International Journal of Foundations of Computer Science Vol. 13 No. 1 (2002) 5–51
© World Scientific Publishing Company

SYNTHESIZING STATE-BASED OBJECT SYSTEMS FROM LSC SPECIFICATIONS*

DAVID HAREL
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel, 76100
harel@wisdom.weizmann.ac.il

HILLEL KUGLER
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel, 76100
kugler@wisdom.weizmann.ac.il

Received 17 December 2000
Revised 6 April 2001
Communicated by David Harel

ABSTRACT

Live sequence charts (LSCs) have been defined recently as an extension of message sequence charts (MSCs; or their UML variant, sequence diagrams) for rich inter-object specification. One of the main additions is the notion of universal charts and hot, mandatory behavior, which, among other things, enables one to specify forbidden scenarios. LSCs are thus essentially as expressive as statecharts. This paper deals with synthesis, which is the problem of deciding, given an LSC specification, if there exists a satisfying object system and, if so, to synthesize one automatically. The synthesis problem is crucial in the development of complex systems, since sequence diagrams serve as the manifestation of use cases — whether used formally or informally — and if synthesizable they could lead directly to implementation. Synthesis is considerably harder for LSCs than for MSCs, and we tackle it by defining consistency, showing that an entire LSC specification is consistent iff it is satisfiable by a state-based object system, and then synthesizing a satisfying system as a collection of finite state machines or statecharts.

Keywords: Synthesis, Sequence Charts, Statecharts, Object-Oriented Systems, Reactive Systems, Software Engineering.

*Early version appeared as Weizmann Institute Technical Report MCS99-20 October 1999. An abridged version appeared in *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, Lecture Notes in Computer Science, Springer-Verlag, July 2000.

1. Introduction

1.1. Background and motivation

Message sequence charts (MSCs) are a popular means for specifying scenarios that capture the communication between processes or objects. They are particularly useful in the early stages of system development. MSCs have found their way into many methodologies, and are also a part of the UML [30], where they are called **sequence diagrams**. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU [33] (previously called the CCITT).

Damm and Harel [8] have raised a few problematic issues regarding MSCs, most notably some severe limitations in their expressive power. The semantics of the language is a rather weak partial ordering of events. It can be used to make sure that the sending and receiving of messages, if occurring, happens in the right order, but very little can be said about what the system actually does, how it behaves when false conditions are encountered, and which scenarios are forbidden. This weakness prevents sequence charts from becoming a serious means for describing system behavior, e.g., as an adequate language for substantiating the use-cases of [15, 30]. Damm and Harel [8] then go on to define **live sequence charts** (**LSC**s), as a rather rich extension of MSCs. The main addition is **liveness**, or **universality**, which provides constructs for specifying not only possible behavior, but also necessary, or mandatory behavior, both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart. Liveness allows for the specification of "anti-scenarios" (forbidden ones), and strengthens structuring constructs like subcharts, branching and iteration. LSCs are essentially as expressive as statecharts. As explained in [8], the new language can serve as the basis of tools supporting specification and analysis of use-cases and scenarios — both formally and informally — thus providing a far more powerful means for setting requirements for complex systems.

The availability of a scenario-oriented language with this kind of expressive power is also a prerequisite to addressing one of the central problems in behavioral specification of systems: (in the words of [8]) to relate scenario-based inter-object specification with state machine intra-object specification. One of the most pressing issues in relating these two dual approaches to specifying behavior is **synthesis**, i.e., the problem of automatically constructing a behaviorally equivalent state-based specification from the scenarios. Specifically, we want to be able to generate a statechart for each object from an LSC specification of the system, if this is possible in principle. The synthesis problem is crucial in the development of complex object-oriented systems, since sequence diagrams serve to instantiate use cases. If we can synthesize state-based systems from them, we can use tools such as Rhapsody (see [13]) to generate running code directly from them, and we will have taken a most significant step towards going automatically from instantiated use-cases to implementation, which is an exciting (and ambitious!) possibility. See the discussion in the recent [12]. And, of course, we couldn't have said this about the (far easier) problem of synthesizing from conventional sequence diagrams, or MSCs, since their limited expressive power would render the synthesized system too weak to be really useful; in particular, there would be no way to guarantee that the synthesized system would satisfy safety constraints (i.e., that bad things — such as a missile firing with the radar not locked on the target — will not happen).

In this paper we address the synthesis problem in a slightly restricted LSC language, and for an object model in which behavior of objects is described by state machines with synchronous communication. For the most part the resulting state machines are orthogonality-free and flat, but in the last section of the paper we sketch a construction that takes advantage of the more advanced constructs of statecharts.

An important point to be made is that the most interesting and difficult aspects in the development of complex systems stem from the interaction between different features, which in our case is modeled by the requirements made in different charts. Hence, a synthesis approach that deals only with a single chart — even if it is an LSC — does not solve the crux of the problem.

The paper is organized as follows. Section 2 introduces the railcar system of [13] and shows how it can be specified using LSCs. This example will be used throughout the paper to explain and illustrate our main ideas. Section 3 then goes on to explain the LSC semantics and to define when an object system satisfies an LSC specification. In Section 4 we define the **consistency** of an LSC specification and prove that consistency is a necessary and sufficient condition for satisfiability. We then describe an algorithm for deciding if a given specification is consistent. The synthesis problem is addressed in Section 5, where we present a synthesis algorithm that assumes fairness. We then go on to show how this algorithm can be extended to systems that do not guarantee fairness. (Lacking fairness, the system synthesized does not generate the most general language as it does in the presence of fairness.) In Section 6 we outline an algorithm for synthesizing statecharts, with their concurrent, orthogonal state components.

1.2. Related work

As far as the limited case of classical message sequence charts goes, there has been quite some work on synthesis from them. This includes the SCED method [18, 19] and synthesis in the framework of ROOM charts [22]. Other relevant work appears in [28, 4, 2, 7, 16, 32, 23]. In addition, there is the work described in [21], which deals with LSCs, but synthesizes from a single chart only: an LSC is translated into a timed Büchi automaton (from which code can be derived).

In addition to synthesis work directly from sequence diagrams of one kind or another, one should realize that constructing a program from a specification is a long-known general and fundamental problem. There has been much research on constructing a program from a specification given in temporal logic (e.g., [25]).

We now provide brief descriptions of some of these efforts.

1.2.1. ROOM charts

The message sequence chart language used in [22] contains basic (existential) MSCs, called bMSCs, and high level MSCs (hMSCs). The hMSCs provide operators for describing the composition and hierarchical arrangement of bMSCs. The semantics assumes mutually exclusive charts: during a run, the system is always in exactly one bMSC.

The synthesis in [22] generates both structural and behavioral components of ROOM models [29]. The structural components are actors, protocols, ports and bindings, and the behavioral components are a variant of statecharts with no concurrency, i.e., no orthogonal components. The mutual exclusion semantics between the charts allows a fairly simple synthesis algorithm.

One of the questions addressed in [22] is how many events should be executed during a single transition. Two main synthesis algorithms are developed according to the answer to this question, one for maximum traceability and one for maximum progress. In the former case each actor has a state for each bMSC, and each such state has substates corresponding to the local state of the chart. The number of events executed during one transition here is relatively small, since events can be sent only if they appear before the end of the bMSC, and moving to a new bMSC must be done by an additional transition. This approach produces ROOMcharts with many states, but the simple traceability to the structure of the MSCs contains important high level design information.

In the maximum progress approach, events are sent during a single transition until the next received event is reached, possibly in a different bMSC. The synthesized ROOMcharts in this approach are flat state machines with a potentially smaller number of states in comparison to the maximum traceability approach. In this approach, the state machines of the different actors will not have identical structure (which is unlike the situation in the maximum traceability approach).

Here, when synthesizing a state machine for an object O the information used by the synthesis algorithm involves only the set of messages directly relevant to O, i.e., those in which O is either the sender or the receiver. We will show later that when the specification language allows both existential and universal quantification and does not assume mutual exclusion semantics, like in LSCs, this projection information is not enough.

1.2.2. SCED

In the SCED project [18, 19], the semantics of the MSC language used allows overlapping of scenarios: the system can be in different scenarios at the same time. This assumption makes the synthesis more difficult than in the case of mutually exclusive MSCs, and the work is thus of special interest. The synthesis algorithm produces a state machine for an object selected by the user, and it is basically an application of an early algorithm of Biermann and Krishnaswamy that deals with constructing programs from example computations [6].

The work in [6] describes a setting in which the user performs example computations that are stored by the system. The system then automatically synthesizes the shortest (in the number of states) possible program capable of executing the observed examples. Furthermore, it is proved in [6] that if the user has a program P in mind, and shows the system example execution traces $T_1, T_2, \ldots$ from any enumeration of the executions of P, after a finite number of examples the system can synthesize a program P_0 capable of executing all the traces of P. This is known as *identification in the limit* in learning theory.

The SCED synthesis algorithm of [18, 19] adapts the algorithm of [6] to the case of synthesizing state machines from standard (existential) message sequence charts. The MSC language from which the synthesis takes place has been extended with conditional and repetition constructs, subscenarios, assertions, actions and states.

A limitation in SCED is the constraint that requires the synthesized state machine to be deterministic. In some situations this causes synthesizing state machines with disjoint sets of states. An additional problem is that the initial state is not fully defined by the synthesis algorithm, which can lead to underspecified state machines that are not executable. For synthesized state machines with disjoint sets of states it is not possible to define an initial state without violating determinicity.

SCED also lacks the ability to express partial traces in the message sequence chart language, by constraining only some of the events relevant to an object but not others, letting any event not listed as participating be free to occur at any time. This is important in order to support appropriate levels of abstraction, but it makes the synthesis problem harder.

Like in ROOM, here too the information used by the synthesis algorithm is only those messages relevant directly to the object in question, which is not sufficient for LSCs.

1.2.3. Verification of timing diagrams

Timing diagrams [5, 1] constitute a graphical specification language especially appropriate for the description of hardware designs. In [28] timing diagrams and their semantics are formally defined based on a translation into temporal logics. The timing diagrams allow to express partial order of events as well as causality constraints. In fact, they were one of the driving motivations behind the LSCs of [8]. It is shown in [28] that the resulting type of formula has an efficient model checking procedure.

1.2.4. Model checking of MSCs

The problem of verifying whether an MSC or an HMSC (hierarchical MSC) satisfies a temporal requirement given by an automaton is studied in [4]. The complexity of different cases is derived for synchronous and asynchronous interpretations. For the MSC case, an automaton with states corresponding to the cuts is constructed and a bound on the size of the automaton is given. Interestingly, this construction and the one we have in Section 4.2 were carried out independently, and have turned out to be very similar.

1.2.5. Interaction interfaces

Interaction interfaces [7] are used to formally specify the interaction between two or more components that co-operate as subsystems of a distributed system. Their format uses predicates to characterize sets of interaction histories. It is shown in [7] how to derive component specifications from a general interaction interface specification (though for liveness properties some external intervention is required to assign responsibilities to specific components). The issue of realizability of components is also studied in that paper. The same authors also co-designed a synthesis algorithm from MSCs to state machines, appearing in [16].

1.2.6. Synthesis from temporal logic

The early work on this kind of synthesis considered *closed systems*, that do not interact with the environment [24, 10]. In this case a program can be extracted from a constructive proof that the formula is satisfiable. This approach is not suited to synthesizing *open systems* that interact with the environment, since satisfiability implies the existence of an environment in which the program satisfies the formula, but the synthesized program cannot restrict the environment. Later work in [25, 26, 3, 31] dealt with the synthesis of open systems from linear temporal logic specifications. The realizability problem is reduced to checking the nonemptiness of tree automata, and a finite state program can be synthesized from an infinite tree accepted by the automaton. The problems of realizability checking and synthesis from linear temporal logic are shown to be 2EXPTIME-complete. Work on synthesis from branching temporal logics, based on alternating tree automata [20], show that the synthesis problems for CTL and CTL* are EXPTIME and 2EXPTIME complete, respectively.

In [27], synthesis of a distributed reactive system is considered. Given an architecture — a set of processors and their interconnection scheme — a solution to the synthesis problem yields finite state programs, one for each processor, whose joint behavior satisfies the specification. It is shown in [27] that the realizability of a given specification over a given architecture is undecidable. Previous work assumed the easy architecture of a single processor, and then realizability was decidable. In our work, an object of the synthesized system can share all the information it has with all other objects, so the undecidability results of [27] do not apply here.

Another important approach discussed in [27] is first synthesizing a single processor program, and then decomposing it to yield a set of programs for the different processors. The problem of finite-state decomposition is an easier problem than realizing an implementation. Indeed, it is shown in [27] that decompositionality of a given finite state program into a set of programs over a given architecture is decidable. The construction we present in Section 5 can be viewed as following parts of this approach by initially synthesizing a global system automaton describing the behavior of the entire system and then distributing it, yielding a set of state machines, one for each object in the system. However, the work on temporal logic synthesis assumes a model in which the system and the environment take turns making moves, each side making one move in its turn. We consider a more realistic model, in which after each move by the environment, the system can make any finite number of moves before the environment makes its next move.

2. An Example

In this section we introduce the railcar system, which will be used throughout the paper as an example to explain and illustrate the main ideas and results. A detailed description of the system appears in [13], while [8] uses it to illustrate LSC specifications. To make this paper self contained and to illustrate the main ideas of LSCs, we now show some of the basic objects and scenarios of the example.

The automated railcar system consists of six terminals, located on a cyclic path. Each pair of adjacent terminals is connected by two rail tracks. Several railcars are available to transport passengers between terminals.

Here now is some of the required behavior, using LSC's. Fig. 1 describes a car departing from a terminal. The objects participating in this scenario are **cruiser**, **car**, **carHandler**. The chart describes the message communication between the objects, with time propagating from top to bottom. The chart of Fig. 1 is universal. Whenever its activation message occurs, i.e., the car receives the message **setDest** from the environment, the sequence of messages in the chart should occur in the following order: the car sends a departure request **departReq** to the car handler, which sends a departure acknowledgment **departAck** back to the car. The car then sends a **start** message to the cruiser in order to activate the engine, and the cruiser responds by sending **started** to the car. Finally, the car sends **engage** to the cruiser and now the car can depart from the terminal.

A scenario in which a car approaches the terminal is described in Fig. 2. This chart is also universal, but here instead of having a single message as an activation, the chart is activated by the prechart shown in the upper part of the figure (in dashed line-style, and looking like a condition, since it is conditional in the cold sense of the word — a notion we explain below): in the prechart, the message **departAck** is communicated between the car handler and the car, and the message **alert100** is communicated between the proximity sensor and the car. If these messages indeed occur as specified in the prechart, then the body of the chart must hold: the car sends the arrival request **arrivReq** to the car handler, which sends an arrival acknowledgment **arrivAck** back to the car.

Figs. 3 and 4 are existential charts, depicted by dashed borderlines. These charts describe two possible scenarios of a car approaching a terminal: stop at terminal and pass through terminal, respectively. Since the charts are existential, they need not be satisfied in all runs; it is only required that for each of these charts the system has at least one run satisfying it. In an iterative development of LSC specifications, such existential charts may be considered informal, or underspecified, and can later be transformed into universal charts specifying the exact activation message or prechart that is to determine when each of the possible approaches happens.

The simple universal chart in Fig. 5 requires that when the proximity sensor

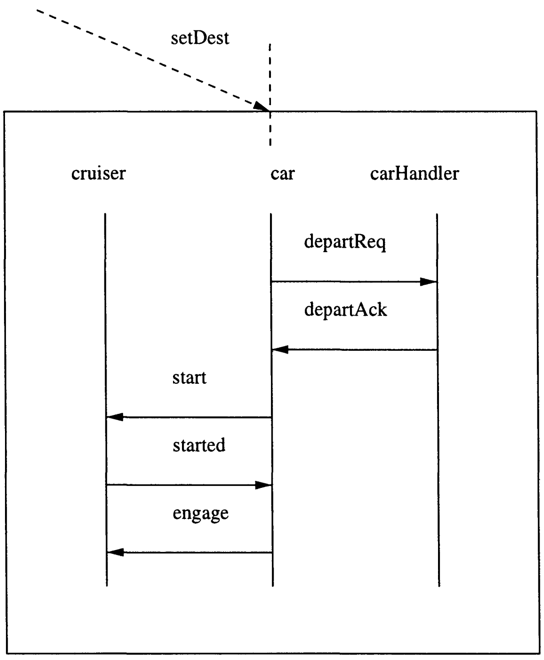


Fig. 1. Perform departure.

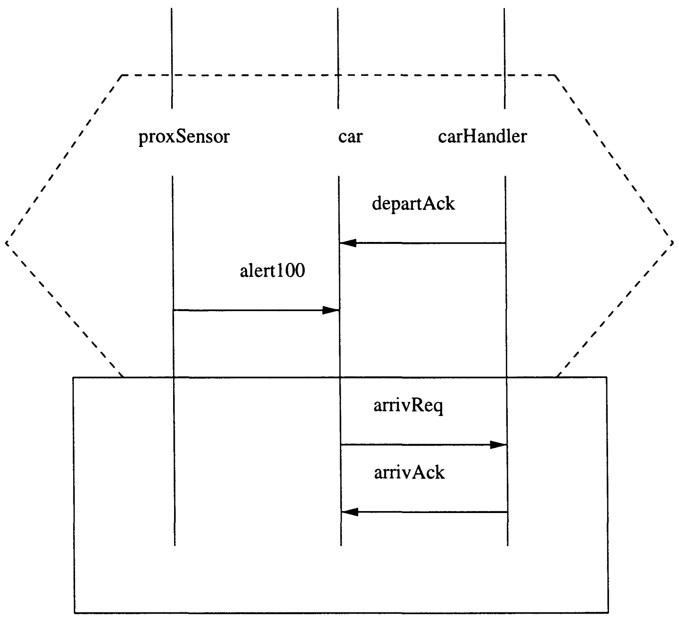


Fig. 2. Perform approach.

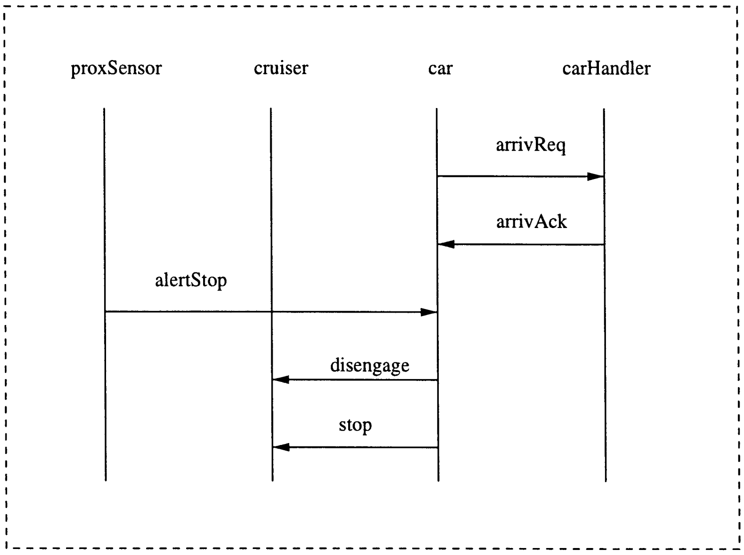


Fig. 3. Stop at terminal.

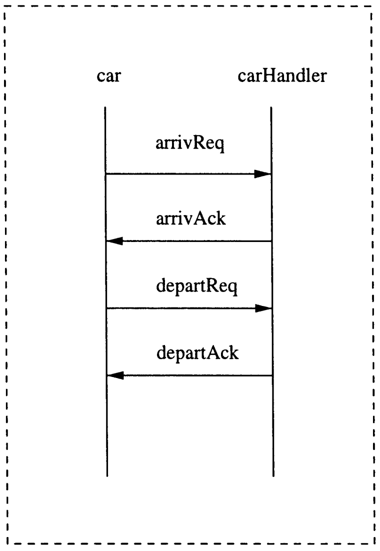


Fig. 4. Pass through terminal.

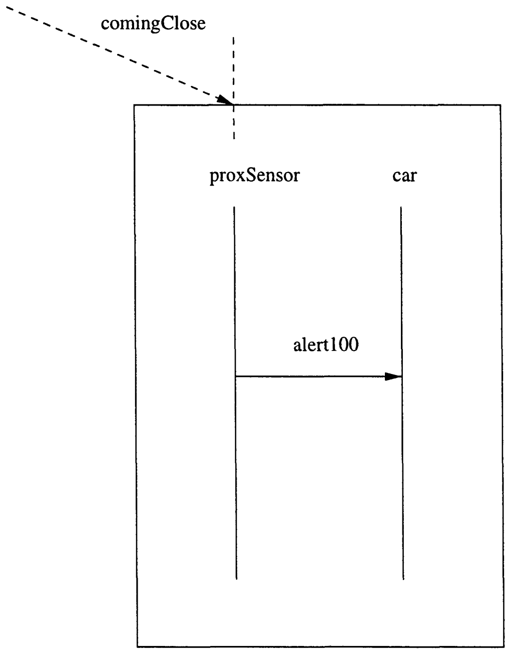


Fig. 5. Coming close to terminal.

receives the message **comingClose** from the environment, signifying that the car is getting close to the terminal, it sends the message **alert100** to the car. This prevents a system from satisfying the chart in Fig. 2 by never sending the message **alert100** from the proximity sensor to the car, so that the prechart is never satisfied and there is no requirement that the body of the chart hold.

The set of charts in Figs. 1–5 can be considered as an LSC specification for (part of) the railcar system. Our goal in this paper is to develop algorithms to decide, for any such specification, if there is a satisfying object system and, if so, to synthesize one automatically. As mentioned in the introduction, what makes our goal both harder and more interesting is in the treatment of a set of charts, not just a single one.

3. LSC Semantics

The semantics of the LSC language[a] is defined in [8], and we now explain some of the basic definitions and concepts of this semantics using the railcar example.

Consider the **Perform Departure** chart of Fig. 1. In Fig. 6 it appears with a labeling of the **locations** of the chart. The set of locations for this chart is thus:

$$\{\langle cruiser, 0\rangle, \langle cruiser, 1\rangle, \langle cruiser, 2\rangle, \langle cruiser, 3\rangle, \langle car, 0\rangle, \langle car, 1\rangle, \langle car, 2\rangle, \langle car, 3\rangle, \langle car, 4\rangle, \langle car, 5\rangle, \langle carHandler, 0\rangle, \langle carHandler, 1\rangle, \langle carHandler, 2\rangle\}$$

[a]A more detailed description of the semantics and the embedding of LSCs into branching temporal logic CTL* appears in the appendix.

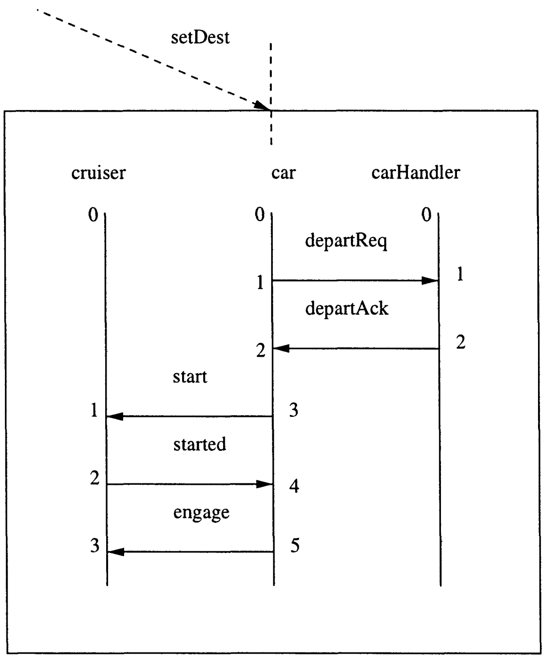


Fig. 6.

The chart defines a partial order $<_m$ on locations. The requirement for order along an instance line implies, for example, $\langle car, 0\rangle <_m \langle car, 1\rangle$. The order induced from message sending implies, for example, $\langle car, 1\rangle <_m \langle carHandler, 1\rangle$. From transitivity we get that $\langle car, 0\rangle <_m \langle carHandler, 1\rangle$.

One of the basic concepts used for defining the semantics of LSCs, and later on in our synthesis algorithms, is the notion of a **cut**. A cut through a chart represents the progress each instance has made in the scenario. Not every "slice", i.e., a set consisting of one location for each instance, is a cut. For example,

$$(\langle cruiser, 1\rangle, \langle car, 2\rangle, \langle carHandler, 2\rangle)$$

is not a cut. Intuitively the reason for this is that to receive the message **start** by the cruiser (in location $\langle cruiser, 1\rangle$), the message must have been sent, so location $\langle car, 3\rangle$ must have already been reached.

The cuts for the chart of Fig. 6 are thus:

$\{(\langle cruiser, 0\rangle, \langle car, 0\rangle, \langle carHandler, 0\rangle), (\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 0\rangle),$
$(\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 1\rangle), (\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 0\rangle, \langle car, 2\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 0\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 1\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 2\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle)$
$(\langle cruiser, 2\rangle, \langle car, 4\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 2\rangle, \langle car, 5\rangle, \langle carHandler, 2\rangle)$
$(\langle cruiser, 3\rangle, \langle car, 5\rangle, \langle carHandler, 2\rangle)\}$

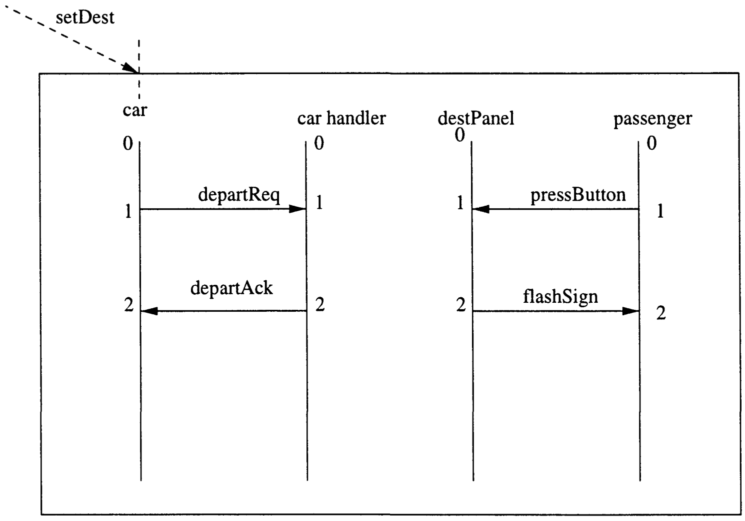


Fig. 7.

The sequence of cuts in this order constitutes a **run**. The **trace** of this run is:

$\{(env, car.setDest), (car, carHandler.departReq), (carHandler, car.departAck), (car, cruiser.start), (cruiser, car.started), (car, cruiser.engage)\}$

This chart has only one run, but in general a chart can have many runs. Consider the chart in Fig. 7. From the initial cut $(0,0,0,0)^b$ it is possible to progress either by the car sending **departReq** to the car handler, or by the passenger sending **pressButton** to the destPanel. Similarly there are possible choices from other cuts. Fig. 8 gives an automaton representation for all the possible runs. This will be the basic idea for the construction of the synthesized state machines in our synthesis algorithms later on. Each state, except for the special starting state s_0, represents a cut and is labeled by the vector of locations. Successor cuts are connected by edges labeled with the message sent. Assuming a synchronous model we do not have separate edges for the sending and receiving of the same message. A path starting from s_0 that returns to s_0 represents a run.

Here are two sample traces from these runs:

$\{(env, car.setDest), (car, carHandler.departReq), (carHandler, car.departAck), (passenger, destPanel.pressButton), (destPanel, passenger.flashSign)\}$

$\{(env, car.setDest), (car, carHandler.departReq), (passenger, destPanel.press Button), (carHandler, car.departAck), (destPanel, passenger.flashSign)\}$

[b]We often omit the names of the objects, for simplicity, when listing cuts.

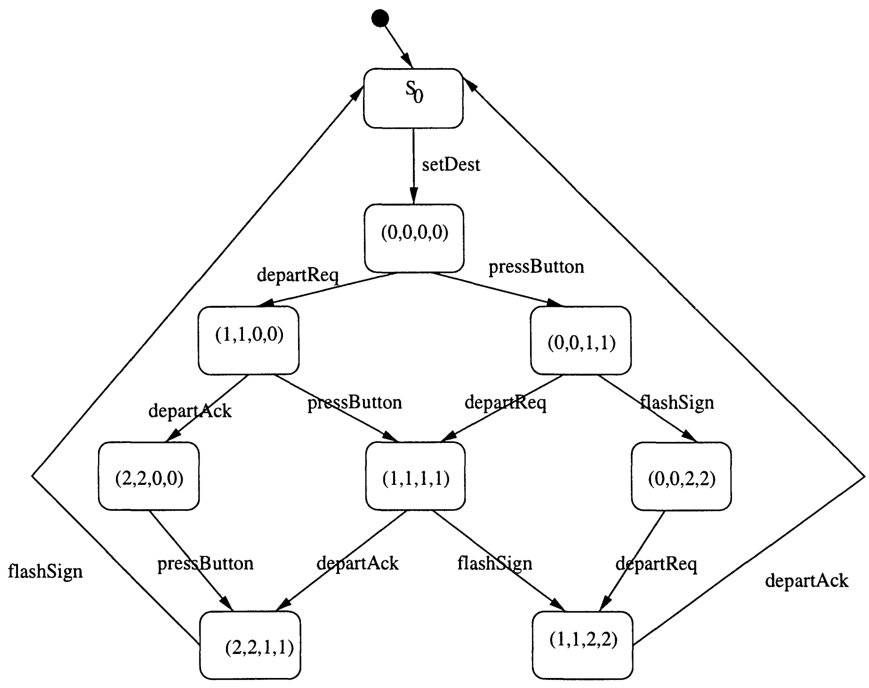


Fig. 8.

As part of the "liveness" extensions, the LSC language enables forcing progress along an instance line. Each location is given a temperature **hot** or **cold**, graphically denoted by solid or dashed segments of the instance line. A run *must* continue down solid lines, while it *may* continue down dashed lines. Formally, we require that in the final cut in a run all locations are **cold**. Consider the **perform approach** scenario appearing in Fig. 9. The dashed segments in the lower part of the car and carHandler instances specify that it is possible that the message **arrivAck** will not be sent, even in a run in which the prechart holds. This might happen in a situation where the terminal is closed or when all the platforms are full.

When defining the languages of a chart in [8], messages that do not appear in the chart are not restricted and are allowed to occur in-between the messages that do appear, without violating the chart. This is an abstraction mechanism that enables concentrating on the relevant messages in a scenario. In practice it may be useful to restrict messages that do not appear explicitly in the chart. Each chart will then have a designated set of messages that are not allowed to occur anywhere except if specified explicitly in the chart; and this applies even if they do not appear anywhere in the chart. A tool may support convenient selection of this message set. Consider the **perform departure** scenario in Fig. 1. By taking its set of messages to include those appearing therein, but also **alert100**, **arrivReq** and **arrivAck**, we restrict these three messages from occurring during the departure scenario, which

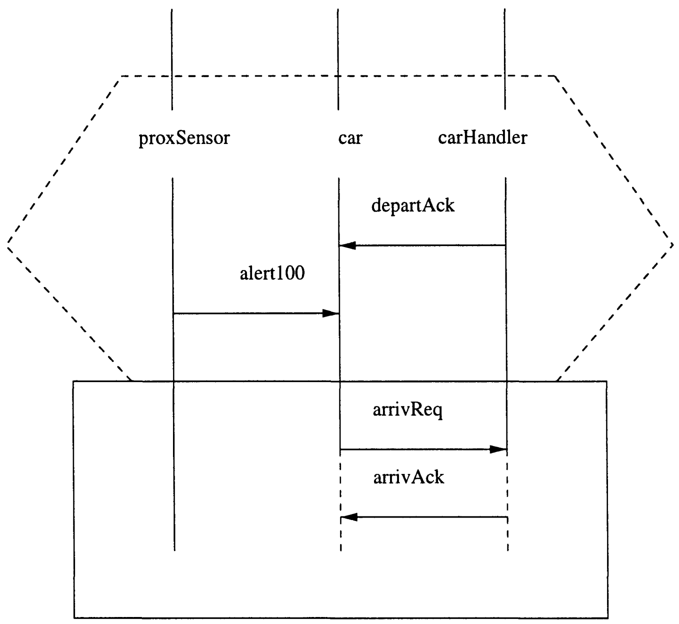


Fig. 9.

makes sense since we cannot arrive to a terminal when we are just in the middle of departing from one.

As in [8], an **LSC specification** is defined as:

$$LS = \langle M, amsg, mod \rangle,$$

where M is a set of charts, and *amsg* and *mod* are the activation messages[c] and the modes of the charts (existential or universal), respectively.

A system **satisfies** an LSC specification if, for every universal chart and every run, whenever the activation message holds the run must satisfy the chart, and if, for every existential chart, there is at least one run in which the activation message holds and then the chart is satisfied. Formally,

Definition 1 *A system S* **satisfies** *the LSC specification $LS = \langle M, amsg, mod \rangle$, written $S \models LS$, if:*

1. $\forall m \in M, \quad mod(m) = universal \Rightarrow \forall \eta \; \mathcal{L}_S^\eta \subseteq \mathcal{L}_m$
2. $\forall m \in M, \quad mod(m) = existential \Rightarrow \exists \eta \; \mathcal{L}_S^\eta \cap \mathcal{L}_m \neq \emptyset$

Here $\mathcal{L}_S^\eta$ is the trace set of object system S on the sequence of directed requests η. We omit a detailed definition here, which can be found, e.g., in [14]. $\mathcal{L}_m$ is the language of the chart m, containing all traces satisfying the chart. We say that an LSC specification is **satisfiable** if there is a system that satisfies it.

[c]In the general case we allow a prechart instead of only a single activation message. However, in this paper we provide the proofs of our results for activation messages, but they can be generalized rather easily to precharts too.

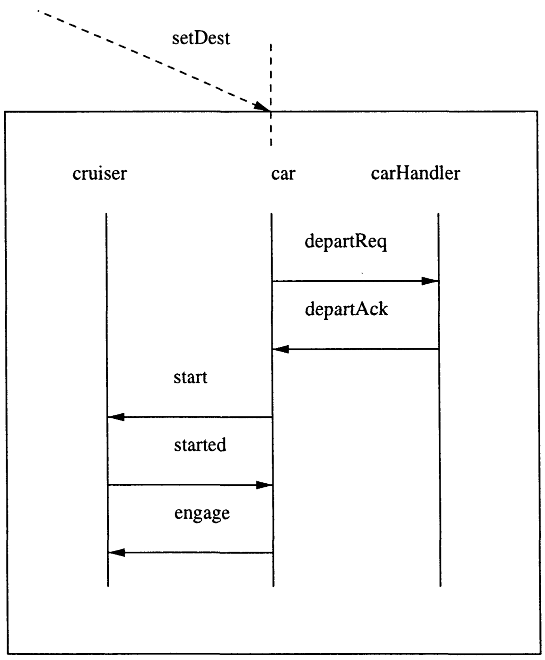


Fig. 10.

4. Consistency of LSCs

Our goal is to automatically construct an object system that is correct with respect to a given LSC specification. When working with an expressive language like LSCs that enables specifying both necessary and forbidden behavior, and in which a specification is a well-defined set of charts of different kinds, there might very well be self contradictions, so that there might be no object system that satisfies it.

Consider an LSC specification that contains the universal charts of Figs. 10 and 11. The message **setDest** sent from the environment to the car activates Fig. 10, which requires that following the **departReq** message, **departAck** is sent from the car handler to the car. This message activates Fig. 11, which requires the sending of **engage** from the car to the cruiser before the **start** and **started** messages are sent, while Fig. 10 requires the opposite ordering. A contradiction.

This is only a simple example of an inconsistency in an LSC specification. Inconsistencies can be caused by such an "interaction" between more than two universal charts, and also when a scenario described in an existential chart can never occur because of the restrictions from the universal charts. In a complicated system consisting of many charts the task of finding such inconsistencies manually by the developers can be formidable, and algorithmic support for this process can help in overcoming major problems in early stages of the analysis.

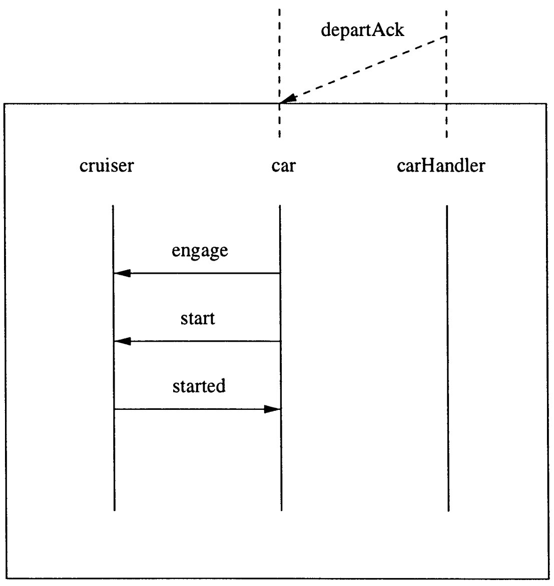


Fig. 11.

4.1. Consistency = Satisfiability

We now provide a global notion of the consistency of an LSC specification. This is easy to do for conventional, existential MSCs, but is harder for LSCs. In particular, we have to make sure that a universal chart is satisfied by *all* runs, from *all* points in time.

We will use the following notation: A_{in} is the alphabet denoting messages sent from the environment to objects in the system, while A_{out} denotes messages sent between the objects in the system.

Definition 2 *An LSC specification $LS = \langle M, amsg, mod \rangle$ is* **consistent** *if there exists a nonempty regular language $\mathcal{L}_1 \subseteq (A_{in} \cdot A_{out}^*)^*$ satisfying the following properties:*

1. $\mathcal{L}_1 \subseteq \bigcap_{m_j \in M,\ mod(m_j)=universal} \mathcal{L}_{m_j}$
2. $\forall w \in \mathcal{L}_1\ \forall a \in A_{in}\ \exists r \in A_{out}^*,\ s.t.\ \ w \cdot a \cdot r \in \mathcal{L}_1.$
3. $\forall w \in \mathcal{L}_1,\ w = x \cdot y \cdot z,\ y \in A_{in} \Rightarrow x \in \mathcal{L}_1.$
4. $\forall m \in M,\ mod(m) = existential \Rightarrow \mathcal{L}_m \cap \mathcal{L}_1 \neq \emptyset.$

The language $\mathcal{L}_1$ is what we require as the set of satisfying traces. Clause 1 in the definition requires all universal charts to be satisfied by all the traces in $\mathcal{L}_1$, Clause 2 requires a trace to be extendible if a new message is sent in from the environment, Clause 3 essentially requires traces to be completed before new messages from the environment are dealt with, and Clause 4 requires existential charts to be satisfied by traces from within $\mathcal{L}_1$.

We now want to prove the first central result of the paper, showing that the consistency of an LSC specification is a necessary and sufficient condition for the existence of an object system satisfying it. As to object systems, we adopt the definitions of an object system appearing in [14], somewhat modified. In [14] a basic computational model for object-oriented designs is presented. It defines the behavior of systems composed of instances of object classes, whose behavior is given by conventional state machines. In our work we assume a single instance of each class during the entire evolution of the system — we do not deal with dynamic creation and destruction of instances. We assume all messages are synchronous and that there are no failures in the system — every message that is sent is received.

We have made several other modifications to [14]:

When considering the languages of words generated by a system, we take the order of requests in the generated behavior to correspond to the order in which the requests were made and not to the order in which they were accomplished (which is the decision made in [14]). This ordering seems more natural when relating systems to LSCs, which specify the order between messages but do not relate explicitly to the accomplishment of requests.

In our setting, systems generate languages over the alphabet $A = (\mathcal{O} \cup env) \times (\mathcal{O}.\Sigma)$, where the letter $(O_j, O_i.\sigma)$ corresponds to object O_j requesting σ from object O_i. The letter $(env, O_i.\sigma)$ corresponds to the environment requesting σ from O_i. In [14], the sender is not taken to be part of the language, so the alphabet is simply $\mathcal{O}.\Sigma$.

We do not allow deadlocks. If a request is made to an object that is in a state with no relevant outgoing transitions, or to an object that is in the midst of making a transition and is therefore suspended, the request is received but it does not affect the receiving object.

We allow objects in the system to have **null transitions**. The semantics prescribes that after entering a new state the transition is completed by taking null transitions, if possible. We add a **fairness requirement**: a null transition that is enabled an infinite number of times is taken an infinite number of times. A **fair cycle** is a loop of states connected by null transitions, which can be taken repeatedly without violating the fairness requirement. We require that the system has no fair cycles, thus ensuring that the system's reactions are finite.

Now comes the result:

Theorem 1 *A specification LS is satisfiable if and only if it is consistent.*

Proof.

($\Rightarrow$) Let the object system S be such that $S \models LS$. We let $\mathcal{L}_S = \cup_{\eta \in A_{in}^*} \mathcal{L}_S^\eta$, and show that $\mathcal{L}_S$ satisfies the four requirements of $\mathcal{L}_1$ in the definition of a consistent specification, Def. 2.

(1) From the definition of an object system it follows that $\mathcal{L}_S$ is regular and nonempty. The system S satisfies the specification LS. Hence, if we set $\mathcal{L} = \bigcap_{m_j \in M, mod(m_j)=universal} \mathcal{L}_{m_j}$, Clause 1 of the definition of satisfaction (Def. 1) implies $\forall \eta\ \mathcal{L}_S^\eta \subseteq \mathcal{L}$. Thus, $\mathcal{L}_S = \cup_\eta \mathcal{L}_S^\eta \subseteq \mathcal{L}$.

(2 and 3) Let $w \in \mathcal{L}_S$. There exists a sequence of directed requests sent by

the environment, $\eta = O^0.\sigma^0 \cdot O^1.\sigma^1 \cdots O^n.\sigma^n$, such that w is the behavior of the system S while reacting to the sequence of requests η. Now, w belongs to the trace set of S on η, so that $w = w_0 \cdot w_1 \cdots w_n$, $w_i \in A^*$, $first(w_i) = (env, O^i.\sigma^i)$, and there exists a sequence of stable configurations $c_0, c_1, ..., c_{n+1}$ such that c_0 is initial and for all $0 \leq i \leq n$, $leads(c_i, w_i, c_{i+1})$. The *leads* predicate is defined in [14]. It describes the reaction of the system to a message sent from the environment to the system that causes a transition of the system from the stable configuration c_i to a new stable configuration c_{i+1}, passing through a set of unstable configurations. The trace describing this behavior is w_i.

As to Clause 2 in Def. 2, the system reaches the stable configuration c_{n+1} at the end of the reactions to η. For any object O_i and request σ, there is a reaction of the system to the directed request $O_i.\sigma$ from the stable configuration c_{n+1}. If we denote by w_{n+1} the word that captures such a reaction, w_{n+1} is in the trace set of S on $O_i.\sigma$ from c_{n+1}, from which we obtain $w \cdot w_{n+1} \in \mathcal{L}_S$.

For Clause 3, assuming that $w = x{\cdot}y{\cdot}z$, $y \in A_{in}$, there exists i with $x = w_0 \cdots w_i$ and therefore $x \in \mathcal{L}_S$.

(4) The system S satisfies the specification LS. Hence, from Clause 2 of Def. 1 we have

$$\forall m \in M, mod(m) = existential \Rightarrow \exists \eta \ \mathcal{L}_S^{\eta} \cap \mathcal{L}_m \neq \emptyset$$

Since $\mathcal{L}_S^{\eta} \subseteq \mathcal{L}_S = \cup_{\eta} \mathcal{L}_S^{\eta}$, we obtain

$$\forall m \in M, mod(m) = existential \Rightarrow \mathcal{L}_S \cap \mathcal{L}_m \neq \emptyset$$

($\Leftarrow$) Let LS be consistent. We have to show that there exists an object system S satisfying LS. To prove this we define the notion of a **global system automaton**, or a **GSA**. We will show that there exists a GSA satisfying the specification and that it can be used to construct an object system satisfying LS.

A GSA A describing a system with objects $\mathcal{O} = \{O_1, ..., O_n\}$ and message set $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ is a tuple $A = \langle Q, q_0, \delta \rangle$, where Q is a finite set of states, q_0 is the initial state, and $\delta \subseteq Q \times \mathcal{B} \times Q$ is a transition relation. Here $\mathcal{B}$ is a set of labels, each one of the form σ/τ, where $\sigma \in A_{in} = (env) \times (\mathcal{O}.\Sigma_{in})$ and $\tau \in A^*_{out} = ((\mathcal{O}) \times (\mathcal{O}.\Sigma_{out}))^*$. Let $\eta = a^0 \cdot a^1 \cdots$ where $a^i \in A_{in}$. The **trace set** of A on η is the language $\mathcal{L}_A^{\eta} \subseteq (A^* \cup A^{\omega})$, such that a word $w = w_0 \cdot w_1 \cdot w_2 \cdots$ is in $\mathcal{L}_A^{\eta}$ iff $w_i = a^i \cdot x^i$, $x^i = x^{i_0} \cdots x^{i_{k_i - 1}} \in A^*_{out}$ and there exists a sequence of states $q^{0_1}, q^{1_1}, ..., q^{1_{k_1}}, q^{2_1}, ..., q^{2_{k_2}}, ...$ with $q^{0_1} = q_0$, and such that for all i, j $(q^{i_j}, /x^{i-1_j}, q^{i_{j+1}}) \in \delta$ and $(q^{i_{k_i}}, a^i/x^{i_0}, q^{i+1_1}) \in \delta$.

The satisfaction relation between a GSA and an LSC specification is defined as for object systems: the GSA A satisfies $LS = \langle M, amsg, mod \rangle$, written $A \models LS$, if $\forall m \in M, mod(m) = universal \Rightarrow \forall \eta \ \mathcal{L}_A^{\eta} \subseteq \mathcal{L}_m$, and $\forall m \in M$, $mod(m) = existential \Rightarrow \exists \eta \ \mathcal{L}_A^{\eta} \cap \mathcal{L}_m \neq \emptyset$.

Since LS is consistent, there exists a language $\mathcal{L}_1$ as in Def. 2. Since $\mathcal{L}_1$ is regular, there exists a DFA $\mathcal{A} = (A, S, s_0, \rho, F)$ accepting it. We may assume that $\mathcal{A}$ is minimal, so all states in S are reachable and each state leads to some accepting state.

From Clause 2 of Def. 2, for every accepting state s of $\mathcal{A}$ and for every $a \in A_{in}$ there exists an outgoing transition with label a leading to a state that is connected to an accepting state by a path labeled $r \in A^*_{out}$. Formally,

$$\forall s \in F \; \forall a \in A_{in} \;\; \rho(s,a) = s' \; \Rightarrow \; \exists r \in A^*_{out} \text{ s.t. } \rho(s',r) \in F$$

From Clause 3 of Def. 2, no nonaccepting states of $\mathcal{A}$ have any outgoing transitions with label $a \in A_{in}$. This is true since if there were such a state $s \notin F$ reachable from the initial state by x, we would have $\rho(s_0, x) = s$ and $\rho(s,a) = s'$, and from s' we can reach an accepting state $\rho(s', z) \in F$. Then $w = x \cdot a \cdot z$ would violate Clause 3.

We have shown that $\mathcal{A}$ has transitions labeled by A_{in} only for accepting states, and for an accepting state there is such a transition for every letter from A_{in}. We now convert $\mathcal{A}$ into an NFA $\mathcal{A}'$ with the same properties, but, in addition, accepting states do not have outgoing transitions labeled A_{out}. This can be done by adding, for each state $s \in F$, an additional state $s' \notin F$. All incoming transitions into s are duplicated so that they also enter s' and all outgoing transitions from s labeled A_{out} are transferred to s'. $\mathcal{A}'$ accepts the same language as $\mathcal{A}$ since it can use nondeterminism to decide if to take a transition to s or s'.

We now transform the automaton $\mathcal{A}'$ into a GSA $\mathcal{B}$ by changing all transitions with a label from A_{out} into null transitions with that letter as an action. All transitions with a label from A_{in} are left unchanged.

We have to show that $\mathcal{B}$ satisfies the specification LS. From the construction of $\mathcal{B}$, we have

$$\mathcal{L}_B = \cup_\eta \mathcal{L}^\eta_B = \mathcal{L}_1$$

From Clause 1 of Def. 2, we have

$$\mathcal{L}_1 \subseteq \bigcap_{m_j \in M, mod(m_j) = universal} \mathcal{L}_{m_j}$$

Hence,

$$\forall m \in M, mod(m) = universal \Rightarrow \mathcal{L}_1 \subseteq \mathcal{L}_m,$$

yielding

$$\forall m \in M, mod(m) = universal \Rightarrow \forall \eta \; \mathcal{L}^\eta_B \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_m$$

This proves Clause 1 of Def. 1.

Now, from Clause 4 of Def. 2, we have

$$\forall m \in M, mod(m) = existential \Rightarrow \mathcal{L}_m \cap \mathcal{L}_1 \neq \emptyset$$

But since $\mathcal{L}_1 = \cup_\eta \mathcal{L}^\eta_B$, this becomes

$$\forall m \in M, mod(m) = existential \Rightarrow \exists \eta \; \mathcal{L}^\eta_B \cap \mathcal{L}_m \neq \emptyset,$$

thus proving Clause 2 of Def. 1.

□

As can be seen in the proof, the basic concept used is the notion of a global system automaton, or a GSA. A GSA describes the behavior of the entire system — the message communication between the objects in the system in response to messages received from the environment. To recap the definition appearing in the proof, a GSA is a finite state automaton with input alphabet consisting of messages sent from the environment to the system (A_{in}), and output alphabet consisting of messages communicated between the objects in the system (A_{out}). The GSA may have null transitions, transitions that can be taken spontaneously without the triggering of a message. We add a fairness requirement: a null transition that is enabled an infinite number of times must be taken an infinite number of times. A fair cycle is a loop of states connected by null transitions, which can be taken repeatedly without violating the fairness requirement. We require that the system has no fair cycles, thus ensuring that the system's reactions are finite.

In the *Consistency* $\Rightarrow$ *Satisfiability* direction of the proof of Theorem 1 we show that it is possible to construct a GSA satisfying the specification. This implies the existence of an object system (a separate automaton for each object) satisfying the specification. Later on, when discussing synthesis, we will show methods for the distribution of the GSA between the objects to obtain a satisfying object system. In section 5.5 we show that the fairness requirement is not essential for our construction — it is possible to synthesize a satisfying object system that does not use null transition and the fairness requirement, although it does not generate the most general language.

4.2. Deciding consistency

It follows from Theorem 1 that to prove the existence of an object system satisfying an LSC specification LS, it suffices to prove that LS is consistent. In this section we present an algorithm for deciding consistency.

A basic construction used in the algorithm is that of a deterministic finite automaton accepting the language of a universal chart. Such an automaton for the chart of Fig. 7 is shown in Fig. 12. The initial state s_0 is the only accepting state. The activation message **setDest** causes a transition from state s_0, and the automaton will return to s_0 only if the messages **departReq, departAck, pressButton** and **flashSign** occur as specified in the chart. Notice that the different orderings of these messages that are allowed by the chart are represented in the automaton by different paths. Each such message causes a transition to a state representing a successor cut. The self transitions of the nonaccepting states allow only messages that are not restricted by the chart. The initial state s_0 has self transitions for message **comingClose** sent from the environment and for all other messages between objects in the system. To avoid cluttering the figure we have not written the messages on the self transitions.

We use the notations and definitions of the formal LSC semantics appearing in the appendix.

For a universal chart m we define the DFA $\mathcal{A} = (A, S, s_0, \rho, F)$, as follows:

- $A = A_{in} \cup A_{out}$ is the alphabet.

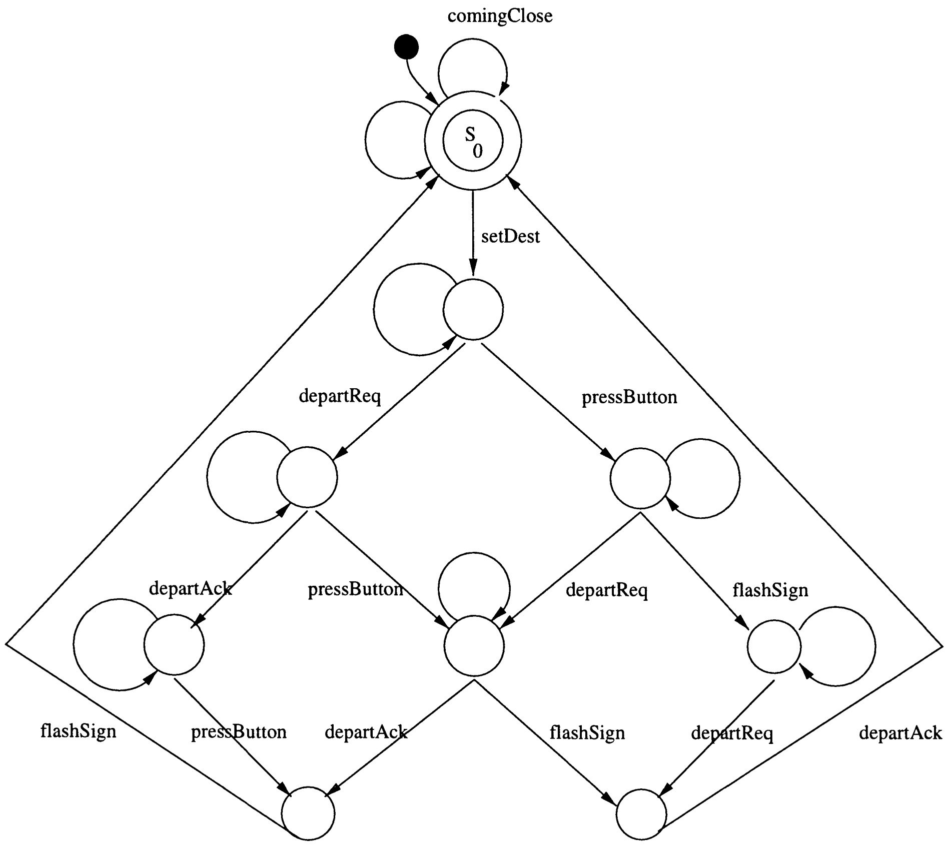

comingClose
S
0
setDest
departReq
pressButton
departAck
pressButton
departReq
flashSign
flashSign
pressButton
departAck
flashSign
departReq
departAck

Fig. 12.

- the set of states S consists of the cuts through m, with an additional state s_0. Thus,
$$S = \{c \mid c \in \mathit{cuts}(m)\} \cup s_0.$$
- assuming the natural mapping f between $(\mathit{dom}(m) \cup \mathit{env}) \times \Sigma \times \mathit{dom}(m)$ to the alphabet A, the transition function ρ is defined as follows:
  - $\rho(s_0, a) = c_0$ if $a = f(\mathit{amsg}(m))$ and c_0 is the initial cut;
  - $\rho(c, a) = c$ if a is not restricted by m;
  - $\rho(c, a) = c'$ if $\mathit{succ}_m(c, <j, l_j>, c'')$ and $\mathit{succ}_m(c'', <j', l_{j'}>, c')$ and $f(\mathit{msg}(m)(<j, l_j>)) = a$ and $<j, l_j>, <j', l_{j'}>$ are send and receive events of the same message and not all locations in c' are cold;
  - $\rho(c, a) = s_0$ if $\mathit{succ}_m(c, <j, l_j>, c')$ and $f(\mathit{msg}(m)(<j, l_j>)) = a$ and all locations in c' are cold;
- the set of accepting states is $F = \{s_0\}$.

Claim: $\mathcal{L}(\mathcal{A}) = \mathcal{L}_m$.

Proof.

$\mathcal{L}(A) \subseteq \mathcal{L}_m$: Assume that $w = w_1 \cdot w_2 \cdots w_n \in \mathcal{L}(A)$, and that $w_i = f(\mathit{amsg}(m))$. We have to show that the chart m is satisfied, i.e., that $\exists i_1, i_2, ..., i_k$ and $\exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}$, such that $i < i_1 < i_2 < ... < i_k$ and $\forall j,\ 1 \leq j \leq k,\ w_{i_j} = v_j$ and $\forall j',\ i \leq j' \leq i_k,\ j' \notin \{i_1, ..., i_k\},\ w_{j'} \notin f(\mathit{Messages}(m))$.

Let $q^0, q^1, ..., q^n$ be the sequence of states that $\mathcal{A}$ goes through while reading w. Clearly, $q^{i-1} = s_0$, otherwise $\mathcal{A}$ will reject when reading w_i since the only transition with $\mathit{amsg}(m)$ is from s_0. Let i_k be the smallest index larger than i such that $q_{i_k} = s_0$. (Such an index exists, since $q^n = s_0$ follows from the acceptance condition of $\mathcal{A}$ and the fact that $w \in \mathcal{L}(\mathcal{A})$.) Now let $i_1, i_2, ..., i_k$ be the ordered list of indices in the sequence $q^i, ..., q^{i_k}$, for which $q_{i_j} \neq q_{i_j - 1}$. From the definition of ρ, the cuts corresponding to the states $q^i, q^{i_1}, ..., q^{i_k}$ are successive, q_i corresponds to the initial cut c_0, and q^{i_k} implies by the definition of ρ a cut in which all locations are cold. Hence, the series of cuts is a run that corresponds to chart m, and which generates the trace $v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}$. For all indices j' with $i \leq j' \leq i_k$ and $j' \notin \{i_1, ..., i_k\}$, we know that $q_{j'} = q_{j'-1}$. Hence, by the definition of ρ, $w_{j'} \notin f(\mathit{Messages}(m))$.

$\mathcal{L}_m \subseteq \mathcal{L}(\mathcal{A})$: This direction is similar, but we have to make use of the assumption that the activation message is not sent before the end of the activity that follows the previous time it was sent.

□

An automaton accepting exactly the runs that satisfy *all* the universal charts can be constructed by intersecting these separate automata. This intersection automaton will be used in the algorithm for deciding consistency. The idea is to start with this automaton, which represents the "largest" regular language satisfying all the universal charts, and to systematically narrow it down in order to avoid states from which the system will be forced by the environment to violate the specification. An the end we must check that there are still representative runs satisfying each of the existential charts.

Here, now is our algorithm for checking consistency:

Algorithm 1

1. *Find the minimal DFA $\mathcal{A} = (A, S, s_0, \rho, F)$ that accepts the language*

$$\mathcal{L} = \bigcap_{m_j \in M,\ mod(m_j)=universal} \mathcal{L}_{m_j}$$

(The existence of such an automaton follows from the construction above.)

2. *Define the sets $Bad_i \subseteq S$, for $i = 0, 1, \ldots$, as follows:*

   $Bad_0 = \{s \in S \mid \exists a \in A_{in},\ s.t.\ \forall x \in A^*_{out}\ \rho(s, a \cdot x) \notin F\}$,

   $Bad_i = \{s \in S \mid \exists a \in A_{in},\ s.t.\ \forall x \in A^*_{out}\ \rho(s, a \cdot x) \notin F - Bad_{i-1}\}$.

   The series Bad_i is monotonically increasing, with $Bad_i \subseteq Bad_{i+1}$, and since S is finite it converges. Let us denote the limit set by Bad_{max}.

3. *From $\mathcal{A}$ define a new automaton $\mathcal{A}' = (A, S, s_0, \rho, F')$, where the set of accepting states has been reduced to $F' = F - Bad_{max}$*

4. *Further reduce $\mathcal{A}$, by removing all transitions that lead from states in $S - F'$, and which are labeled with elements of A_{in}. This yields the new automaton $\mathcal{A}''$.*

5. *Check whether $\mathcal{L}(\mathcal{A}'') \neq \emptyset$ and whether, in addition, $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') \neq \emptyset$ for each $m_i \in M$ with $mod(m_i) = existential$. If both are true output* YES*; otherwise output* NO.

Theorem 2 *Algorithm 1 is correct: given a specification LS, it terminates and outputs* YES *iff LS is consistent.*

Proof.

($\Rightarrow$) Assume the algorithm outputs YES. We will show that $\mathcal{L}(\mathcal{A}'')$ satisfies the four properties of the language $\mathcal{L}_1$ in the consistency definition, Def. 2.

(1) We constructed $\mathcal{A}$ to be the intersection of runs satisfying the universal charts, so $\mathcal{L}(\mathcal{A}) = \bigcap_{m_j \in M, mod(m_j)=universal} \mathcal{L}_{m_j}$. When moving from $\mathcal{A}$ to $\mathcal{A}''$ we make accepting states nonaccepting and remove transitions. This cannot add words to the language only remove words. Hence, $\mathcal{L}(\mathcal{A}'') \subseteq \mathcal{L}(\mathcal{A})$.

(2) Assume $w \in \mathcal{L}(\mathcal{A}'')$ and let $s = \rho(s_0, w)$. Since w is accepted by $\mathcal{A}''$, which is deterministic, it must be that $s \in F'$. Therefore, $s \notin Bad_{max}$. In particular, $s \notin Bad_0$; so $\forall a \in A_{in}\ \exists x \in A^*_{out}$, s.t. $\rho(s, a \cdot x) \in F - Bad_{max}$. Disconnecting transitions labeled from A_{in} does not change the existence of x, since $x \in A^*_{out}$. This shows that $\forall w \in \mathcal{L}(\mathcal{A}'')\ \forall a \in A_{in}\ \exists x \in A^*_{out}$, s.t. $w \cdot a \cdot x \in \mathcal{L}(\mathcal{A}'')$

(3) Assume $w \in \mathcal{L}(\mathcal{A}'')$, $w = x \cdot y \cdot z$ and $y \in A_{in}$. Let $s = \rho(s_0, x)$. Then there is a transition labeled y from state s, since otherwise w will be rejected. From the construction of $\mathcal{A}''$ we disconnected all transitions with labels in A_{in} from the states in $S - F'$, so it must be that $s \in F'$. Therefore, x is accepted by $\mathcal{L}(\mathcal{A}'')$.

(4) This property follows directly, since the algorithm outputs YES if for every $m_i \in M$ such that $mod(m_i) = existential$ we have $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') \neq \emptyset$.

($\Leftarrow$) We will show that for every $\mathcal{L}_1$ satisfying the four clauses of Def. 2, $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A}'')$. Then, the fact that for every $m_i \in M$ such that $mod(m_i) = existential$ we have $\mathcal{L}_{m_i} \cap \mathcal{L}_1 \neq \emptyset$, will imply that also $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') \neq \emptyset$.

Let $\mathcal{L}_1$ be any language satisfying the clauses of Def. 2. From Clause 1, $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A})$. We will first show that for every w with $\rho(s_0, w) \in Bad_{max}$, we have $w \notin \mathcal{L}_1$. Then $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A}')$, since $\mathcal{A}'$ was created from $\mathcal{A}$ by turning states in Bad_{max} into nonaccepting states.

Let $s = \rho(s_0, w)$ and $s \in Bad_{max}$, and assume, for the sake of obtaining a contradiction, that $w \in \mathcal{L}_1$. The fact that $s \in Bad_{max}$ for $Bad_{max} = Bad_i$ implies that $\exists a \in A_{in}\ \forall x \in A^*_{out}\ \rho(s, a \cdot x) \notin F - Bad_{i-1}$. We assumed $w \in \mathcal{L}_1$; hence, from Clause 2 we know that $\exists r \in A^*_{out}$ s.t. $w \cdot a \cdot r \in \mathcal{L}_1$, form which we get $\rho(s, a \cdot r) \in F$, since $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A})$. Now, from $\rho(s, a \cdot r) \in F$ and $\rho(s, a \cdot r) \notin F - Bad_{i-1}$ we get $\rho(s, a \cdot r) \in Bad_{i-1}$. For $w^1 = w \cdot a \cdot r$, $w^1 \in \mathcal{L}_1$, $\rho(s_0, w^1) \in Bad_{i-1}$. In the same way, we can find $w^2, w^3, ..., w^i$, so that for every j, $w^j \in \mathcal{L}_1$, $\rho(s_0, w^j) \in Bad_{i-j}$. For w_i we get that $w_i \in \mathcal{L}_1$ and $\rho(s_0, w_i) \in Bad_0$. Let $s' = \rho(s_0, w^i)$, $s' \in Bad_0$ therefore, $\exists a \in A_{in} \forall x \in A^*_{out}\ \rho(s', a \cdot x) \notin F$ but $w^i \in \mathcal{L}_1$, so that $\exists r \in A^*_{out}$ s.t. $w^i \cdot a \cdot r \in \mathcal{L}_1$. Since $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A})$, we obtain $\rho(s', a \cdot r) \in F$, which is a contradiction.

So, we have showed that $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A}')$, and we still have to show that $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A}'')$. Let $w \in \mathcal{L}_1$ and assume, again for contradiction, that $w \notin \mathcal{L}(\mathcal{A}'')$. Since $w \in \mathcal{L}_1$ and $\mathcal{L}_1 \subseteq \mathcal{L}(\mathcal{A}')$, we have $w \in \mathcal{L}(\mathcal{A}'')$, so that $\rho(s_0, w) \in F'$. Now, it must be the case that $\mathcal{A}'$, when reading w, uses one of the transitions that were removed when we went from $\mathcal{A}'$ to $\mathcal{A}''$, since otherwise $\mathcal{A}''$ would accept w. Let $w = x \cdot y \cdot z$, with $y \in A_{in}$ and $s = \rho(s_0, x)$, and assume that the transition $\rho(s, a)$ was removed in $\mathcal{A}''$. From Clause 3 of Def. 2 $x \cdot y \cdot z \in \mathcal{L}_1$ and $y \in A_{in}$ implies $x \in \mathcal{L}_1$, so $x \in \mathcal{L}(\mathcal{A}')$ and $s = \rho(s_0, x) \in F'$, contradicting the fact that the transition $\rho(s, a)$ was removed, since transitions were removed only from states in $S - F'$.

□

In case the algorithm answers YES, the specification is consistent and it is possible to proceed to automatically synthesize the system, as we show in the next section. However, for the cases where the algorithm answers NO, it would be very helpful to provide the developer with information about the source of the inconsistency. Step 5 of our algorithm provides the basis for achieving this goal. Here is how.

The answer is NO if $\mathcal{L}(\mathcal{A}'') = \emptyset$ or if there is an existential chart m_i such that $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') = \emptyset$. In the second case, this existential chart is the information we need. The first case is more delicate: there is a sequence of messages sent from the environment to the system (possibly depending on the reactions of the system) that eventually causes the system to violate the specification. Unlike verification against a specification, where we are given a specific program or system and can display a specific run of it as a counter-example, here we want to **synthesize** the object system so we do not yet have any concrete runs. A possible solution is to let the supporting tool play the environment and the user play the system, with the aim of locating the inconsistency. The tool can display the charts graphically and highlight the messages sent and the progress made in the different charts. After each message sent by the environment (determined by the tool using the information obtained in the consistency algorithm), the user decides which messages are sent between the objects in the system. The tool can suggest a possible reaction of the system, and allow the user to modify it or choose a different one. Eventually, a universal chart will be violated, and the chart and the exact location of this violation can be displayed.

5. Synthesis of FSMs from LSCs

We now show how to automatically synthesize a satisfying object system from a given consistent specification. We first use the algorithm for deciding consistency (Algorithm 1), relying on the equivalence of consistency and satisfiability (Theorem 1) to derive a global system automaton, a GSA, satisfying the specification. Synthesis then proceeds by distributing this automaton between the objects, creating a desired object system.

The synthesis is demonstrated on our example, taking the charts in Figs. 1–5 to be the required LSC specification. For the universal charts, Figs. 1, 2 and 5, we assume that the sets of restricted messages (those not appearing in the charts) are { **alertStop**, **alert100**, **arrivReq**, **arrivAck**, **disengage**, **stop** }, { **departReq**, **start**, **started**, **engage** } and { **departAck**}, respectively.

Figs. 13, 14 and 15 show the automata for the **perform departure**, **perform approach** and **coming close** charts, respectively. Notice that in Fig. 14 there are two accepting states s_0 and s_1, since we have a prechart with messages **departAck** and **alert100** that causes activation of the body of the chart. To avoid cluttering the figures we have not written the messages on the self transitions. For nonaccepting states these messages are the non-restricted messages between objects in the system, while for accepting states we take all messages that do not cause a transition from the state, including messages sent by the environment.

The intersection of the three automata of Figs. 13, 14 and 15 is shown in Fig. 16. It accepts all the runs that satisfy all three universal charts of our system.

The global system automaton (GSA) derived from this intersection automaton is shown in Fig. 17. The two accepting states have as outgoing transitions only messages from the environment. This has been achieved using the techniques described in the proof of Theorem 1. Notice also the existence of runs satisfying each of the existential charts. We have used the path extraction methods of Section 5.5 to retain these runs.

After constructing the GSA, the synthesis proceeds by distributing the automaton between the objects, creating a desired object system. To illustrate the distribution we focus on a subautomaton of the GSA consisting of the states q_0, q_1, q_2, q_3, q_4

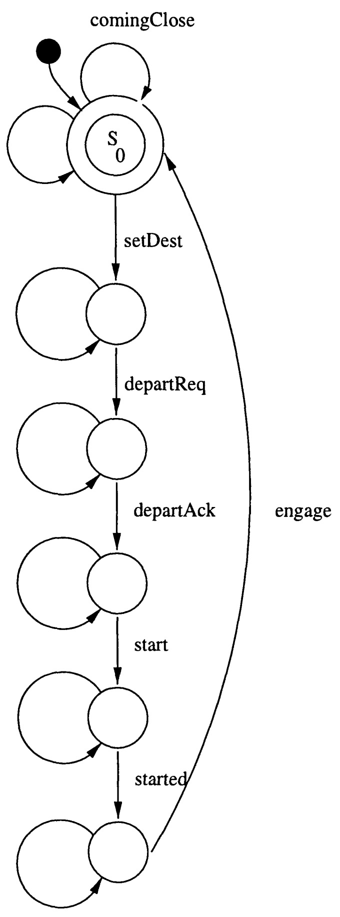


Fig. 13. Automaton for perform departure.

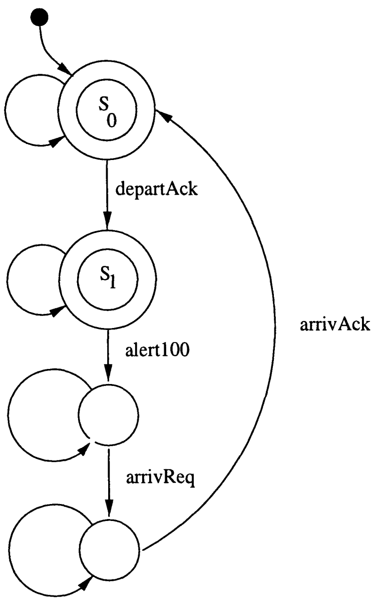


Fig. 14. Automaton for perform approach.

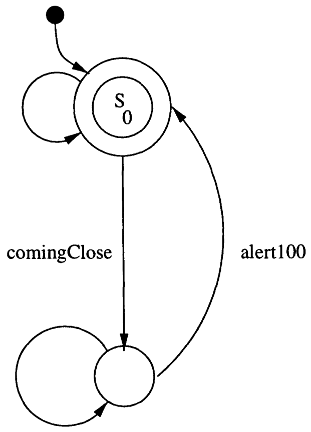


Fig. 15. Automaton for coming close.

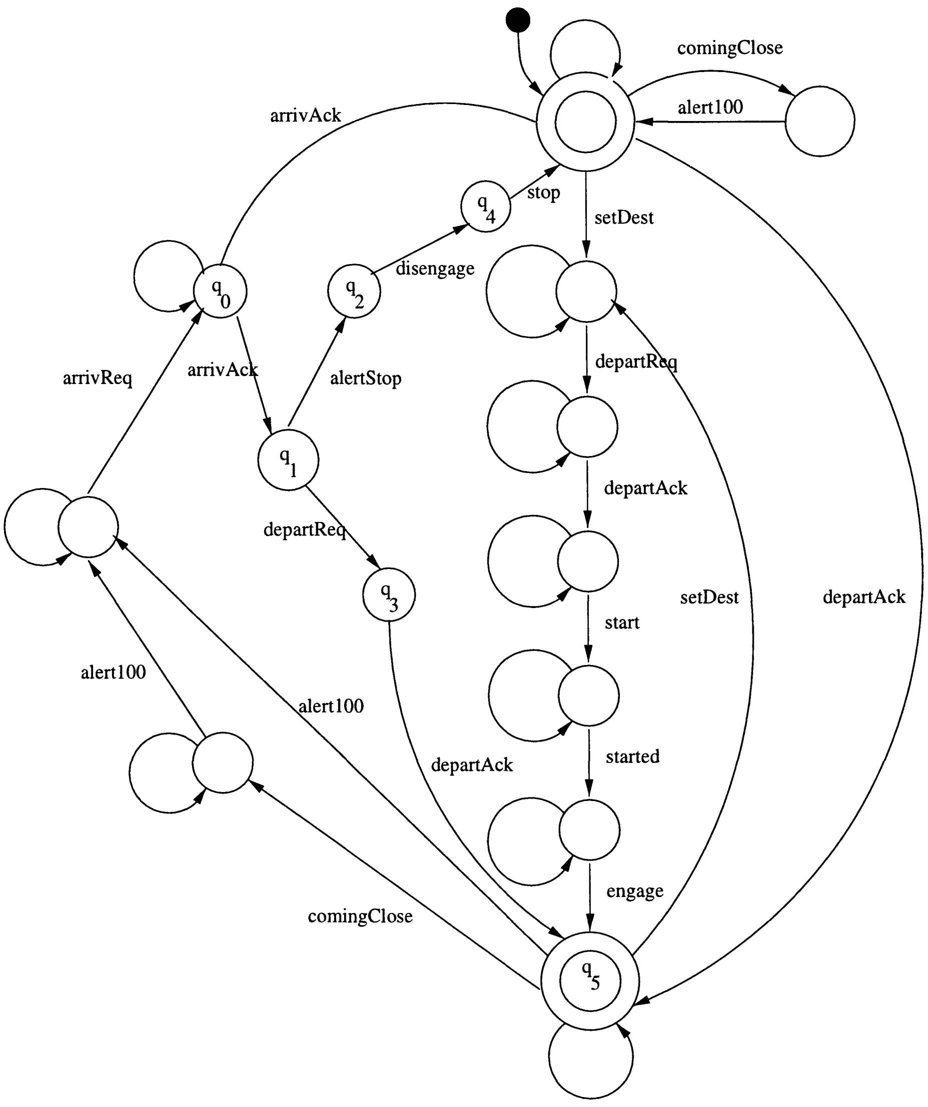


Fig. 17. The global system automaton.

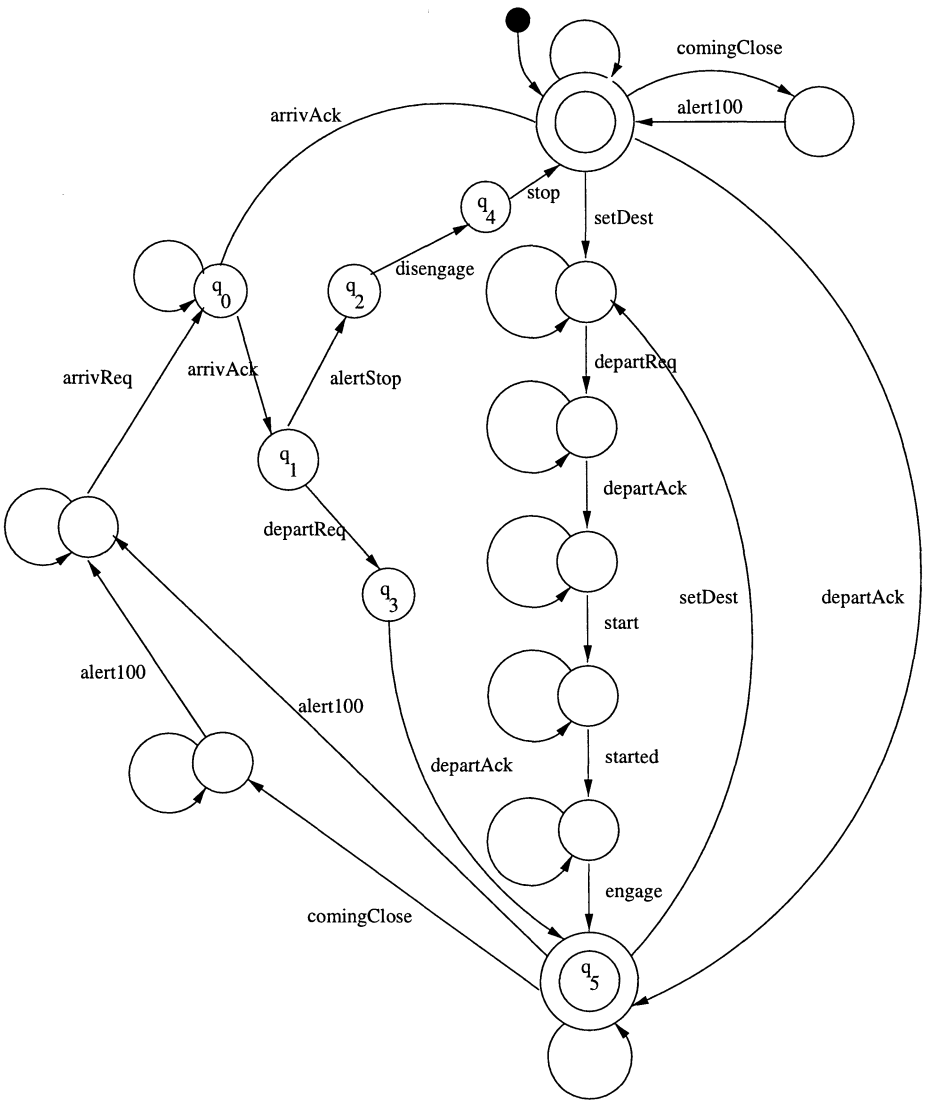


Fig. 17. The global system automaton.

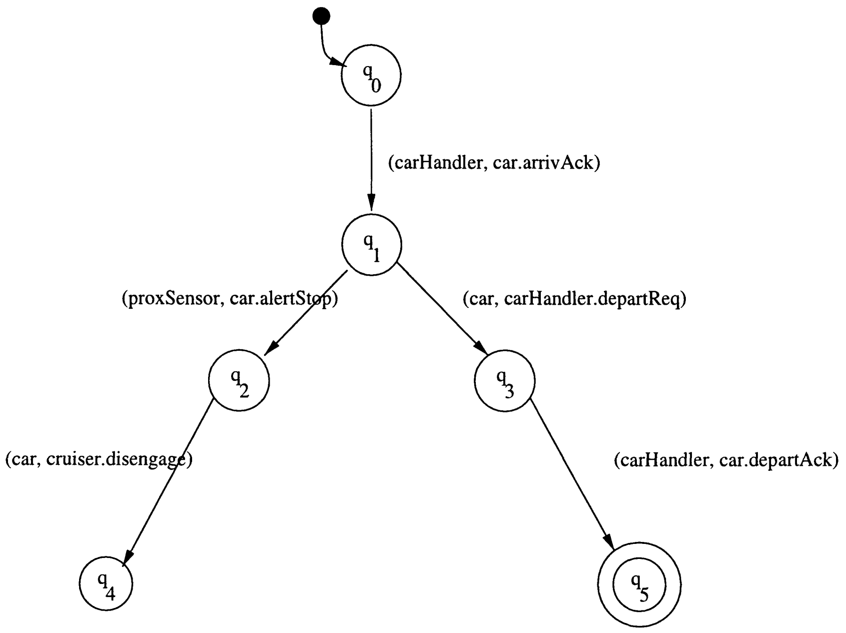

q0
(carHandler, car.arrivAck)
q1
(proxSensor, car.alertStop)
(car, carHandler.departReq)
q2
q3
(car, cruiser.disengage)
(carHandler, car.departAck)
q4
q5

Fig. 18. Subautomaton of the GSA.

and q_5, as appearing in Fig. 17. This subautomaton is shown in Fig. 18. In this figure we provide full information about the message, the sender and receiver, since this information is important for the distribution process.

In general, let $A = \langle Q, q_0, \delta \rangle$ be a GSA describing a system with objects $\mathcal{O} = \{O_1, ..., O_n\}$ and messages $\Sigma = \Sigma_{in} \cup \Sigma_{out}$. Assume that A satisfies the LSC specification LS. Our constructions employ new messages taken from a set Σ_{col}, where $\Sigma_{col} \cap \Sigma = \emptyset$. They will be used by the objects for collaboration in order to satisfy the specification, and are not restricted by the charts in LS.

There are different ways to distribute the global system automaton between the objects. In the next three subsections we discuss three main approaches — **controller object**, **full duplication**, and **partial duplication** — and illustrate them on the GSA subautomaton of Fig. 18. The first approach is trivial and is shown essentially just to complete the proof of the existence of an object system satisfying a consistent specification. The second method is an intermediate stage towards the third approach, which is more realistic.

5.1. Controller object

In this approach we add to the set of objects in the system $\mathcal{O}$ an additional object O_{con} which acts as the controller of the system, sending commands to all the other objects. These will have simple automata to enable them to carry out the commands.

Let $|\Sigma_{col}| = |A_{in}| + |A_{out}|$, and let f be a one-to-one function

$$f : A_{in} \cup A_{out} \rightarrow \Sigma_{col}$$

We define the state machine of the controller object O_{con} to be $\langle Q, q_0, \delta_{con} \rangle$, and the state machines of object $O_i \in \mathcal{O}$ to be $\langle \{q_{O_i}\}, q_{O_i}, \delta_{O_i} \rangle$.

The states and the initial state of O_{con} are identical to those of the GSA. The transition relation δ_{con} and the transition relations δ_{O_i} are defined as follows:

If $(q, a, q') \in \delta$ where $a \in A_{in}$, $a = (env, O_i.\sigma_i)$ then

$$(q, f(a), q') \in \delta_{con} \quad \text{and} \quad (q_{O_i}, \sigma_i/O_{con}.f(a), q_{O_i}) \in \delta_{O_i}.$$

If $(q, /a, q') \in \delta$ where $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$ then

$$(q, /O_i.f(a), q') \in \delta_{con} \quad \text{and} \quad (q_{O_i}, f(a)/O_j.\sigma_j, q_{O_i}) \in \delta_{O^i}.$$

This construction is illustrated in Fig. 19, which shows the object system obtained by the synthesis from the GSA of Fig. 18. It includes the state machine of the controller object O_{con}, and the transitions of the single-state state machines of the objects **carHandler**, **car** and **proxSensor**.

The size of the state machine of the controller object O_{con} is equal to that of the GSA, while all other objects have state machines with one state. Section 5.4 discusses the total complexity of the construction.

5.2. Full duplication

In this construction there is no controller object. Instead, each object will have the state structure of the GSA, and will thus "know" what state the GSA would have been in.

Recalling that $A = \langle Q, q_0, \delta \rangle$ is the GSA, let k be the maximum outdegree of the states in Q. A labeling of the transitions of A is a one-to-one function *tn*:

$$tn : \delta \rightarrow \{1, ..., k\}$$

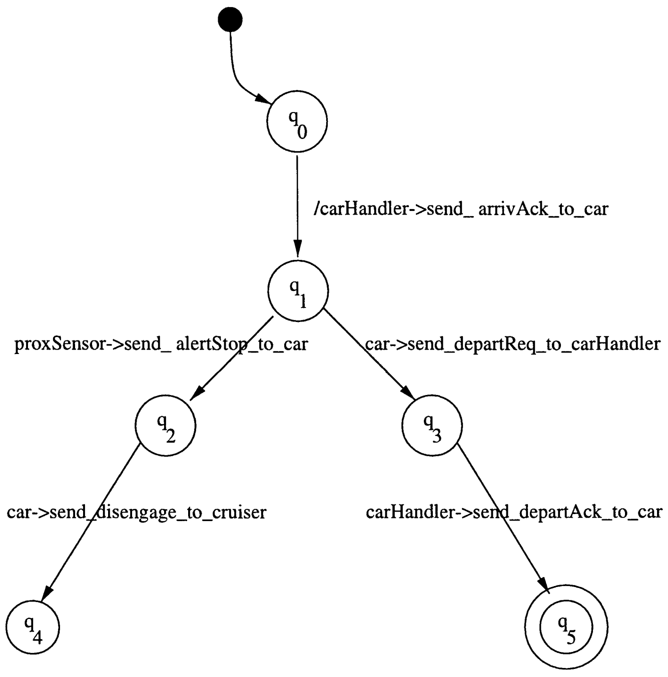
Let $|\Sigma_{col}| = k$ and let f be a one-to-one function

$$f : \{1, ..., k\} \rightarrow \Sigma_{col}$$

The state machine for object O_i in $\mathcal{O}$ is defined to be $\langle Q, q_0, \delta_{O_i} \rangle$. If $(q, a, q') \in \delta$, where $a \in A_{in}$, $a = (env, O_i.\sigma_i)$ and $a' = f(tn(q, a, q')) \in \Sigma_{col}$, then $(q, \sigma_i/O_{i+1}.a') \in \delta_{O_i}$ and for every $j \neq i$, $(q, a'/O_{j+1}.a', q') \in \delta_{O_j}$.

If $(q, /a, q') \in \delta$, where $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$ and $a' = f(tn(q, /a, q')) \in \Sigma_{col}$, then $(q, /O_j.\sigma_j; O_{i+1}.a', q') \in \delta_{O_i}$ and for every $j \neq i$, $(q, a'/O_{j+1}.a', q') \in \delta_{O_j}$.

This construction is illustrated in Fig. 20 on the sub-GSA of Fig. 18. The maximal outdegree of the states of the GSA in this example is 2, and the set of collaboration messages is $\Sigma_{col} = \{1, 2\}$. Again, complexity is discussed in Section 5.4.


q0
/carHandler->send_ arrivAck_to_car
q1
proxSensor->send_ alertStop_to_car
car->send_departReq_to_carHandler
q2
q3
car->send_disengage_to_cruiser
carHandler->send_departAck_to_car
q4
q5

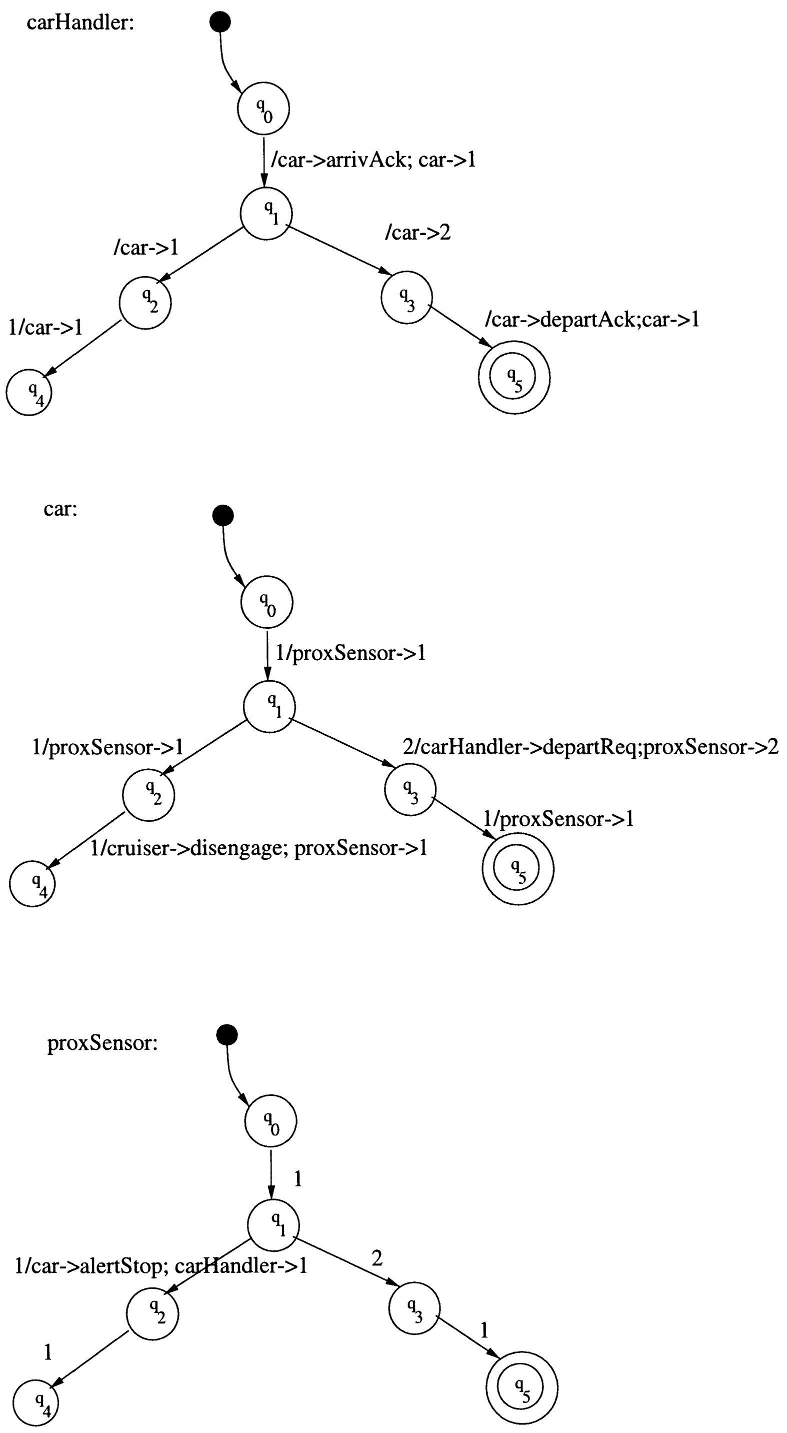


Fig. 20. Full duplication.

5.3. Partial duplication

The idea here is to distribute the GSA as in the full duplication construction, but to merge states that carry information that is not relevant to this object in question. In some cases this can reduce the total size, although the worst case complexity remains the same.

The state machine of object O_i is defined to be $\langle Q_{O_i} \cup q_{idle}, q_0, \delta_{O_i}\rangle$, where $Q_{O_i} \subseteq Q$ is defined by

$$Q_{O_i} = \left\{ q \in Q \;\middle|\; \begin{array}{l} \exists q' \in Q \; \exists a \in A_{out} \text{ s.t.} \\ a = (O_i, O_j.\sigma_j), (q, /a, q') \in \delta \text{ or} \\ \exists q' \in Q \; \exists a \in A_{in} \text{ s.t.} \\ a = (env, O_i.\sigma_i), (q', a, q) \in \delta \end{array} \right\}$$

Thus, in object O_i we keep the states that the GSA enters after receiving a message from the environment, and the states from which O_i sends messages.

Let $|\Sigma_{col}| = |Q|$, and let f be a one-to-one function

$$f : Q \to \Sigma_{col}$$

The transition relation δ_{O_i} for object O_i is defined as follows:
If $(q, a, q') \in \delta$, $a = (env, O_i.\sigma_i)$ then $(q, \sigma_i/O_{i+1}.f(q'), q') \in \delta_{O_i}$.
If $(q, /a, q') \in \delta$, $a = (O_j, O_i.\sigma_i)$, then either $q' \in Q_{O_j}$ and then $(q, /O_i.\sigma_i; O_{j+1}.f(q'), q') \in \delta_{O_j}$, or $q' \notin Q_{O_j}$ and then $(q, /O_i.\sigma_i; O_{j+1}.f(q'), q_{idle}) \in \delta_{O_j}$.
If $q \in Q_{O_i}$, $q' \in Q_{O_i}$ then $(q, f(q'), q') \in \delta_{O_i}$. If $q \in Q_{O_i}$, $q' \notin Q_{O_i}$ then $(q, f(q'), q_{idle}) \in \delta_{O_i}$.
For every $q \in Q_{O_i}$, $(q_{idle}, f(q), q) \in \delta_{O_i}$.

This construction is illustrated in Fig. 21. The states of the GSA of Fig. 18 that were eliminated are q_1, q_2, q_4 and q_5 for **carHandler**, q_0 and q_3 for **car** and q_0, q_2, q_3 and q_4 for **proxSensor**.

5.4. Complexity issues

In the previous sections we showed how to distribute the satisfying GSA between the objects, to create an object system satisfying the LSC specification LS. We now discuss the size of the resulting system, relative to that of LS.

We take the size of an LSC chart m to be $|m| = |dom(m)| = |\#$ of locations in $m|$, and the size of an LSC specification $LS = \langle M, amsg, mod\rangle$ to be $|LS| = \sum_{m \in M} |m|$. The size of the GSA $A = \langle Q, q_0, \delta\rangle$ is simply the number of states $|Q|$. We ignore the size of the transition function δ which is polynomial in the number of states $|Q|$. Similarly, the size of an object is the number of states in its state machine.

Let LS be a consistent specification, where the universal charts in M are charts $\{m_1, m_2, ..., m_t\}$. Let A be the satisfying GSA derived using the algorithm for deciding consistency (Algorithm 1). A was obtained by intersecting the automata $A_1, A_2, ..., A_t$ that accept the runs of charts $m_1, m_2, ..., m_t$, respectively, and then performing additional transformations that do not change the number of states in A. The states of automaton A_i correspond to the cuts through chart m_i, as illustrated, for example, in Fig. 9.

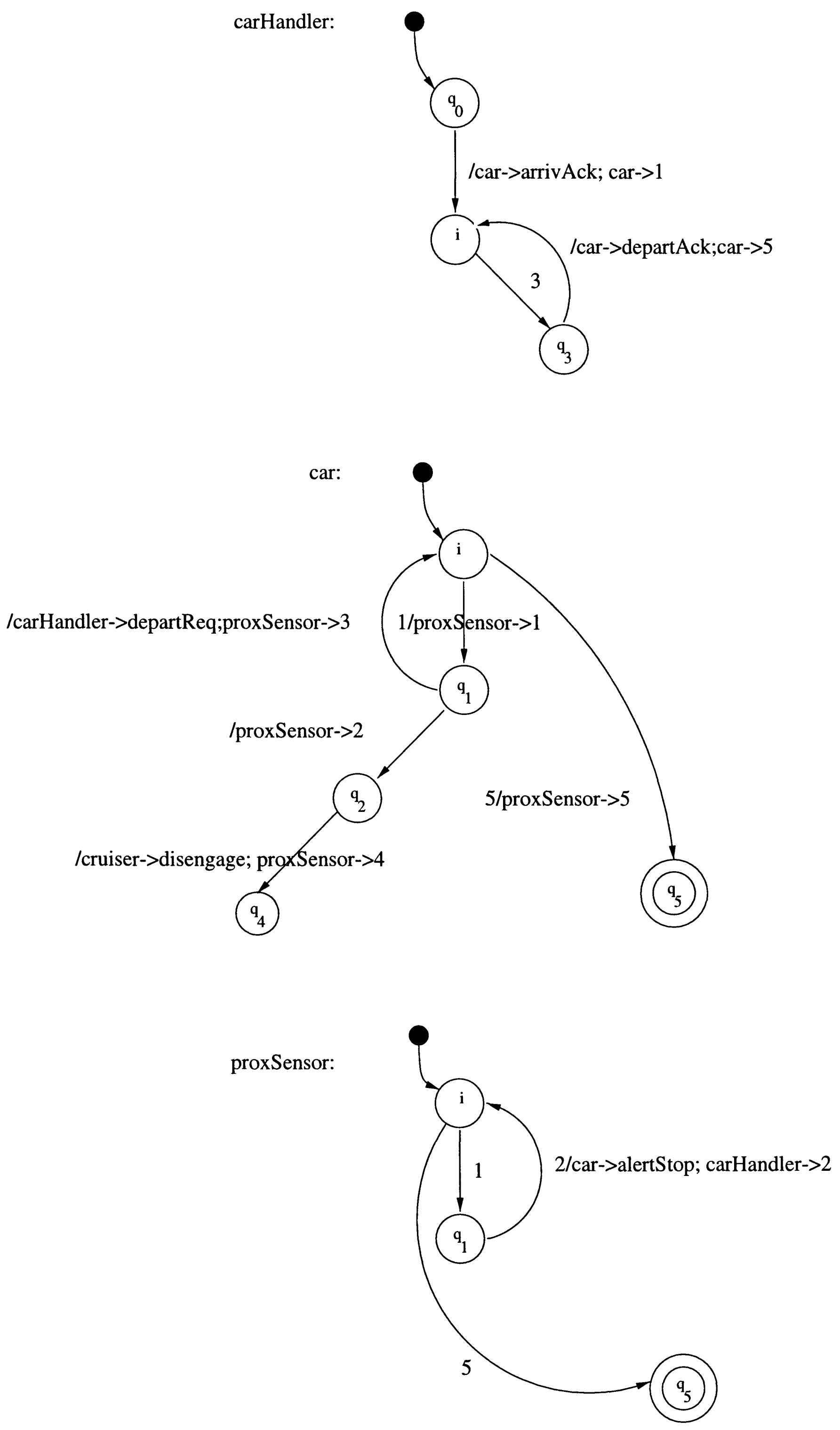


Fig. 21. Partial Duplication.

Proposition 1 *The number of cuts through a chart m with n instances is bounded by $|m|^n$.*

Proof. A cut through a chart is specified by the locations of all the instances appearing in it. Since there is only one location for each instance in a cut and the number of locations of instance i is $|dom(m,i)|$, the number of cuts is at most $\prod_{i \in inst(m)} |dom(m,i)|$. (Actually, the number of cuts will often be smaller because of the order induced by message sending that restricts some of the possibilities.) □

This is consistent with the estimate given in [4] for their analysis of the complexity of model checking for MSCs. We now estimate the size of the GSA.

Proposition 2 *The size of the GSA automaton A constructed in the proof of Theorem 1 satisfies*

$$|A| \leq \prod_{i=1}^{t} |m_i|^{n_i} \leq |LS|^{nt},$$

where n_i is the number of instances appearing in chart m_i, n is the total number of instances appearing in LS, and t is the number of universal charts in LS.

Proof. Since intersection is multiplicative in size, we have

$$|A| = \prod_{i=1}^{t} |A_i| \leq \prod_{i=1}^{t} \prod_{j \in inst(m_i)} |dom(m_i, j)|$$

The result then follows immediately from Proposition 1. (Notice that in construction of an automaton for a universal chart the constructed automaton was deterministic, hence the intersection automaton is deterministic too, and there is no need for a determinization procedure that could have added an exponential factor.) □

The size of the GSA A is thus polynomial in the size of the specification LS, if we are willing to treat the number of objects in the system and the number of charts in the specification as constants. In some cases a more realistic assumption would be to fix one of these two, in which case the synthesized automaton would be exponential in the remaining one. The time complexity of the synthesis algorithm is polynomial in the size of A.

The size of the synthesized object system is determined by the size of the GSA A. In the controller object approach (Section 5.1), the controller object is of size $|A|$ and each of the other objects is of constant size (one state). In the full duplication approach (Section 5.2), each of the objects is of size $|A|$, while in the partial duplication approach (Section 5.3), the size of each of the objects can be smaller than $|A|$, but the total size of the system is at least $|A|$.

5.5. Synthesis without fairness assumptions

We have shown that for a consistent specification we can find a GSA and then construct an object system that satisfies the specification. This construction used null transitions and a fairness assumption related to them, i.e., that a null transition that is enabled an infinite number of times is taken an infinite number of times. We now show that consistent specifications also have satisfying object systems with no null transitions.

Let $A = \langle Q, q_0, \delta \rangle$ be the GSA satisfying the specification LS, derived using the algorithm for deciding consistency. We partition Q into two sets: Q_{stable}, the states in Q that do not have outgoing null transitions, and $Q_{transient}$, the states of which all the outgoing transitions are null transitions. Such a partition is possible, as implied by the proof of Theorem 1. Now, A may have a loop of null transitions consisting of states in $Q_{transient}$. Such a loop represents an infinite number of paths and it will not be possible to maintain all of them in a GSA without null transitions. To overcome this, we create a new GSA $A' = \langle Q', q_0, \delta' \rangle$, with $Q' \subseteq Q$ and $\delta' \subseteq \delta$, as follows.

Let $m \in M$ be an existential chart, $mod(m) = existential$. A satisfies the specification LS, so there exists a word w, with $w \in \mathcal{L}_A \cap \mathcal{L}_m$. Let $q^0, q^1, ...$ be the sequence of states that A goes through when generating w, and let $\delta^0, \delta^1 ...$ be the transitions taken. Since $w \in \mathcal{L}_m$, let $i_1, ..., i_k$, such that $w_{i_1} \cdot w_{i_2} \cdots w_{i_k} \in \mathcal{L}_m^{trc}$. Let j be the minimal index such that $j > i_k$ and $q^j \in Q_{stable}$. The new GSA A' will retain all the states that appear in the sequence $q^0, ..., q^j$ and all the transitions that were used in $\delta^0, ..., \delta^j$. This is done for every existential chart $m \in M$.

In addition, for every $q_i, q_j \in Q$ and for every $a \in A_{in}$, if there exists a sequence of states $q_i, q^1, ..., q^l, q_j$ such that $(q_i, a, q^1) \in \delta$ and for every $1 \leq k < l$ there is a null transition $\delta^k \in \delta$ between q^k and q^{k+1}, then for one such sequence we keep in A' the states $q^1, ..., q^k$ and the transitions $\delta^1, ..., \delta^k$.

All other states and transition of A are eliminated in going from A to A'.

Proposition 3 *The GSA A' satisfies the specification LS.*

Proof. We know that A satisfies LS. Hence, $\forall m \in M, mod(m) = universal \Rightarrow \forall \eta\ \mathcal{L}_A^\eta \subseteq \mathcal{L}_m$. When moving from A to A' we only removed states and transitions, so $\mathcal{L}_{A'}^\eta \subseteq \mathcal{L}_A^\eta$, and therefore $\forall m \in M, mod(m) = universal \Rightarrow \forall \eta\ \mathcal{L}_{A'}^\eta \subseteq \mathcal{L}_m$. Notice that the construction preserves representative paths, so that from every stable state the new automaton can react to every message from the environment.

We show that $\forall m \in M, mod(m) = existential \Rightarrow \exists \eta\ \mathcal{L}_{A'}^\eta \cap \mathcal{L}_m \neq \emptyset$. Let m be an existential chart, and $w \in \mathcal{L}_A \cap \mathcal{L}_m$, as in the construction. Let $\eta = \eta^0, \eta^1, ...$ be the sequence of directed requests sent to the GSA, that generated w. Let $\eta' = \eta^0, \eta_1, ..., \eta_j$ be a prefix of η which A responded to by tracing the state path $q^0, q^1, ..., q^j$ and using transitions $\delta^1, ..., \delta^j$. Denote by w' the word generated by this trace. Then $w' \in \mathcal{L}_{A'}^{\eta'}$ and $w' \in \mathcal{L}_m$. □

6. Synthesizing statecharts

We now outline an approach for a synthesis algorithm that uses the main succinctness feature of statecharts [11] (see also [13]), namely, concurrency, via orthogonal states.

Consider a consistent specification $LS = \langle M, amsg, mod \rangle$, where the universal charts in M are $M_{universal} = \{m_1, m_2, ..., m_t\}$. In the synthesized object system each object O_i will have a top-level AND state with t orthogonal component OR states, $s_1, s_2, ..., s_t$. Each s_j has substates corresponding to the locations of object O_i in chart m_j.

Assume that in a scenario described by one of the charts in $M_{universal}$, object O_i has to send message σ to object O_j. If object O_i is in a state describing a location just before this sending, O_i will check whether O_j is in a state corresponding to the right location, and is ready to receive. (This can be done using one of the mechanisms of statecharts for sensing another object's state.) The message σ can then be sent and the transition taken. All the other component states of O_i and O_j will synchronously take the corresponding transitions if necessary.

This was a description of the local check that an object has to perform before sending a message and advancing to the next location for one chart. Actually, the story is more complicated, since when advancing in one chart we must check that this does not contradict anything in any of the other charts. Even more significantly, we also must check that taking the transition will not get the system into a situation in which it will not be able to satisfy one of the universal charts.

To deal with these issues the synthesis algorithm will have to figure out which state configurations should be avoided. Specifically, let c_i be a cut through chart m_i. We say that $C = (c_1, c_2, ..., c_t)$ is a **supercut** if for every i, c_i is a cut through m_i. We say that supercut $C' = (c'_1, c'_2, ..., c'_t)$ is a **successor** of supercut $C = (c_1, c_2, ..., c_t)$, if there exists i with $succ_{m_i}(c_i, (j, l_j), c'_i)$ and such that for all $k \neq i$ the cut c'_k is consistent with communicating the message $msg(j, l_j)$ while in cut c_k.
Now, for $i = 0, 1, ...$, define the sets

$$Bad_i \subseteq \{\text{all supercuts s.t. at least one of the cuts has at least one hot location}\}$$

as follows:

$$Bad_0 = \left\{C \;\middle|\; C \text{ has no successors } \right\}$$

$$Bad_i = \{C | C \in Bad_{i-1} \text{ or all successors of } C \text{ are in } Bad_{i-1}\}$$

The series Bad_i is monotonically increasing under set inclusion, so that $Bad_i \subseteq Bad_{i+1}$. Since the set of all supercuts is finite the series converges. Denote its limit by Bad_{max}. The point now is that before taking a transition the synthesized object system will have to check that it does not lead to a supercut in Bad_{max}.

The construction is illustrated in Figs. 22, 23, 24 and 25 which show the statecharts for car, carHandler, proxSensor and cruiser, respectively, obtained from the railcar system specification. Notice that an object that does not actively participate in some universal chart, does not need a component in its statechart for this scenario, for example proxSensor does not have a **Perform Departure** component. Notice the use of the *in* predicate in the statechart of the proxSensor for sensing if the car is in the **stop** state.

The number of states in this kind of synthesized statechart-based system is on the order of the total number of locations in the charts of the specification. Now, although in the GSA solution the number of states was exponential in the number of universal charts and in the number of objects in the system, which seems to contrast sharply with the situation here, the comparison is misleading; the guards of the transitions here may involve lengthy conditions on the states of the system.

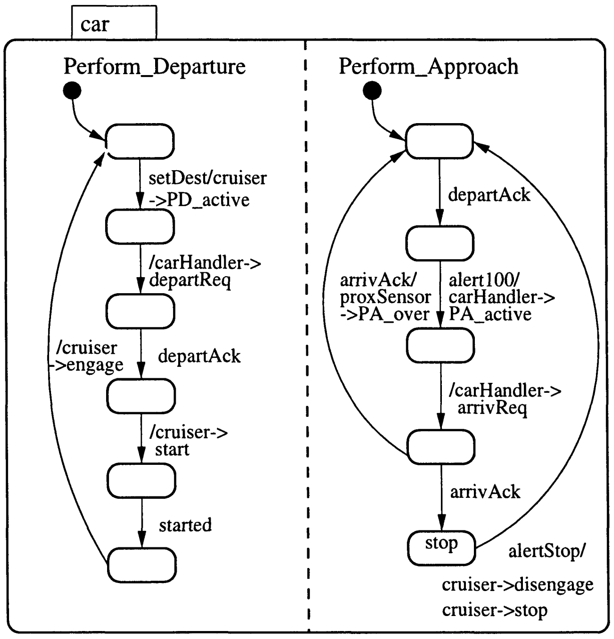


Fig. 22. Statechart of car.

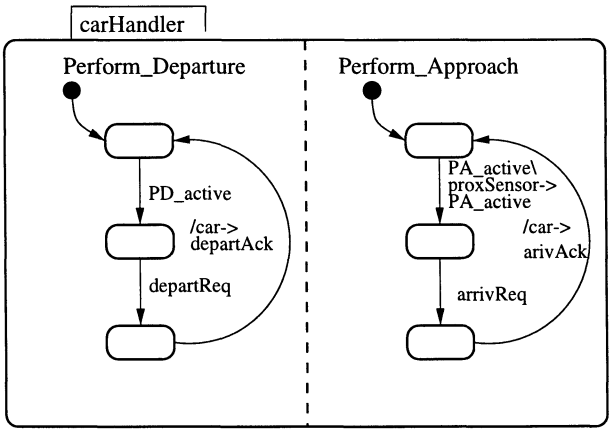


Fig. 23. Statechart of carHandler.

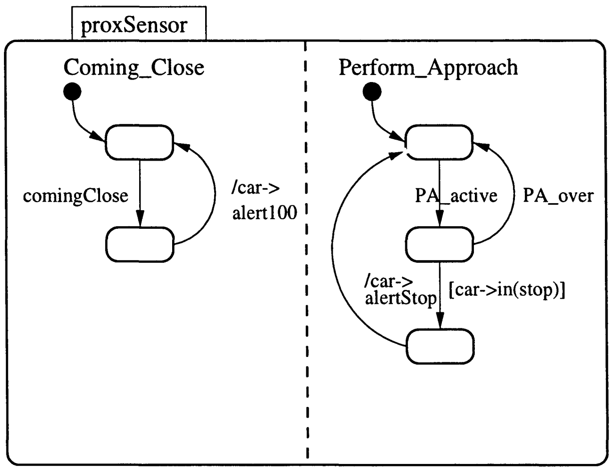


Fig. 24. Statechart of proxSensor.

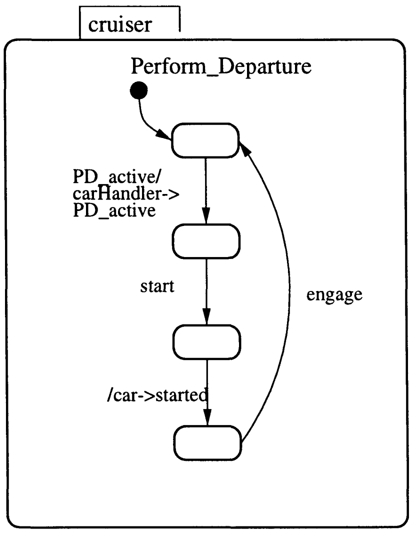


Fig. 25. Statechart of cruiser.

In practice, it may prove useful to use OBDD's for efficient representation and manipulation of conditions over the system state space.

Acknowledgements: We would like to thank Orna Kupferman, Tarja Systa and Erkki Mäkinen for their insightful comments on an early version of the paper.

References

1. Amon, T., G. Boriello and C. Sequin, "Operation/Event Graphs: A design representation for timing behavior", *Proc. '91 Conf. on Computer Hardware Description Language*, pp. 241 – 260, 1991.
2. Alur, R., K. Etessami and M. Yannakakis, "Inference of Message Sequence Charts", *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.
3. Abadi, M., L. Lamport and P. Wolper, "Realizable and unrealizable concurrent program specifications", *Proc. 16th Int. Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, pp. 1–17, 1989.
4. Alur, R., and M. Yannakakis, "Model Checking of Message Sequence Charts", *Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, Lecture Notes in Computer Science, vol. 1664, Springer-Verlag, pp. 114–129, 1999.
5. Boriello, G., "A new interface specification methodology and its application to transducer synthesis", Technical Report UCB/CSD 88/430, University of California Berkeley, 1988.
6. Biermann, A.W., and R. Krishnaswamy, "Constructing Programs from Example Computations", *IEEE Trans. Softw. Eng.*, SE-2 (1976), 141–153.
7. Broy, M., and I. Krüger, "Interaction Interfaces — Towards a Scientific Foundation of a Methodological Usage of Message Sequence Charts", In *Formal Engineering Methods*, (J. Staples, M. G. Hinchey, and Shaoying Liu, eds), IEEE Computer Society, 1998, pp. 2–15.
8. Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design* **19**:1 (2001). (Early version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems*, (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, pp. 293–312, 1999.)
9. Emerson, E.A., "Temporal and modal logic", *Handbook of Theoretical Computer Science*, (1990), 997–1072.
10. Emerson, E.A., and E.M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons", *Sci. Comp. Prog.* **2** (1982), 241–266.
11. Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)
12. Harel, D., "From Play-In Scenarios To Code: An Achievable Dream", *IEEE Computer*, (January 2001), 53 – 60. (Also in *Proc. Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, Vol. 1783 (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22-34.)
13. Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer*, (July 1997), 31–42. (Also in *Proc. 18th Int. Conf. Soft. Eng.*, Berlin,

IEEE Press, March, 1996, pp. 246–257.)

14. Harel, D., and O. Kupferman, "On the Inheritance of State-Based Object Behavior", *Proc. 34th Int. Conf. on Component and Object Technology*, IEEE Computer Society, Santa Barbara, CA, August 2000.

15. Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

16. Krüger, I., R. Grosu, P. Scholz and M. Broy, "From MSCs to Statecharts", *Proc. DIPES'98*, Kluwer, 1999.

17. Kugler, H., D. Harel, A. Pnueli, Y. Lu and Y. Bontemps, "Temporal Logic for Live Sequence Charts", in preparation.

18. Koskimies, K., and E. Makinen, "Automatic Synthesis of State Machines from Trace Diagrams", *Software — Practice and Experience* **24**:7 (1994), 643–658.

19. Koskimies, K., T. Systa, J. Tuomi and T. Mannisto, "Automated Support for Modeling OO Software", *IEEE Software* **15**:1 (1998), 87–94.

20. Kupferman, O., and M.Y. Vardi, "Synthesis with incomplete information", *Proc. 2nd Int. Conf. on Temporal Logic (ICTL97)*, pp. 91–106, 1997.

21. Klose, J., and H. Wittke, "Automata Representation of Live Sequence Charts", *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

22. Leue, S., L. Mehrmann and M. Rezai, "Synthesizing ROOM Models from Message Sequence Chart Specifications", University of Waterloo Tech. Report 98-06, April 1998.

23. Mukund, M., K.N. Kumar, and M. Sohoni, "Synthesizing Distributed Finite-State Systems from MSCs", *Proc. 11th Int. Conf. on Concurrency Theory (CONCUR'00)*, Lecture Notes in Computer Science, vol. 1877, Springer-Verlag, pp. 521–535, 2000.

24. Manna, Z., and R.J. Waldinger, "A deductive approach to program synthesis" *ACM Trans. Prog. Lang. Sys.* **2** (1980), 90–121.

25. Pnueli, A., and R. Rosner, "On the Synthesis of a Reactive Module", *Proc. 16th ACM Symp. on Principles of Programming Languages*, Austin, TX, January 1989.

26. Pnueli, A., R. Rosner, "On the Synthesis of an Asynchronous Reactive Module", *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, pp. 652–671, 1989.

27. Pnueli, A., R. Rosner, "Distributed Reactive Systems are Hard to Synthesize", *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pp. 746–757, 1990.

28. Schlor, R. and W. Damm, "Specification and verification of system-level hardware designs using timing diagrams", *Proc. European Conference on Design Automation*, Paris, France, IEEE Computer Society Press, pp. 518 – 524, 1993.

29. Selic, B., G. Gullekson and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

30. Documentation of the Unified Modeling Language (UML), available from the Object Management Group (OMG), http://www.omg.org.

31. Wong-Toi, H., and D.L. Dill, "Synthesizing processes and schedulers from temporal specifications", In *Computer-Aided Verification '90* (Clark and Kurshan, eds.), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 3, pp. 177–186, 1991.

32. Whittle, J. and J. Schumann, "Generating Statechart Designs from Scenarios",

Proc. 22nd Int. Conf. on Software Engineering (ICSE'00), Limerick, Ireland, June 2000.

33. Z.120 ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.

APPENDIX: Live Sequence Charts (LSCs)

Basic LSC definitions

This section defines the languages specified by a set of LSCs. The LSC definition is a restricted and simplified version of the one in [8]. It does not include conditions, and so does not have to deal with variables. We assume that all messages are synchronous and that there are no failures in the system. An additional restriction is that communication with the environment can appear only as an activation condition of a chart and not as part of the messages in the chart itself.

We assume the LSC specification relates to a system composed of a set of objects $\mathcal{O} = \{O_1 \dots O_n\}$. The instance identifiers in the charts are objects from $\mathcal{O}$. The LSC specifies the behavior of the system in terms of the message communication between the objects in the system.

As explained earlier, messages sent from the environment to the system are distinguished from those sent within the system by the two subsets of Σ, Σ_{in} and Σ_{out}. Let $A_{in} = (env) \times (\mathcal{O}.\Sigma_{in})$, $A_{out} = \mathcal{O} \times (\mathcal{O}.\Sigma_{out})$, and $A = A_{in} \cup A_{out}$.

We want to define the notion of satisfiability of an LSC specification. In other words, we want to capture the languages $\mathcal{L} \subseteq A^* \cup A^\omega$ generated by the object systems that satisfy the LSC specification.

Let $inst(m)$ be the set of all instance-identifiers referred to in chart m. With each instance i we associate a finite number of locations $dom(m,i) \subseteq \{0, ..., l\_max(i)\}$. We collect all locations of m in the set

$$dom(m) = \{< i, l > \mid i \in inst(m) \wedge l \in dom(m,i)\}$$

We associate with each location in the chart m a temperature from the set $Temp = \{hot, cold\}$, by the mapping:

$$temp(m) : dom(m) \rightarrow Temp$$

The messages appearing in m are triples

$$Messages(m) = dom(m) \times \Sigma \times dom(m),$$

where $(< i, l >, \sigma, < i', l' >)$ corresponds to instance i, while at location l, sending σ to instance i' at location l'. Each location can appear in at most one message in the chart. The relationship between locations and messages is given by the mapping

$$msg(m) : dom(m) \rightarrow Messages(m)$$

The *msg* function induces two Boolean predicates *send* and *receive*. We define the binary relation $R(m)$ on $dom(m)$ to be the smallest relation satisfying the following axioms and closed under transitivity and reflexivity:

- order along an instance line:

$$\forall < i,l > \in dom(m), l < l\_max(i) \Rightarrow < i,l > R(m) < i,l+1 >$$

- order induced from message sending:

$$\forall msg \in Messages(m), msg = (< i,l >, \sigma, < i',l' >) \Rightarrow$$
$$< i,l > R(m) < i',l' >$$

- messages are synchronous; they block sender until receipt:

$$\forall msg \in Messages(m), msg = (< i,l >, \sigma, < i',l' >) \Rightarrow$$
$$< i',l' > R(m) < i,l+1 >$$

We say that the chart m is **well-formed** if the relation $R(m)$ is acyclic. We assume all charts to be well-formed, and use $\leq_m$ to denote the partial order $R(m)$.

We denote the **preset** of a location $< i,l >$ containing all elements in the domain of a chart smaller than $< i,l >$ by

$$^\bullet < i,l > = \{< i',l' > \in dom(m) | < i',l' > \leq_m < i,l >\}.$$

We denote the partial order induced by the order along an instance line by $\prec_m$, thus $< i,l > \prec_m < i',l' >$ iff $i = i'$ and $l < l'$.

A **cut** through m is a set c of locations, one for each instance, such that for every location $< i,l >$ in c, the preset $^\bullet < i,l >$ does not contain a location $< i',l' >$ such that $< j,l_j > \prec_m < i',l' >$ for some location $< j,l_j >$ in c. A cut c is specified by the locations in all of the instances in the chart:

$$c = (< i_1,l_1 >, < i_2,l_2 >, ..., < i_n,l_n >)$$

For a chart m with instances $i_1, ..., i_n$ the **initial cut** c_0 has location 0 in all the instances. Thus, $c_0 = (< i_1,0 >, < i_2,0 >, ..., < i_n,0 >)$. We denote $cuts(m)$ the set of all cuts through the chart m.

Dynamic semantics of LSCs

For chart m, some $1 \leq j \leq n$ and cuts c, c', with

$$c = (< i_1,l_1 >, < i_2,l_2 >, ..., < i_n,l_n >)$$
$$c' = (< i_1,l'_1 >, < i_2,l'_2 >, ..., < i_n,l'_n >),$$

we say that c' is a $< j,l_j >$**-successor** of c, and write $succ_m(c, < j,l_j >, c')$, if c and c' are both cuts and

$$l'_j = l_j + 1 \ \wedge \ \forall i \neq j, l'_i = l_i$$

Notice that the successor definition requires that both c and c' are cuts, so that advancing the location of one of the instances in c is allowed only if the obtained set of locations remains unordered.

A **run** of m is a sequence of cuts, $c_0, c_1, ..., c_k$, satisfying the following:

- c_0 is an initial cut.
- for all $0 \leq i < k,$ there is $1 \leq j_i \leq n$, such that $succ_m(c_i, <j_i, l_{j_i}>, c_{i+1})$.
- in the final cut c_k all locations are *cold*.

Assume the natural mapping f between $(dom(m) \cup env) \times \Sigma \times dom(m)$ to the alphabet A, defined by

$$f(<i,l>, \sigma, <j,l'>) = (O_i, O_j.\sigma)$$

$$f(env, \sigma, <j,l>) = (env, O_j.\sigma)$$

Using this notation, $f(Messages(m))$ will be used to denote the letters in A corresponding to messages that are restricted by chart m:

$$f(Messages(m)) = \{f(v) \mid v \in Messages(m)\}$$

Let $c = c_0, c_1, ..., c_k$ be a run. The **execution trace**, or simply the **trace** of c, written $w = trace(c)$, is the word $w = w_1 \cdot w_2 \cdots w_k$ over the alphabet A, defined by:

$$w_i = \begin{cases} f(msg(m)(<j,l_j>)) & \text{if } succ_m(c_{i-1}, <j,l_j>, c_i) \ \wedge \ send(<j,l_j>) \\ \epsilon & \text{otherwise} \end{cases}$$

We define the **trace language** generated by chart m, $\mathcal{L}_m^{trc} \subseteq A^*$, to be

$$\mathcal{L}_m^{trc} = \{w \mid \exists (c_0, c_1, ..., c_k) \in Runs(m) \ s.t. \ w = trace(c_0, c_1, ..., c_k)\}$$

There are two additional notions that we associate with an LSC, its **mode** and its **activation message**. These are defined as follows:

$$mod : m \rightarrow \{existential, universal\}$$

$$amsg : m \rightarrow (dom(m) \cup env) \times \Sigma \times dom(m)$$

The activation message of a chart designates when a scenario described by the chart should start, as we describe below. In the original LSCs [8], there were *activation conditions*, which were assertions over the variables of the system, and *precharts* which were charts that designate a communication sequence that leads to the activation point. Here, we take the activation condition to be a message, in order to simplify the model and not deal explicitly with variables. Our version is not too restricting because a condition being true can trigger the sending of an appropriate message as activation in our model. Notice that in our restricted model we allow activation messages to be sent from the environment while all messages in the chart itself are communications between the objects in the system. Although we will not consider precharts the results in this paper can be extended for such a model. The main modification involves the construction of the automaton for a chart.

The charts and the two additional notions are now put together to form a specification. An **LSC specification** is a triple

$$LS = \langle M, amsg, mod \rangle,$$

where M is a set of charts, and *amsg* and *mod* are the activation messages and modes of the charts, respectively.

The **language** of the chart m, denoted by $\mathcal{L}_m \subseteq A^* \cup A^\omega$, is defined as follows:

- For an existential chart, $mod(m) = existential$, we require that the activation message is relevant (i.e., sent) at least once, and that the trace will then satisfy the chart:

$$\begin{aligned}\mathcal{L}_m \quad = \quad & \{w = w_1 \cdot w_2 \cdots \mid \exists i_0, i_1, ..., i_k \text{ and } \exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}, \text{ s.t.} \\ & (i_0 < i_1 < ... < i_k) \wedge (w_{i_0} = f(amsg(m))) \wedge \\ & (\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \wedge \\ & (\forall j', i_0 \leq j' \leq i_k, j' \notin \{i_0, i_1, ..., i_k\} \Rightarrow w_{j'} \notin f(Messages(m)))\}\end{aligned}$$

  The formula requires that the activation message is sent once ($w_{i_0} = f(amsg(m))$), and then the trace satisfies the chart; i.e., there is a subsequence belonging to the trace language of chart m ($v = v_1 \cdot v_2 \cdots v_k = w_{i_1} \cdot w_{i_2} \cdots w_{i_k} \in \mathcal{L}_m^{trc}$), and all the messages between the activation message until the end of the satisfying subsequence ($\forall j', i_0 \leq j' \leq i_k$) that do not belong to the subsequence ($j' \notin \{i_0, i_1, ..., i_k\}$) are not restricted by the chart m ($w_{j'} \notin f(Messages(m))$).

- For a universal chart, $mod(m) = universal$, we require that each time the activation message is sent the trace will satisfy the chart:

$$\begin{aligned}\mathcal{L}_m \quad = \quad & \{w = w_1 \cdot w_2 \cdots \mid \forall i, w_i = f(amsg(m)) \Rightarrow \exists i_1, i_2, ..., i_k \text{ and} \\ & \exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}, \text{ s.t. } (i < i_1 < i_2 < ... < i_k) \wedge \\ & (\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \wedge \\ & (\forall j', i \leq j' \leq i_k, j' \notin \{i_1, ..., i_k\} \Rightarrow w_{j'} \notin f(Messages(m)))\}\end{aligned}$$

  The formula requires that after each time the activation message is sent ($\forall i, w_i = f(amsg(m))$), the trace will satisfy the chart m (this is expressed in the formula in a similar way to the case for an existential chart.)

Now come the main definitions, which finalize the semantics of our version of LSCs by connecting them with an object system:

Definition

A system S **satisfies** the LSC specification $LS = \langle M, amsg, mod \rangle$, written $S \models LS$, if:

1. $\forall m \in M, \quad mod(m) = universal \Rightarrow \forall \eta \ \mathcal{L}_S^\eta \subseteq \mathcal{L}_m$
2. $\forall m \in M, \quad mod(m) = existential \Rightarrow \exists \eta \ \mathcal{L}_S^\eta \cap \mathcal{L}_m \neq \emptyset$

Here, $\mathcal{L}_S^\eta$, the trace set of object system S on the sequence of directed requests η, is as defined in [14].

Definition An LSC specification LS is **satisfiable** if there is a system that satisfies it.

Embedding LSCs into CTL*

We now show that LSC specifications can be embedded in the branching temporal logic CTL* [9]. Let $LS = \langle M, amsg, mod\rangle$ be an LSC specification. For a chart $m \in M$, we define the formula ψ_m as follows:

- If $mod(m) = universal$, then $\psi_m = AG(amsg(m) \rightarrow \bigvee_{w \in \mathcal{L}_m^{trc}} \phi_w)$.
- If $mod(m) = existential$, then $\psi_m = EF(amsg(m) \wedge (\bigvee_{w \in \mathcal{L}_m^{trc}} \phi_w))$.

Here, if $w = m_1 m_2 m_3 ... m_k$ is an execution of m, we define

$$\phi_w = NU(m_1 \wedge (X(NU(m_2 \wedge (X(NU(m_3 ...)))))),$$

where the formula N is given by $N = \neg m_1 \wedge \neg m_e ... \wedge \neg m_k$.

The following can now be proved:

Proposition Given $LS = \langle M, amsg, mod\rangle$, let ψ be the CTL* formula $\bigwedge_{m \in M} \psi_m$, and let S be an object system. Then

$$S \models \psi \quad \Leftrightarrow \quad S \models LS.$$

It is noteworthy that the reverse is not true: CTL* cannot be embedded in the language of LSCs. In particular, given the single level quantification mechanism of LSCs, the language cannot express formulas of with alternating path quantifiers. However, it shouldn't be too difficult to extend LSCs to allow certain kinds of quantifier alternation, as noted in [8]. This was not done there, since it was judged to have been too complex and unnecessary for real world usage of sequence charts. Since the writing of the current paper we have extended the translation described here, the details will be published separately, see [17]