
```

**Fig. 2.** A STRIPS problem example

the rooms. The initial conditions are that both objects and the bag are in room R1, and the goal is to have both the objects in room R2. The `insert` action represents inserting an object into the bag; its preconditions are that the object and the bag are in the same room, and the effect is that the object is in the bag and no longer in the room. Similarly, the `remove` action removes an object from the bag. Finally, the `move` action moves the bag from one room to the other; its precondition is the bag being in the `from` room, and its effect is removing it from the `from` room, and placing it in the `to` room. A simple legal plan in this example is to insert both objects into the bag, then move the bag, and then remove both objects.

The ADL [27] language is an extension of STRIPS [7], in which conditional and universally quantified effects are added, as well as negative goals. Our main interest in ADL, as we explain later, is in the conditional effects and negative goals.

## 3.1 Main Approaches to Planning

There are three main approaches to solving planning problems in the STRIPS representation: (1) translation into a different problem, e.g., a formula in propositional logic, which is then solved by an external black-box algorithm, e.g., a SAT solver [5,22,23], (2) heuristic-based state-space search [3,**?**], and (3) Graphplan and its descendants.

Tha Graphplan approach [2], which we will use, calls for constructing a polynomial-sized graph in a way that encodes many of the inherent constraints of the problem. This graph and the constraints that arise from it are used in order to significantly reduce the amount of search needed.

Graphplan's main data structure is a *planning graph*, a polynomial-sized graph that represents some of the constraints in the planning problem. Nodes in the planning graph represent either propositions or actions, and are divided into levels depicting timesteps. Two actions in the same timestep have no order between them, but all actions in a specific timestep must occur before those in the following one.

An important part of the graph analysis is detecting pairs of actions that can never appear in a plan in the same timestep, and propositions that can never be true together in the same timestep, and mark them as mutual exclusions (*mutexes*).

### 3.2 Planning Graphs

For lack of space, we omit from this version of the paper the more detailed description of the planning graph, its construction, and its usage in plan extraction; see [16].

### 3.3 IPP

An important feature that ADL adds to STRIPS is the notion of conditional effects. This allows actions to have different effects according to the state in which they are performed. One of the algorithms that support this feature in a Graphplan-like fashion is IPP [25,20]. Negative goals and negative preconditions are also supported by IPP.

## 4 Translating Play-Out into Planning

Our approach to finding legal supersteps is to represent play-out as a planning problem and to solve it using the IPP planner [25,20]. The domain theory is derived from the LSC specification, whereas the initial conditions are derived from the current system state. The goals are independent of the specification, and call simply for finishing the superstep safely. We have enhanced the IPP algorithm so that multiple plans are generated in a single run. These plans are then translated back into supersteps and are fed into the Play-Engine.

Even though we describe here the usage of the IPP planner, the problem is represented in the ADL language, and cen be fed to any planner that supports the relevant subset of ADL. The translation of play-out into planning is done so that a plan exists if and only if a legal superstep exists, and plans can be translated back into supersteps.

Each LSC is represented by an object, and its state is captured by various predicates. An LSC can be either active or not (i.e., its main chart is active or not) — a property represented by the predicate *active*. Each location in the LSC can be enabled or disabled (according to the cut at any given moment), represented by *enabled_loc_X*, for each location X in the LSC.

We assume that only a single copy of each LSC can be active at any given moment, hence a single object per LSC is sufficient. In the future, we plan (no pun intended...) to support multiple running copies by creating multiple objects of the same class, where the class represents the LSC.

### 4.1 Initial Conditions

The initial conditions of the planning problem are derived directly from the initial LSC state. Similarly to the smart play-out mechanism of [10], our translation is invoked after an external event has occurred, in a state in which some LSCs are active. The initial values of all the predicates are therefore determined according to the set of active LSCs and their enabled locations in the given initial configuration.

In the elevator example, suppose the superstep starts in a state where the buttons for floors 1 and 3 are on, the elevator is on floor 2, and the `Close-Doors` button is pressed. This causes charts Check1, Check2 and Check3 to be activated and the superstep to start. The initial condition will therefore be $active(Check1) \wedge active(Check2) \wedge active(Check3)$ and the *enabled* predicates corresponding to the location of the cut are true as well (any predicate not explicitly true is assumed to be false). In addition, the predicates representing the light in button floors 1 and 3 being on are both true.

## 4.2 The Goal

A legal superstep is a sequence of actions that the LSCs take, after which all LSCs are inactive. Hence the goal is:
$(\neg active(o_1)) \wedge \cdots \wedge (\neg active(o_n))$.
This is similar to the way it is done for smart play-out [10].

## 4.3 Actions

Actions represent the possible transitions of the LSC system. Each action stands for a possible step in a superstep, e.g., sending a message, advancing a condition, etc. For each action we formulate its precondition and effect.

There are two types of steps, local and global. Global steps are message sending and receiving, and are global in the sense that many charts must be considered when deciding to perform them. Other steps, such as conditions, are local, in the sense that only a single chart is relevant to them.

**Conditions.** We now describe the translation of conditions. Other constructs, such as if-then-else and unbounded loops are translated in a similar fashion.

Both hot and cold conditions have an action for advancing them, and cold conditions have an additional action for violation. A condition in an LSC is synchronized with one or more lifelines, for each of which there is one location that is *relevant* to the condition, and which must be active in order for the condition to be advanced. Therefore, the precondition for the action of advancing a condition is that all the relevant locations are enabled and that the condition holds. The effect of this action is that the previous locations become disabled and those subsequent to the condition become enabled.

For example, the action representing the hot condition of the doors being closed in the TestCase LSC (which is at location 4 in the Floor2Btn lifeline, and at location 6 in the Elevator lifeline) is:

*Action advance_condition_1_TestCase (TestCase, Elevator)*
  *Precondition:* $enabled\_loc\_Floor2Btn\_4(TestCase) \wedge$
    $enabled\_loc\_Elevator\_6(TestCase) \wedge$
    $Doors\_Closed(Elevator)$

*Effect:* $(\neg enabled\_loc\_Floor2Btn\_4(TestCase)) \wedge$
$(\neg enabled\_loc\_Elevator\_6(TestCase)) \wedge$
$enabled\_loc\_Floor2Btn\_5(TestCase) \wedge$
$enabled\_loc\_Elevator\_7(TestCase)$

One slightly more delicate case is that in which the condition appears at the end of the main chart or the prechart, respectively. In this case, the effect is conditional: (1) if all other lifelines are already in their final location, terminate the LSC or enable the main chart, respectively, and (2) otherwise advance the locations as described above.

As mentioned earlier, cold conditions have an additional action for violation. It has similar preconditions, but if the condition does not hold then if it is in a subchart the effect is to move to locations subsequent to the subchart and otherwise the chart is deactivated and all locations are restarted.

**Messages.** Messages are not local: when one formalizes the action of sending a message (both its precondition and its effect), many charts must be considered. For simplicity, assume each message appears at most once in each LSC, and that the message is synchronous. Asynchronous messages are also supported, but are not described in this version of the paper.

In our translation, each message is transformed into an action. According to the LSC semantics, a message must be triggered, i.e., there must be at least one universal chart that causes it to be sent. Therefore, the precondition for the action of sending a message is that at least one of the charts that contain the message in their main chart is active, and that the message is enabled in it.

Upon sending a message that is enabled in an LSC, the cut is advanced past it. If the sent message appears in an LSC but is not enabled in it, the LSC is assumed to have a cold violation, so it is deactivated and the cut is reset to its initial location. Thus, for each LSC containing the message, there are two conditional effects: one stating that if the message is enabled the cut is forwarded, and the second stating that if the message is not enabled the chart is inactivated and the cut is reset to its initial location. Similarly to the case of conditions, if the message is, or can be, the last action in the main chart (or the prechart), the first effect must be divided into two different conditional effects, according to the locations of the other lifelines.

Note that the only difference between prechart and main chart messages is in the precondition: only main chart locations are considered in the precondition (that is, only they affect the decision of sending a message).

For example, the message `Set floor(2)` sent from the elevator to itself appears in the main charts of *Goto1, Goto2* and *Goto3* (in all of them at location 7 of lifeline *Elevator*) and in the prechart of *TestCase* (though it will never be the last message in that prechart). Its translation is as follows (and similarly for charts *Goto2, Goto3* and *TestCase*):[3]

[3] This translation can be made more efficient. For example, when violating a chart, it is sufficient to disable only main chart locations if the chart is necessarily active. These optimizations are not discussed in this version of the paper.

*Action send_Floor2_Elevator_Elevator(Goto1, Goto2, Goto3, TestCase)*
*Precondition:* $enabled\_loc\_Elevator\_7(Goto1) \vee$
$enabled\_loc\_Elevator\_7(Goto2) \vee$
$enabled\_loc\_Elevator\_7(Goto3)$
*Effect: when* $(enabled\_loc\_Elevator\_7(Goto1):$
$\neg enabled\_loc\_Elevator\_7(Goto1) \wedge$
$enabled\_loc\_Elevator\_8(Goto1))$
*when* $\neg(enabled\_loc\_Elevator\_7(Goto1):$
$enabled\_loc\_Elevator\_3(Goto1)$
$\neg enabled\_loc\_Elevator\_4(Goto1) \wedge \neg enabled\_loc\_Elevator\_5(Goto1) \wedge$
$\neg enabled\_loc\_Elevator\_6(Goto1) \wedge \neg enabled\_loc\_Elevator\_7(Goto1) \wedge$
$\neg enabled\_loc\_Elevator\_8(Goto1) \wedge \neg enabled\_loc\_Elevator\_9(Goto1) \wedge$
$\neg enabled\_loc\_Elevator\_10(Goto1) \wedge$
$enabled\_loc\_Floor2Btn\_2(Goto1) \wedge$
$\neg enabled\_loc\_Floor2Btn\_3(Goto1) \wedge \neg enabled\_loc\_Floor2Btn\_4(Goto1) \wedge$
$\neg enabled\_loc\_Floor2Btn\_5(Goto1))$

A slightly simpler translation can be made if the user chooses not to allow messages to violate main charts (which is sufficient in many cases). In this case, the precondition of the action is that each LSC that contains the message in its main chart is either inactive or else the message is enabled in it. Moreover, the conditional effect that violates the main chart can be skipped.

### 4.4 More Formally

The formal definition of the translation is omitted in this version of the paper; see [16].

## 5 Extending IPP

### 5.1 Forced Mutexes

In some cases, especially when conditional effects are used, not all mutexes inherent in the problem arise from IPP's planning graph. We introduce the notion of *forced mutexes* into the ADL language. These are facts that the user knows should always be mutex, and he/she can therefore explicitly specify them in the problem description, adding them to those discovered by the standard IPP algorithm. We have used this feature to specify that every two locations on the same lifeline should always be mutex. Surprisingly, this small addition results in huge performance improvements, causing problems that caused devastating performance issues in an earlier implementation of our algorithm to be solved within seconds. Further performance issues are discussed in section 6.3.

Mutexes in IPP are implemented using the efficient bit-vector idea introduced in [26]. We adopt these ideas in the implementation of forced mutexes as well. The result of this efficient implementation is that checking whether two facts are mutex is performed in constant time, which does *not* depend on the number of forced mutexes.

### 5.2 Finding Many Plans

IPP is a Graphplan-based planner that supports conditional effects, negative preconditions and negative goals. Like most Graphplan-based planners, IPP is an iterative process that halts once a solution is found. In order to be able to find multiple supersteps we have enhanced the IPP planner to find multiple plans in a single run. This is achieved by changing the halting condition and by adding memoization of positive results.

The new halting condition is as follows. The user states in advance the number of timesteps he/she wishes to continue calculating beyond the shortest plan. IPP will then find the shortest plan as before, but will keep iterating until all plans bounded by the specified length are found.

In order to keep the running time feasible, one must memoize positive results. IPP introduces an efficient memoization mechanism, as described in [17]. This mechanism is used for memoizing negative results (i.e., unachievable subgoals in the backtracking stage), in order to avoid re-checking them. Now that the process is not halted upon finding the first plan, positive results (i.e., achievable subgoals and subplans achieving them) must be memoized as well, since they could be useful for other plans. We have implemented this using the same mechanism as in the original IPP planner, with the addition that subgoals are augmented with all subplans that achieve them.

The output of the extended IPP algorithm is a leveled DAG representing multiple plans that achieve the goals. In it, nodes on level $i$ represent states achievable in $i$ timesteps and edges represent the actions that drive the system from one state to another. If an edge is labeled with more than one action, there is no explicit order between them.

A snippet of the DAG generated in our example can be seen in Figure 3.

## 6 Results

### 6.1 Traversable Play-Out

Once the play-out problem has been translated into a planning one and multiple plans have been found, this information can be used for what we call *traversable play-out* (TPO). In the TPO mode, play-out is performed interactively: at each step, a list of possible actions is given to the user, who is then allowed to choose his/her preferred action. The user can also undo previous steps and explore other paths of execution. Note that only "smart" steps are allowed, i.e., only those that can lead to a successful superstep. If a certain action is enabled at some given time, yet performing it will cause all future runs to fail (i.e., there is no legal way to finish all the LSCs), then the action will *not* be presented as a possible valid action at that time.

In the TPO mode the user can explore various possibilities, find new supersteps not considered earlier, and get a general feeling for all the options the LSC system provides. We feel that these abilities are one of the most significant advantages of our method.

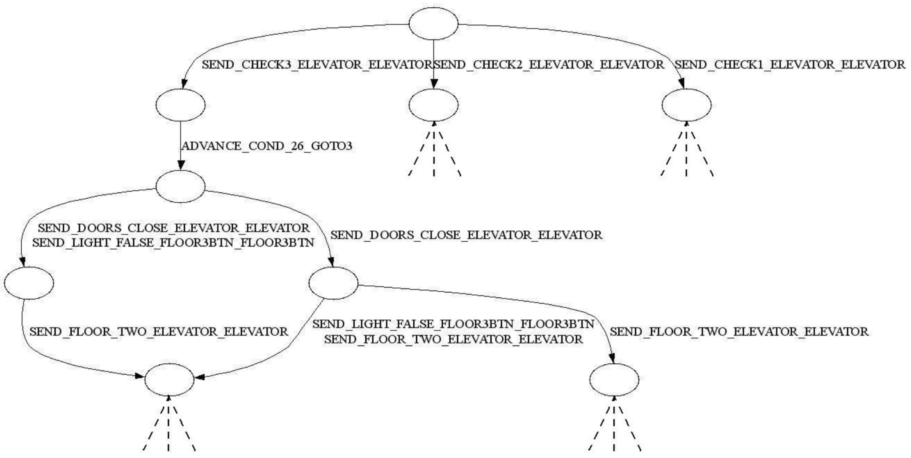

SEND_CHECK3_ELEVATOR_ELEVATOR
SEND_CHECK2_ELEVATOR_ELEVATOR
SEND_CHECK1_ELEVATOR_ELEVATOR
ADVANCE_COND_26_GOTO3
SEND_DOORS_CLOSE_ELEVATOR_ELEVATOR
SEND_LIGHT_FALSE_FLOOR3BTN_FLOOR3BTN
SEND_DOORS_CLOSE_ELEVATOR_ELEVATOR
SEND_FLOOR_TWO_ELEVATOR_ELEVATOR
SEND_LIGHT_FALSE_FLOOR3BTN_FLOOR3BTN
SEND_FLOOR_TWO_ELEVATOR_ELEVATOR
SEND_FLOOR_TWO_ELEVATOR_ELEVATOR

**Fig. 3.** Part of the DAG generated for the elevator example. In the first timestep, the elevator sends Check1, Check2 or Check3 to itself. If Check3 was sent, then the condition in the Goto3 chart is advanced, and then the doors close and the floor3 button turns off (with no explicit order between the last two).

Figure 4 demonstrates the dialog box for TPO with the valid steps at the beginning of the example. A video demonstrating the traversable play-out mode can be downloaded from [19].

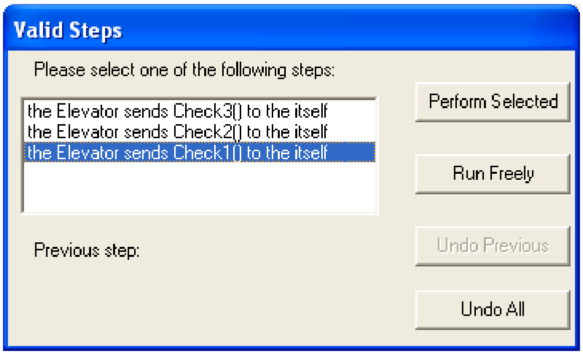
## 6.2 Cut-Queries

Another feature that is part of our method is that of *cut-queries*, which query for locations of cuts during the run. The user can, for example, state that he/she wishes to see runs in which either a specific case in a switch-case statement is chosen, or two specific locations in an LSC are simultaneously enabled. Note that in general such queries cannot be described by simply adding an LSC to the specification, since one cannot directly describe by an LSC the requirement that a specific location should be enabled in another LSC.

Each such query can be described as an AND/OR combination of a set of atomic queries (those describing simultaneous locations of a cut). The query is then translated into the planning domain along with the play-out problem in a way that ensures that only compliant supersteps are found. For each atomic query, a predicate is added, together with an action. The precondition is that the cut is at the correct locations, and the effect is to enable the predicate. The AND/OR combination of the atomic queries can then be added to the goals as a similar combination of the corresponding predicates.

## 6.3 Performance

Planning is known to be NP-complete or worse under most reasonable assumptions [6]. Moreover, Graphplan-based planners usually do not scale up to large



the Elevator sends Check3() to the itself
the Elevator sends Check2() to the itself
the Elevator sends Check1() to the itself

**Fig. 4.** Valid options for the first step in the superstep of the elevator example

domains and large plans. Unfortunately (but not surprisingly), we inherit these limitations.

Still, for the very simple elevator example discussed here, all supersteps (19 timesteps) were found in 300 milliseconds on a standard PC. A different elevator example, in which the "Goto" LSCs are more than twice the size of the ones described in this paper (and all the GUI buttons are takein into account), where 44 timesteps are needed for all plans, the full execution takes about 4 seconds. These results raise the hope that finding all supersteps in much larger specifications will also become eventually feasible. But, of course, the jury is not in on this yet.

It is important to note that finding all supersteps is usually not expected to be a run-time feature, but rather a design-time tool. Therefore, a running time on the scale of a few minutes for a large specification is quite satisfactory.

## 7 Future Work

We have described a framework in which supersteps of LSC specifications are calculated in advance using planning techniques, thus allowing users to interact with the system during play-out. The system lets the user choose steps during the run, but in a way that guarantees completion of the superstep: whatever the user chooses is legal and will lead to a successful completion of all LSCs. We also introduced cut-queries that allow the user to define in advance which of the supersteps are of interest.

A subset of the full LSC language of [14] is currently supported in our implementation of planned and traversable play-out, including synchronous and asynchronous messages, hot and cold conditions, switch-cases, infinite loops and main chart-scoped forbidden elements. Thus, for example, we do not yet support time and symbolic instances.

There are two main issues to be resolved in the context of finding all supersteps: supporting the full LSC language and finding a representation of *all* supersteps.

As mentioned above, only a subset of the features of the LSC language is currently supported, and we have described some assumptions made along the way. This subset should be extended to support the full language, and the assumptions should be removed. In our opinion, some of the constructs will turn out to be easier to support than others. For example, we feel that symbolic instances will not pose a serious problem. A predicate can determine to which object the instance is bound and this predicate should be checked in each relevant action. Similarly, multiple running copies of an LSC can be represented by multiple objects, all derived from the same LSC class. On the other hand, constructs that have a numeric essence, like time, numeric variables, and bounded loops, might be more difficult. This can perhaps be solved using *planning with resources* (see, e.g., [24]), but then the Graphplan solution might not be a good approach.

Our implementation finds many supersteps in a single run, but usually not all of them. In general, there can be infinitely many different supersteps, but these often have a finite representation. For example, a loop that can be iterated any number of times is often an adequate finite representation of infinitely many different supersteps. Future research should address this issue.

## 8 Related Work

In [30], a symbolic simulation engine for LSC specifications is described. It uses constraint logic programming, which in turn uses a form of backtracking in order to find a solution, and can be used to find many supersteps. It is noteworthy that this approach finds full-order supersteps only, hence the backtracking stage will be slower when trying to find interestingly-different supersteps (i.e., ones that differ not only in the order of the steps). Moreover, the approach is used mainly for simulation and finding violations in specifications, whereas our approach is intended for execution and finding many valid supersteps.

There has been an attempt, carried out recently in our group, of representing partial orders derived from LSCs as digraphs [8]. These digraphs were then merged in a way that represented all possible supersteps appropriate for a set of LSCs. The approach, however, was never implemented in the Play-Engine or extended to support more than messages.

In another piece of recent work in our group, LSC specifications are compiled into AspectJ code [13]. This code can then be compiled using any AspectJ Java compiler into a stand-alone application. The current compilation implements naïve play-out, yet stronger and more sophisticated play-out mechanisms can probably be adopted too.

In several projects, model checkers have been augmented to produce multiple counter-examples. In [1], a heuristic BDD-based algorithm for finding multiple behavior paths is introduced, in order to explore and debug hardware design. In [9], a model checker is used iteratively in order to refine an abstracted model.

**Acknowledgments.** We'd like to thank Orna Kupferman for suggesting to us the possible relevance of planning and AI, and Ron Merom for the idea of the elevator example.

## References

1. S. Barner, S. Ben-David, A. Gringauze, B. Sterin and Y. Wolfsthal, "An Algorithmic Approach to Design Exploration", *Proc. International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right* (FME'02), 2002, pp. 146-162.
2. A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis", *Proc. 14th Intl. Joint Conf. on Artificial Intelligence* (IJCAI'95), 1995, pp. 1636-1642 (Extended version appears in *Artificial Intelligence*, **90**(1-2), 1997, pp. 281-300 )
3. B. Bonet and H. Geffner, "HSP: Heuristic Search Planner", *Proc. 4th Intl. Conf. on Artificial Intelligence Planning Systems (AIPS'98) Planning Competition*, Pittsburgh, 1998.
4. W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts", *Formal Methods in System Design*, **19(1):45/22680**, 2001. Preliminary version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (FMOODS'99).
5. M. Ernst, T. Millstein and D. Weld, "Automatic SAT-Compilation of Planning Problems", *Proc. 15th Intl. Joint Conf. on Artificial Intelligence* (IJCAI'97), 1997, pp. 1169-1176.
6. K. Erol, D. S. Nau and V. S. Subrahmanian, "Complexity, Decidability and Undecidability Results for Domain-Independent Planning", *Artificial Intelligence*, **76**(1-2), July 1995, pp. 75-88.
7. R. Fikes and N. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Journal of Artificial Intelligence*, **2**(3/4), 1971, pp. 189-208.
8. Amos Gilboa, MSC Thesis, The Weizmann Institute of Science, 2003, "Finding all Possible Supersteps in LSCs".
9. M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer and M. Y. Vardi, "Multiple-Counterexample guided Iterative Abstraction Refinement: An Industrial Evaluation", *9th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'03), 2003, pp. 176-191.
10. D. Harel, H. Kugler, R. Marelly and A. Pnueli, "Smart Play-Out of Behavioral Requirements", *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design* (FMCAD'02), November 2002, pp. 378-398.
11. D. Harel, H. Kugler and A. Pnueli, "Smart Play-Out Extended: Time and Forbidden Elements", *Proc. 4th Int. Conf. on Quality Software* (QSIC'04), IEEE Computer Society Press, 2004, pp. 2-10.
12. D. Harel and S. Maoz, "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams", *Proc. 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools* (SCESM'06), 2006, pp. 13-20.
13. D. Harel and S. Maoz, "From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ", *14th ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE'14)*, Portland, November 2006.
14. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.
15. D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach", *Software and Systems Modeling* (SoSyM) **2**, 2003, pp. 82-107.
16. D. Harel and I. Segall, "Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs", Technical Report MCS07-01, The Weizmann Institute of Science, 2007.

17. J. Hoffmann and J. Koehler, “A new Method to Query and Index Sets”, *Proc. 16th Intl. Joint Conf. on Artificial Intelligence* (IJCAI’99), 1999, pp. 462-467.
18. J. Hoffmann and B. Nebel, “The FF Planning System: Fast Plan Generation Through Heuristic Search”, *J. Artificial Intelligence Research*, **14**, 2001, pp. 253-302.
19. `http://www.wisdom.weizmann.ac.il/~itais/video/TPO-Example.avi`
20. IPP website, `http://www.informatik.uni-freiburg.de/~koehler/ipp.html.`
21. ITU-TS Recommendation Z.120: “Message Sequence Chart (MSC)”. ITU-TS, Geneva, 1996.
22. H. Kautz and B. Selman, “Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search”, *Proc. 13th National Conf. on Artificial Intelligence* (AAAI’96), Portland, 1996, pp. 1194-1201.
23. H. Kautz and B. Selman, “Blackbox: A New Approach to the Application of Theorem Proving to Problem Solving”, *Workshop on Planning as Combinatorial Search, Artificial Intelligence Planning Systems* (AIPS’98), June 1998, pp. 58-60.
24. J. Koehler, “Planning under Resource Constraints”, *13th biennial European Conf. on Artificial Intelligence* (ECAI’98), 1998, pp. 489-493.
25. J. Koehler, B. Nebel, J. Hoffmann and Y. Dimopoulos, “Extending Planning Graphs to an ADL Subset”, *Proc. 4th European Conf. on Planning* (ECP’97), Springer LNAI, **1348**, 1997, pp. 273-285.
26. D. Long and M. Fox, “Efficient Implementation of the Plan Graph in STAN”, *Journal of Artificial Intelligence Research*, **10**(1999), pp. 87-115.
27. E. P. D. Pednault, ”ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus,” *Proc. 1st Intl. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, 1989, pp. 324-332.
28. UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), `http://www.omg.org`.
29. M. M. Veloso, “Nonlinear problem solving using intelligent casual-commitment”, Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.
30. T. Wang, A. Roychoudhury, R.H.C. Yap and S.C. Choudhary, “Symbolic Execution of Behavioral Requirements”, *Proc. 6th Intl. Symp. on Practical Aspects of Declarative Languages* (PADL’04), 2004, pp. 178-192.