"Politehnica" University of Bucharest
Faculty of Automatic Control and Computers

Diploma Paper

Synthesis of Distributed Implementation from Sequence Charts

Student:
Cărbunaru Cristina

Advisors:
Ph.D. **Nicolae Ţăpuş,** "Politehnica" University of Bucharest
Ph.D. **Jin Song Dong,** National University of Singapore

Bucharest
2007

Table of Contents

Introduction

Specification languages and notations have much to offer in the achievement of technical quality in system development. Precise notations and languages help to make specifications unambiguous while improving intuitiveness, increasing consistency and making it possible to detect errors during specification rather than implementation. Over the last few decades, many formal modeling languages have been proposed. Different formalisms focus on different systems, different aspects of complex systems, and systems at different stages of development. Some of them have proven successful in reducing development costs and significantly enhancing quality and reliability.

Formal specification languages and notations can be distinguished by their description techniques. The choice of description technique is important because it shapes the system development process.

Distinguished by description techniques, the formalisms can be divided into two categories. One is logic-based formalisms, including those that have a strict mathematical basis and are usually textual. Z Specification is one of these formalisms. The other category is visual formalisms, including diagrammatic modeling languages and notations. One group of them are of particular interest in this paper: scenario-based diagrams, e.g., Message Sequence Charts (MSC) and its variations like Live Sequence Charts (LSC).

Visual formalisms are easy to apply and therefore, widely accepted by the industry. They are used throughout the system development process. In the early analysis stage, scenario-based diagrams are used to specify patterns of interaction between agents as the manifestation of use cases. In the design stage, system design based on state machines specifies system behaviors precisely and may lead directly to implementation. In the testing stage, sequence diagrams are used to capture test cases. Visual formalisms with formal semantics also have tool support for simulation and verification, e.g., *Play-Engine* for LSC. However, as intuition is the primary concern of diagrammatic languages, they can be overwhelming (for instance, with large number of charts) and some are semi-formal (for instance, with *ad hoc* symbols).

Therefore, they are often hard to reason about, and they may impede synthesizing implementations from early analysis stage models. Logic-based and visual formalisms rely on different description techniques and yet their unique strengths naturally complement each other. Recent works on integrating specification languages have evidenced that combinations of logic-based formalisms and visual formalisms can be used to specify a wide range of systems [8].

An ultimate challenge of software engineering is to automatically generate implementations from high-level specifications. One high-level specification of great interest is scenario-based sequence diagrams, which serves as an abstract and natural way of capturing inter-object system requirements in early stages of system development. Sequence diagrams have been a popular means of specifying scenarios of reactive systems for decades. They have found their ways into many methodologies, e.g., Sequence Diagrams in Unified Modeling Languages (UML), Messages Sequence Chart (MSC) in Specification and Description Language (SDL), etc. In this paper, a method to generate prototypes automatically from a variant of sequence diagrams, namely Live Sequence Charts (LSC) is proposed.

Live Sequence Charts (LSC), proposed by Damm and Harel [4], has been rapidly recognized as a rather rich and useful extension of MSC. A large set of constructs has been provided for specifying not only possible behaviors, but also mandatory behaviors. Given a set of scenarios specified using live sequence diagrams, the problem of distributed synthesis is of deciding whether there exists a satisfying distributed object system and if so, synthesize one automatically. LSC have limited expressiveness on modeling the data and functional aspects underlying the scenarios, but compared to the classic message sequence chart, is by far more appropriate to describe system behavior.

A mechanical support for synthesizing Spec# programs from LSCs enriched with data assertions is proposed in this paper. The Spec# programming language is a superset of the C# with additional program-based specification assertions. By targeting a popular programming language like C#, a practical solution to the synthesis problem is provided. For achieving this, LSC specification has to be enriched with a set of data assertions: class invariants, preconditions and post-conditions. The assertions capture the data and functional aspects of the system objects. The problem is of synthesizing a distributed Spec# program which complies not only the LSC model, but also the data assertions.

The concrete work for the project includes developing a visual user interface for describing LSCs and a synthesizer module which produce Spec# programs. Considering the fact that Spec# programs have to be generated, the whole application was implemented in C# programming language. The final objective is to be able to simulate the systems execution, system described using LSCs.

First chapter presents a short introduction about Live Sequence Charts and the way they are represented. Second chapter shows the previous method of synthesizing states machines from LSC specifications. Third chapter describes the enrichments that Spec# brings to C# programming language. The last chapter is dedicated to the new method proposed for generating an implementation from LSCs.

1. Live Sequence Charts

Introduction

Message sequence charts (MSCs) are a well established visual formalism for the description of inter-working of processes or objects. While message sequence charts (MSCs) are widely used in industry to document the interworking of processes or objects, they are expressively weak, being based on the modest semantic notion of a partial ordering of events as defined in the ITU standard. Despite the widespread use of MSCs, several fundamental issues have been left unaddressed. One of the most basic of these is: "*What does an MSC specification mean: does it describe all behaviors of a system, or does it describe a set of sample behaviors of a system?*".

This chapter provides an introduction to the Live Sequence Chart (LSC) language, introduced by Damm and Harel in [4], as a conservative extension to the standard MSC language. In fact, LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart. This makes it possible to specify forbidden scenarios, for example, and enables naturally specified structuring constructs such as subcharts, branching and iteration. The first section motivates the need for another visual formalism and describe the basic features of the LSC language informally. In the second section sketches the formal semantics of LSCs and some examples. The last section presents the language used for describing LSC.

1.1. Shortcomings of Message Sequence Charts

Typically MSCs are used to capture sample scenarios corresponding to use-cases. While the system model becomes refined and conditions characterizing use cases evolve, the intended interpretation often undergoes a metamorphosis from an *existential* to a *universal* view: earlier one wants to say that a condition can become true and that when true the scenario can happen, but later on one wants to say that if the condition characterizing the use case indeed becomes true the system must adhere to the scenario described in the chart.

While the distinction between *mandatory* and *possible* behavior is one of the most urging deficiencies which needs to be addressed in order to construct semantically meaningful computerized tools for describing and analyzing use cases and scenarios, the formal semantics for MSCs leave a good deal of questions beyond unanswered:

Existential or universal view. An MSC shows only one sample run of the system, one scenario, i.e. it is not possible to specify a mandatory protocol between the communicating entities.

Safety and Liveness properties. The MSC semantics offers no distinction, whether progress is enforced or not, i.e. The semantics only define permitted sequences of events; the occurrence of an event can not be enforced. MSCs can only express safety (nothing bad ever happens), but not liveness properties (something good will happen eventually).

Semantics of Conditions. Conditions in MSCs have no formal semantics. In other words: "The semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.". This is obviously not the way to treat conditions from a more formal point of view.

Simultaneous events. MSCs do not allow more than one event to happen exactly at the same time, i.e. there is no notation of simultaneity.

Activation time. An MSC does not state explicitly when the behavior it describes should be observed, i.e. there is no indication of when the MSC should be activated during a system run.

Time treatment. The treatment of time is only rudimentary, since quantitative timing is not covered by the semantics, i.e. timer durations are ignored. Only the correct sequence of timer events, respectively intervals is enforced.

1.2. Adding Life into Basic Message Charts

The Live Sequence Chart (LSC) is a extension to the MSC which addresses these shortcomings presented in section 1 and provides a fully worked out formal semantics. LSCs can therefore accomplish the requirements for the application to more advanced use cases like formal verification.

The question of which parts of behavior are provisional and which are mandatory is not only an issue when an entire chart is considered. It arises in full force already within a single LSC. Should a message arc linking instances *i* and *i'* entail that the communication *will indeed* take place, or just that it *can* take place? Does an instance have to carry out all events indicated along its instance line or can it stop at some point, without continuing? What is the fate of false conditions? Are they mandatory; that is, does the run abort if a false condition is reached? Or are they provisional, meaning that there is some escape route that is taken in such a case?

These are fundamental questions, and one of the main features of the LSC language, which turns it into a true enrichment of MSCs, is the ability to answer them in any of the two ways in each individual case. This is done by adding liveness to the individual parts of the charts, via the ability to specify mandatory, and not only provisional, behavior. Thus, there is allowed for local parts of the chart to be labeled as mandatory or provisional, and this labeling is carried out graphically. Referring to the distinction regarding an internal chart element as the element's *temperature*, mandatory elements are *hot* and provisional elements are *cold*. The graphical notation

are made simple and clear, trying to remain as close as possible to the visual appeal of the ITU standard for MSCs. Here, now, are the extensions themselves.

Along the horizontal dimension of a chart there is not only a distinguish between asynchronous and synchronous message-passing by two kinds of arrow-heads (solid for synchronous and open-ended for asynchronous), but the arrows themselves now come in two variants: a dashed arrow depicts provisional behavior - the communication *may* indeed complete - and a solid one depicts mandatory behavior - the communication *must* complete. Along the vertical dimension dashed line segments are used to depict provisional progress of the instance - the run *may* continue downward along the line - while solid lines indicate mandatory progress - the run *must* continue.

As far as conditions go, in order to help in capturing assertions that characterize use cases, the conditions are turned into first-class citizens. Conditions can thus qualify requirements as assertions over instance variables, and they too come in two flavors, in line with the basic spirit of LSCs: mandatory (hot) conditions, denoted by solid-line condition boxes, and provisional (cold) ones, denoted by dashed-line boxes. If a system run encounters a false *mandatory* condition, an error situation arises and the run aborts abnormally. In contrast, a false *provisional* condition induces a normal exit from the enclosing subchart (or the chart itself, if it is on the top-level).

This two-type interpretation of conditions is quite powerful. Mandatory conditions (that is, hot ones), together with other hot elements, make it possible to specify *forbidden* scenarios, i.e., ones that the system is not allowed to exhibit. This is extremely important and allows the behavioral specifier to say early on which are the “yes-stories” that the system adheres to and which are the “no-stories” that it must not adhere to. Also provisional (cold) conditions provide the ability to specify conventional flow of control, such as conditional behavior and various forms of iteration.

Along the vertical time axis, each instance is associated with a set of *locations*, which carry the temperature annotation for progress within an instance. As explained, provisional progress between locations is represented by dashed lines and mandatory progress by solid lines.

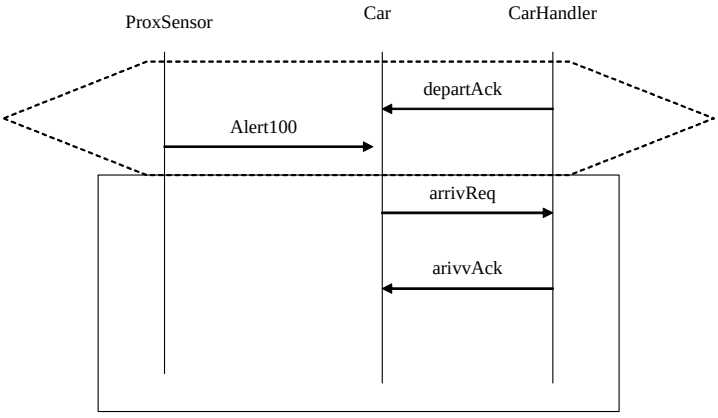


Figure 1.1

Consider figure 1, depicting the *perform approach* scenario. The top part of the figure shows notation for pre-charts: a dashed frame, like that of a cold condition, surrounding the pre-chart, thus indicating that the scenario is relevant only if the pre-chart has been traversed successfully. 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 pre-chart holds. This might happen in a situation where the terminal is closed or when all the platforms are full.

The following table summarizes the dual mandatory/provisional notions supported in LSCs, with their informal meaning:

		Mandatory	Provisional
Chart	Mode Semantics	*Universal* All runs of the system satisfy the chart	*Existential* At least one run of the system satisfies the chart
Location	Temperature Semantics	*Hot* Instance run must move beyond location	*Cold* Instance run need not move beyond location
Message	Temperature Semantics	*Hot* If message is sent it will be received	*Cold* Receipt of message is not guaranteed
Condition	Temperature Semantics	*Hot* Condition must be met; otherwise abort	*Cold* If condition not met exit current (sub)chart

Table 1: Differences between mandatory and provisional notions.

1.2.1. LSC Graphical Representation

This section presents the key elements of the Live Sequence Chart language. The basic idea of LSCs is to allow a distinction between *mandatory* and *possible* behavior, i.e. most LSC elements can be designated to belong to either one category or the other. This distinction is also expressed graphically, which contributes largely to the easy understanding of LSC specifications. Mandatory elements are depicted by solid lines, possible ones by dashed lines.

Instances and Messages. Instances and messages are the elementary building blocks of LSCs. The graphical representation for instances has been adopted from MSCs, i.e. LSC instances consist of an instance head carrying the instance name, an instance axis and an instance end, as the example LSC in figure 1 shows.

As for MSCs, the horizontal dimension is the structural dimension and the vertical dimension corresponds to the time dimension. Deviating from MSCs the environment is described by an instance of its own rather than the border of the LSC. When using LSCs for formal verification, an explicit environment instance offers the possibility of expressing assumptions on the behavior of the environment within the LSC by employing the same elements as for the other instances.

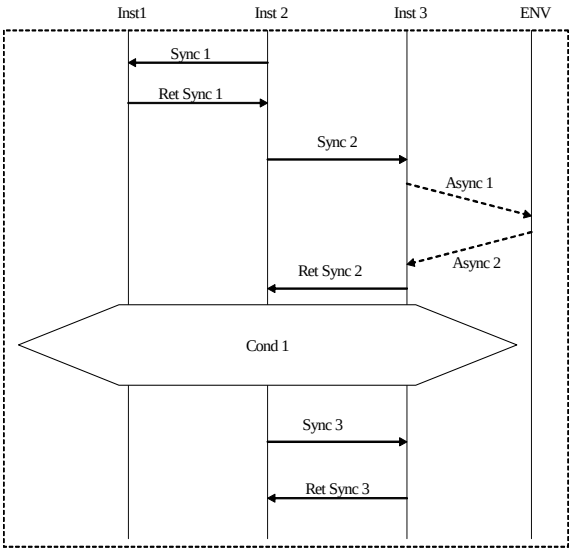


Figure 1.2

There are considered two kinds of messages: asynchronous and instantaneous ones. Messages are visualized by arrows, as shown in figure 1.2. Instantaneous messages have to be drawn horizontally to indicate simultaneity of sending and receiving, while asynchronous ones are drawn slanted to indicate the passage of time between sending and receipt.

Liveness and Temperatures. One deficiency of MSCs is their inability to enforce progress, as mentioned in section 1.1. LSCs overcome this drawback by associating a temperature with both locations and messages. Locations are those points on an instance axis, where some event is attached, e.g. sending or receipt of a message, conditions, etc. The temperature can be either *hot* or *cold*, the former indicating that progress is enforced. The analogy here is that one cannot remain at a hot location for an infinite amount of time, because then one would burn ones feet. This obviously requires that a hot location has to be left, i.e. the following location has to be reached. At a cold location one can stay forever without harming ones feet, i.e. the following location need not be reached. In terms of messages this means that a hot message has to be delivered, whereas a cold message may be lost along the way. Progress information is thus expressed by the temperatures of the messages and along the instance lines. Graphically hot temperatures are represented by solid lines, cold ones by dashed lines.

Conditions and Local Invariants. In order to make statements about the state of the system boolean conditions referring to attributes or data items of the involved entities are used. Graphically, conditions are represented as in MSCs by an elongated hexagon (cf. figure 1). Conditions also come in two variants: *mandatory* and *possible*. A mandatory condition must be satisfied, i.e. the boolean expression associated with it has to hold; violation of the condition is considered an error. Possible conditions do not generate an error when they are not satisfied, but merely constitute an exit from the enclosing LSC. Mandatory conditions are denoted by solid lines (e.g. Cond1 in figure 2) and possible ones by dashed lines. Conditions constrain attributes or data items of entities at one point in time, but often it is desired to express validity of a condition over one instance.

Simultaneous Regions and Coregions. The default ordering of these basic elements is one after the other from top to bottom along the instance axis. Ordering between instances is induced only by messages and conditions ranging over more than one instance. Simultaneous regions allow to group several elements, which should be observed at the same time. This is essential for determining reference points for conditions, local invariants and timers. Graphically they are represented by enlarging the location in question into a small filled circle; see figure 1 for examples.

A coregion is used to indicate that no ordering is imposed on the events it contains, i.e. they may occur in any order. This corresponds to the classical MSC view of a coregion with the exception that - as a consequence of the simultaneous region construct – it is also allowed events in a coregion to take place simultaneously. Coregions are represented graphically by a dotted line running in parallel to the instance axis. This differs from the representation in MSCs, where they are depicted as dashed portions of the instance axis.

Pre-charts. In the preceding paragraphs the graphical elements describing the communication behavior of several interacting entities are presented. This paragraph adds information about when this behavior should be observed and whether it specifies a sample behavior or a protocol to be obeyed, therefore addressing the primary criticism outlined before. The quantification information represents the distinction between mandatory and possible behavior on the chart level. The sample-run or scenario view of MSCs, i.e. the interpretation that there exists a run, which fulfills the LSC, is covered by the possible mode, which is called *existential*. The mandatory mode, which is missing in MSCs as laid out in section 1.1, expresses that the behavior specified in the LSC must be fulfilled by all runs, for which reason it is called the *universal* view. Graphically, the quantification information is depicted by the border style of the LSC: a solid border indicates a universal chart, a dashed border an existential one.

For universal LSCs it is vital to be able to characterize the activation point. If every run has to fulfill the universal LSC, it must be possible to state at which point(s) of the run the LSC should be considered, otherwise the behavior of the entire system has to be specified in one LSC, which is clearly undesirable. Often it is necessary to know more about the history of a run, before being able to decide, whether the LSC should be activated. There may be e.g. more than one way for a run to arrive at a certain system state (characterized by a condition), but the LSC should only be activated, if the run has followed a specific “route”. Activating an LSC is interesting for instance, when no errors have occurred so far. This motivates the introduction of pre-charts which allow to specify a prefix of a run acting as a trigger for the actual LSC. Pre-charts allow to specify a prefix or history, which must be fulfilled by a run in order to activate the LSC. A pre-chart is essentially an LSC, i.e. all language constructs can be used in a pre-chart, but its semantics is different, since the message sequence of the pre-chart is not required to hold in the system, but rather must be observed before activating the actual LSC. The informal semantics of an LSC with pre-chart is consequently: If prechart is completed, then the LSC is activated.

Time. LSCs allow the specification of time constraints either in form of an MSCstyle timer or in interval notation, with a lower and an upper bound. The graphical representation of timers is identical to the one given in MSC, i.e. the setting of a timer is represented by an hour glass symbol, which is annotated by a name and a duration and is connected to the instance axis by a simple line; a timeout symbol is represented by an hour glass symbol, which is connected to the instance axis by an arrow; see figure 1.3 for examples.

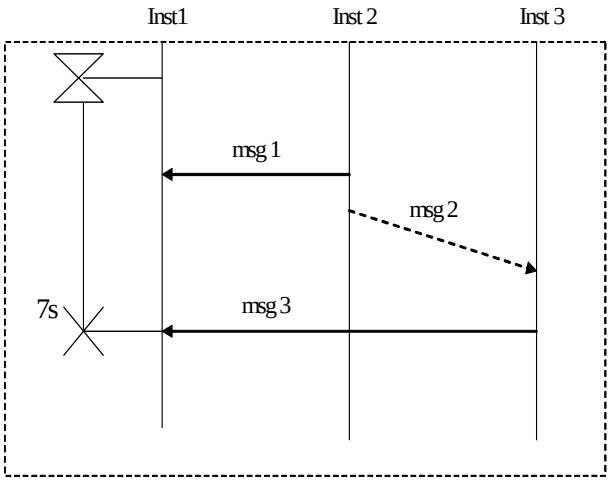


Figure 1.3

Timing intervals express quantitative local liveness properties, since they refer to neighboring atoms (Atoms are the most basic building blocks of an LSC, e.g. instance heads, instance ends, sending a message or receiving a message) They are used to give both a minimum and a maximum delay between two directly consecutive atoms. The delimiting atoms can either be located on the same instance axis, or be the sending and receipt of an asynchronous message. The intervals are placed next to the instance axis between the two locations which delimit them or are attached to the identifier of the constrained asynchronous message (cf. msg2 in figure 3).

1.2.2. Example

Fig. 1.4 shows two typical scenarios of the LCS (Light Control System). When a user enters a room, the motion detector senses the presence of the person and the room controller reacts by sensing the current daylight level and tuning the light with appropriate illumination if the light is already on. Whenever a user leaves a room (leaving it empty), the detector senses no movement, the room controller waits for a safe number of *nomotion* to make sure the room is empty, and then turns off the light. There are a number of important features of LSC presented in the chart, i.e., hot location, hot condition, and forbidden events. It requires that, in order to complete this scenario, no movement should be detected before the chart ends and the light is eventually turned off before it is turned on again.

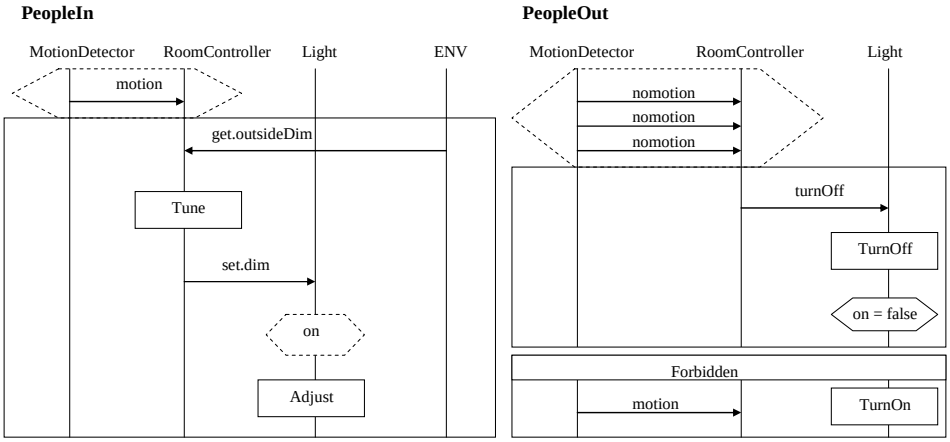


Figure 1.4

Other scenarios of the LCS include the charts in Fig. 1.5, where the occupant may turn the light on/off by pushing the button or the system regularly adjusts the illumination of the light.

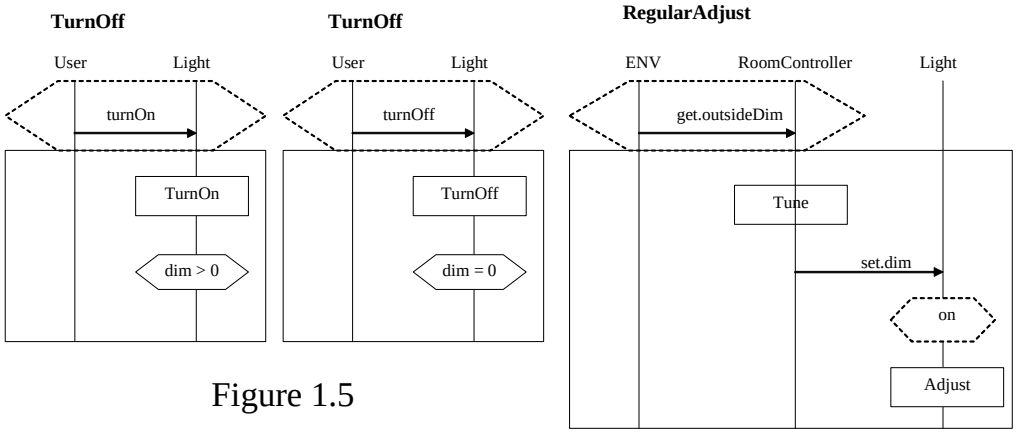


Figure 1.5

LSC also supports advanced features like hierarchy, symbolic instances and messages, etc. The discussion in the rest of the paper assumes that the LSC specification is well-formed, i.e., the weak event ordering relation defined by each chart is acyclic.

1.3. LSC Language

The tool Play-Engine provides the medium to sketch these diagrams (play-in) and see the behavior in operation (play-out), and it's the only tool of this type. Play-Engine, developed by Rami Marelly and David Harel, at the Weizmann Institute of Science, in Rehovot, Israel, enables the analysts to perform system specification by directly interacting with the prototype GUI of the system and making use of the visual constructs of the language, which is referred to as playing in. Play-Engine uses a language for describing LSCs.

The input to my experimental tool is an XML representation of the LSC model. There is not yet a standard interchange format for LSC. The XML format used in PlayEngine is not designed to communicate LSC. No schema or DTD definition is developed. Therefore, the syntax of LSC is defined using XML Schema.

Next, the XML Schema is presented. There are comments on the way LSCs are defined. The XML file that respects this Schema is a valid input for the tool.

A LSC Specification contains charts definitions, class definitions, instance definitions, message types definitions, and type definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="LscSpec">
    <xs:annotation>
      <xs:documentation>An LSC Specification</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TypeDef" type="TYPEDEF" minOccurs="0"
        maxOccurs="unbounded"/>
        <xs:element name="Classes" type="CLASSDEFS"/>
        <xs:element name="MessageTypes" type="MESSAGETYPEDEFS"/>
        <xs:element name="Instances" type="INSTDEFS"/>
        <xs:element name="Chart" type="CHART" minOccurs="0"
        maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="SpecName" type="SPECNAME" use="optional"/>
    </xs:complexType>
  </xs:element>
```

A Message Type definition is represented by the name of the message.

```
<xs:complexType name="MESSAGETYPEDEFS">
  <xs:sequence>
    <xs:element name="MessageType" type="MESSAGETYPEDEF"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="MESSAGETYPEDEF">
  <xs:attribute name="Name" type="MESSAGETYPENAME"/>
</xs:complexType>
```

A message type is described by its name.

```
<xs:complexType name="MESSAGETYPE">
  <xs:sequence>
    <xs:element name="Type" type="MESSAGETYPENAME"/>
  </xs:sequence>
</xs:complexType>
```

A instance definition is represented by the name of the instance and the class to which the instance belongs to.

```
<xs:complexType name="INSTDEFS">
  <xs:sequence>
    <xs:element name="Instance" type="INSTDEF"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="INSTDEF">
```

```
  <xs:sequence>
    <xs:element name="Class" type="CLASS"/>
  </xs:sequence>
  <xs:attribute name="Name" type="INSTNAME" use="required"/>
</xs:complexType>
```

A class definition is composed of invariant definitions, function definitions, variable definitions (members in class). A class has a name.

```
<xs:complexType name="CLASSDEFS">
  <xs:sequence>
    <xs:element name="Class" type="CLASSDEF" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="CLASSDEF">
  <xs:sequence>
    <xs:element name="Invariant" type="INVARIANT" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="Function" type="FUNCTIONDEF" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="Variable" type="VARIDEF" minOccurs="0"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" type="CLASSNAME" use="required"/>
</xs:complexType>
```

A variable use is defined by the variable name and the instance (*instance.name*).

```
<xs:complexType name="VARIABLE">
  <xs:sequence>
    <xs:element name="VariableName" type="VARINAME"/>
    <xs:element name="Instance" type="INSTANCE"/>
  </xs:sequence>
</xs:complexType>
```

A variable definition contains the name, the type, the initial value and the class to which the variable belongs to:

```
<xs:complexType name="VARIDEF">
  <xs:sequence>
    <xs:element name="InitialValue" type="VARIVALUE" minOccurs="0"/>
    <xs:element name="Class" type="CLASS"/>
  </xs:sequence>
  <xs:attribute name="VariName" type="VARINAME" use="required"/>
  <xs:attribute name="Type" type="VARITYPE" use="required"/>
</xs:complexType>
```

A type definition can be a enumeration or a bounded integer:

```
<xs:complexType name="TYPEDEF">
  <xs:choice>
    <xs:element name="BoundedInteger" type="BOUNDEDINT"/>
    <xs:element name="Enumeration" type="ENUMERATION"/>
  </xs:choice>
  <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="BOUNDEDINT">
  <xs:all>
    <xs:element name="LowerBound" type="xs:int"/>
    <xs:element name="UpperBound" type="xs:int"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ENUMERATION">
  <xs:sequence>
```

```
    <xs:element name="Element" type="xs:string"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

A chart has a prechart and a list of events. Also, can contain forbidden events. A chart can be universal or existential.

```
<xs:complexType name="CHART">
  <xs:sequence>
    <xs:element name="ForbiddenEvent" type="EVENT" minOccurs="0"
axOccurs="unbounded"/>
    <xs:element name="Prechart" type="PRECHART" minOccurs="0"/>
    <xs:element name="Event" type="EVENT" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Type" type="CHARTTYPE" use="required"/>
  <xs:attribute name="LscName" type="LSCNAME" use="required"/>
  <xs:attribute name="ActivationMode" type="ACTMODE"
  use="optional"/>
</xs:complexType>
```

A prechart is a sequence of events:

```
<xs:complexType name="PRECHART">
  <xs:sequence>
    <xs:element name="Event" type="EVENT" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

A subchart can be a sequence of events, an if-then-else block or a loop.

```
<xs:complexType name="SUBCHART">
  <xs:choice>
    <xs:element name="Event" type="EVENT" maxOccurs="unbounded"/>
    <xs:element name="If-Then-Else" type="IFTHENELSE"
    maxOccurs="unbounded"/>
    <xs:element name="Loop" type="LOOP" maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
```

A if-then-else block contains a condition, an if-subchart and an else-subchart:

```
<xs:complexType name="IFTHENELSE">
  <xs:sequence>
    <xs:element name="Condition" type="CONDITION"/>
    <xs:element name="If-Then" type="SUBCHART"/>
    <xs:element name="Else" type="SUBCHART" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

A loop contains the condition to exist the loop and the block of events inside the loop:

```
<xs:complexType name="LOOP">
  <xs:sequence>
    <xs:element name="Condition" type="CONDITION"/>
    <xs:element name="Subchart" type="SUBCHART"/>
  </xs:sequence>
</xs:complexType>
```

An invariant is described using a condition:

```
<xs:complexType name="INVARIANT">
  <xs:all>
    <xs:element name="Condition" type="CONDITION"/>
  </xs:all>
</xs:complexType>
```

An event can be a simple event definition or a coregion:

```
<xs:complexType name="EVENT">
  <xs:choice>
    <xs:element name="EventDef" type="EVENTDEF"/>
    <xs:element name="Coregion" type="COREGION"/>
  </xs:choice>
  <!-- <xs:attribute name="Temperature" type="TEMPERATURE"
  use="required"/> -->
</xs:complexType>
<xs:complexType name="COREGION">
  <xs:sequence>
    <xs:element name="EventDef" type="EVENTDEF"
    maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

An event definition is represented by a message, a condition, a subchart, a function call or a timer:

```
<xs:complexType name="EVENTDEF">
  <xs:choice>
    <xs:element name="Message" type="MESSAGE"/>
    <xs:element name="Condition" type="CONDITION"/>
    <xs:element name="Subchart" type="SUBCHART"/>
    <xs:element name="SetTimer" type="SETTIMER"/>
    <xs:element name="Action" type="FUNCTIONCALL"/>
  </xs:choice>
</xs:complexType>
```

A timer is defined by the duration of waiting and the instance that waits:

```
<xs:complexType name="SETTIMER">
  <xs:all>
    <xs:element name="Duration" type="DURATION"/>
    <xs:element name="Instance" type="INSTANCE"/>
  </xs:all>
</xs:complexType>
```

A message is defined by the type, content, sender, receiver, and temperature:

```
<xs:complexType name="MESSAGE">
  <xs:all>
    <xs:element name="Type" type="MESSAGETYPE"/>
    <xs:element name="Content" type="CONTENT"/>
    <xs:element name="From" type="INSTANCE"/>
    <!-- sender will call the function from the content -->
    <xs:element name="To" type="INSTANCE"/>
  </xs:all>
  <xs:attribute name="Temperature" type="TEMPERATURE"
use="required"/>
</xs:complexType>
```

A message content is represented by a function call or a text (string):

```
<xs:complexType name="CONTENT">
  <xs:choice>
    <xs:element name="Function" type="FUNCTIONCALL"/>
    <xs:element name="Message" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

A condition is represented by a Boolean function call and the temperature of the condition:

```
<xs:complexType name="CONDITION">
  <xs:sequence>
```

```
    <xs:element name="Instance" type="INSTANCE"
    maxOccurs="unbounded"/>
    <xs:element name="Expression" type="FUNCTIONCALL"/>
  </xs:sequence>
  <xs:attribute name="Temperature" type="TEMPERATURE"
  use="required"/>
</xs:complexType>
```

A function definition is described by the class that contains the function, the formal parameters of the function, the return type, the pre/post-conditions and the name of the function:

```
<xs:complexType name="FORMALPARAM">
  <xs:all>
    <xs:element name="Name" type="VARINAME"/>
    <xs:element name="Type" type="VARITYPE"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="FUNCTIONDEF">
  <xs:sequence>
    <xs:element name="Class" type="CLASS"/>
    <xs:element name="ReturnType" type="VARITYPE" minOccurs="0"/>
    <xs:element name="FormalParameter" type="FORMALPARAM"
    minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Precondition" type="CONDITION" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="Postcondition" type="CONDITION" minOccurs="0"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="FunctionName" type="FUNCNAME" use="required"/>
</xs:complexType>
```

A function call is defined by the name of the called function, the instance for which the function is called and the effective parameters of the call:

```
<xs:complexType name="EFFECTIVEPARAM">
  <xs:choice>
    <xs:element name="Variable" type="VARIABLE"/>
    <xs:element name="Value" type="xs:string"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="FUNCTIONCALL">
  <xs:sequence>
    <xs:element name="FunctionName" type="FUNCNAME"/>
    <xs:element name="Instance" type="INSTANCE"/>
    <xs:element name="EffectiveParameter" type="EFFECTIVEPARAM"
    minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Here are some definitions of simple types – strings or numbers:

```
<xs:simpleType name="INSTNAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="CLASSNAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="FUNCNAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="LSCNAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="VARINAME">
```

```
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="MESSAGETYPENAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="VARIVALUE">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="CLASS">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="INSTANCE">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="TIMER">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DURATION">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SPECNAME">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="TEMPERATURE">
  <xs:restriction base="xs:string">
       <xs:enumeration value="cold"/>
    <xs:enumeration value="hot"/>
  </xs:restriction>
</xs:simpleType>
```

Some restricted strings:

```
<xs:simpleType name="ACTMODE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="initial"/>
    <xs:enumeration value="invariant"/>
    <xs:enumeration value="iterative"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CHARTTYPE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="existential"/>
    <xs:enumeration value="universal"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
<xs:simpleType name="VARITYPE">
  <xs:restriction base="xs:string">
    <xs:enumeration value="void"/>
    <xs:enumeration value="Int32"/>
    <xs:enumeration value="Boolean"/>
    <xs:enumeration value="String"/>
    <xs:enumeration value="Ddouble"/>
    <xs:enumeration value="Information"/>
    <!-- to be added more data types -->
  </xs:restriction>
</xs:simpleType>
```

2. Z Specification integrated in LSC

Introduction

The Z specification language and its extension are adopted as representatives of state-oriented specification languages. The reasons are that Z is widely known and accepted, and well-developed in terms of specification and refinement. Z is a state-based formal specification language based on the established mathematics of set theory and first-order logic. The set theory used includes standard set operators, set comprehension, Cartesian products, and power sets. Z has been used to specify data and functional models of a wide range of systems, including transaction processing systems and communication protocols. It has been standardized by ISO. In Z, mathematical objects and their properties are collected together in schemas: patterns of declaration and constraint. The schema language is used to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for reuse. A schema contains a declaration part and a predicate part. The declaration part declares variables and the predicate part expresses requirements about the values of the variables.

2.1. Z Specification Language

In 1992, the Queen's Award for Technological Achievement was conferred upon IBM United Kingdom Laboratories Limited and Oxford University Computing Laboratory for "the development and use of an advanced programming method that reduces development costs and significantly enhances quality and reliability": namely, the Z specification language. Z is a state-based formal specification language based on the established mathematics of set theory and first-order logic. The set theory used includes standard set operators, set comprehension, Cartesian products, and power sets.

Mathematical objects and their properties are further collected together in schemas: patterns of declaration and constraint. Z has been used to specify data and functional models of a wide range of systems,including transaction processing systems and communication protocols. It has been standardized by ISO.

One of the fundamental parts of Z logic is the logic of propositions and the logic of predicates. In the Z notation, the two kinds (universal or existential) of quantified expression have a similar syntax:

$Qx : \boldsymbol{R} \mid c \cdot p$

where Q is a quantifier ($\forall$ or $\exists$), x is the bound variable, R is the range of x , c is the constraint and p is the predicate. The optional constraint c restricts the set of objects under consideration: only those objects in R that satisfy c are to be considered. The constraint takes on the role of a conjunction or an implication, depending upon the quantifier concerned.

Example 1: (Quantified predicate)

$$\forall\ x : \mathbf{Z} \mid x > 0 \bullet\ \exists\ y : \mathbf{N} \bullet y > x$$

where Z is the set of integers and N is the set of natural numbers. The expression reads as: for all integers x which are greater than 0, there exists a natural number y which is greater than x.

The other fundamental part of Z logic is the set theory: specifications in Z find their meanings as operations upon sets. Another characteristic of Z notation is its way of constructing definitions. In the Z notation, there are several ways of defining an object. The simplest way is to declare it as a given type: for example, the declaration [Predicate] introduces a new basic type called Predicate.

Example 2: (Abbreviation definition)

$$Illumination == 0 \,..\, 100$$

The abbreviation definition introduces a new name ***Illumination*** for the set of natural numbers ranging from 0 to 100.

Example 3: (Axiom definition)

$$pythagorean : N \times N \rightarrow N$$

$$\forall\ x , y, z : N \bullet (x , y)\ pythagorean\ z \Leftrightarrow x * x + y * y = z * z$$

The axiom defines a total relation among three natural numbers. A relation is a set of tuples. The axiom reads as: the tuple $((x , y), z)$ is in set *pythagorean* if and only if $x^2 + y^2 = z^2$

In addition, there are special mechanisms for free types and schemas. Free types are a more elegant, concise alternative for specifying enumerated collections, compound objects, and recursively defined structures.

Example 4: (Free type definition)

The set **N** could be introduced in Z notations by the following free type definition:

$$nat ::= 0 \mid succ\langle\langle nat\rangle\rangle$$

where *succ* is a constructor function. Every element of *nat* is either 0 or the successors of a natural number, and every element of *nat* has a unique successor.

In Z, the schema language is used to structure and compose descriptions: collating pieces of information, encapsulating them and naming them for re-use. A schema contains a declaration part and a predicate part. The declaration part declares variables and the predicate part expresses requirements about the values of the variables.

Example 5: (State schema)

The following schema encapsulates the state information of a light object.

Light

dim : Illumination *on* : **B**
dim > 0 $\Leftrightarrow$ *on = true*

where **B** is the Boolean type. The declaration part declares two variables. The variable *dim* is the illumination of the light object, ranging of value from 0 to 100 (in percent). The variable *on* is a Boolean variable indicating whether the light object has been turned on or not. The predicate part, referred to as a state invariant, places a constraint upon the values of the two variables, i.e., the *dim* is non-zero if and only if the light object is *on*.

A specification in Z typically consists of a number of state and operation schemas. A state schema groups together state variables and defines the relationship that holds between their values, for instance, the Light schema in example 2.1.5. An operation schema defines the relationship between the 'before' and 'after' valuations of one or more state schemas upon an operation. External inputs to an operation schema are written as variables followed by a question mark in the declaration part.

Example 6: (Operation schema)

The following operation schema defines the operation Adjust by stating how the state variables of the Light schema are updated:

Adjust

Δ*Light* *dim? : Illumination*
on = true $\wedge$ *dim' = dim?*

The variable *dim?* is an input from the environment. The state-update is expressed using a predicate involving both primed and un-primed state variables. The primed variables denote the values of the variables after the operation. The remark is if *dim?* is zero, the state variable on will be set to false because of the state invariant.

A large Z specification can be divided into packages. A package contains one state schema, one initial schema which identifies the initial valuation of the state schema and a number of operation schemas which update the state schema. A Z package thus identifies the state space of an object.

Example 7: (Z package)

____*Light*_____

dim : Illumination
on : **B**

dim > 0 ⇔ on = true

____*TurningOn*_____

Δ*Light*

on = false ∧ dim' = 100 ∧ on' = true

____*Adjust*_____

Δ*Light*
dim? : Illumination

on = true ∧ dim' = dim?

____*TurningOff*_____

Δ*Light*

on = true ∧ dim' = 0 ∧ on' = false

____LightInit_____

Light'

Dim' = 0 ∧ on' = false

These schemas constitute a *Light* package. The state invariant states that the light level is larger than *zero* if and only if the *light* is *on*. The schema named *LightInit* identifies the initial state of the object, i.e., the *light* is *off.* Operation schema *TurningOn, TurningOn* and *Adjus*t are defined to *turn on* or *turn off* the light or set the light level to a specific level.

Z is a powerful language for specifying data and functional models. However, it is not intended for description of non-functional properties, such as usability, performance, size and reliability. Neither is it intended for timed or concurrent behaviors. There are other formal methods that are well suited for these purposes. Z may use in combination with these methods to relate state and state-change information to complementary aspects of design.

2.2. An Integration of Z and LSC

The interaction-based modeling language like LSC and state-based modeling language like Z naturally complement each other. LSC lacks the expressiveness to capture complicated data and functional models. For instance, local actions are often ignored or treated as abstract events in the study of the verification and synthesis problem. It is often assumed to be local variables associated with each object. However, there is no way to specify what exactly the data space of the object is and how the local actions update the local variables except using concrete implementations, which is not desirable since sequence diagrams are used in the early stages of system development. On the other hand, in Z specification, the system behavior patterns are often implicitly embedded within various state/operational constraints. Without explicit system behavior representations, it is difficult to visualize or implement those abstract models. Moreover, Z lacks the expressiveness to

capture dynamic interactive behaviors between the system components. Therefore, a simple yet effective integration of LSC and Z is proposed, integrasion which allows specification of systems with not only complicated interactive behaviors, but also complex data structures.

The combined system specification designed consists of two parts. One is a set of LSC universal charts, which specify mandatory interaction scenarios between system components. The other is a Z specification, which specifies the data and functional models associated with the objects in the system. Each object in the LSC model with nontrivial data states is associated with a Z package in the Z part. Each local action in the LSC model is defined in the respective z package as a Z operation schema. Conditions in the LSC model can only mention variables defined in the respective Z state schema. The system designing process will start with building scenarios from system requirements from which the universal charts are identified. During the process, local actions with abstract meanings are identified.

The designer may then specify each local action using one Z operation schema to formally state how each local action updates the data state. For example, all instances in Fig. 1.4 and Fig. 1.5 (chapter 1) with nontrivial data states are associated with Z packages, i.e., the *Light* package for the *Light* object and the *RoomController* package for the *RoomController* object. Local actions like *Adjust*, *TurnOn*, *TurnOff*, *Tune*, are defined as operation schemas in the respective package. Therefore, the Z specification and the LSC model constitute an integrated specification of the LCS.

Graphically, links from an instance in the chart to its Z state schema and links from local actions to Z operation schemas will be provided, e.g., the Z schema is shown in the popup window once the instance is highlighted and so are the operation schemas. The result is a rigid system architecture, which has its advantages: The data and functional model and the interaction-based model remain orthogonal throughout development and, so, can be analyzed or refined separately using existing tools or theorems. Once both parts stabilize, the integrated specifications will contain sufficient information on both data and control aspects of the system, which allows us to automatically synthesize implementable designs.

2.3. Synthesis of a Distributed Object System

This section shows how to synthesize a distributed object system from the universal charts of the combined specification. For the time being, local actions are treated as abstract events. The synthesized object system is refined in the next section to handle data-related requirements.

The key idea of the synthesis is of the use of a set of special synchronization barriers to monitor completion of universal charts locally. The principles are, first, the synthesis should be robust with the notion of data refinement so that the synthesized design remains valid after refinement of the Z operations and, second the global state machine should never be constructed so that state explosion is avoided and, above all, the synthesized design should be consistent with the specification. The construction uses the notion of finite state machine.

2.3.1. Terminology

Definition 1. *A state machine is a 6-tuple*

$$M \equiv (S, S_0, F, \sum, T, I)$$

where S is a set of states, S0 $\subseteq$ S is a set of initial states, F $\subseteq$ S is a set of accepting states, $\sum$ is the alphabet, and T : S $\times$ $\sum$ $\rightarrow$ S is a transition relation and I labels each state with a Boolean formula.

The Boolean formula labeled with a state is also referred to as a state invariant. Graphically, an initial state is indicated by an arrow from nowhere. A double-lined circle represents an accepting state. A run of a state machine, $\langle s_1, e_1, s_2, e_2, ..., s_i, e_i, s_{i+1}, ... \rangle$, is an alternating sequence of states and events subject to the following: $\forall i \in N : i \geq 1 \bullet (s_i, e_i, s_{i+1}) \in T$ and $s_i \in S_0$. An accepting run is a finite run ending with an accepting state or an infinite one where at least one accepting state repeats infinitely. A state is reachable if and only if there is a finite run that reaches it.

For simplicity, all states subsequently mentioned are reachable. A *false* state, i.e., a state labeled with *false*, is always removed.

2.3.2. Synthesizing Local State Machines

First a state machine for each instance in a single chart is constructed. Given a basic chart *m* (a main chart or a subchart of a main chart without hierarchy), let $M_m^i \equiv (S, S_0, F, \sum, T, I)$ be a state machine synthesized from instance *i* in chart *m*. The basic idea is to construct a state for each location. Thus, S is the set of states corresponding to the locations along the instance. S_0 contains exactly the state corresponding to the first location. *F* contains the states corresponding to the cold locations. For locations labeled with cold conditions, an additional state labeled with the negation of the condition is added so that, if the condition is violated, the additional state is reached and a special event is engaged to terminate the (activation of) the chart. For locations labeled with hot conditions, the condition is labeled with the respective state and no additional state is added. This prevents behaviors that might keep the hot condition from happening. Besides, there is a transition (s_1, e, s_2) in *T* if the location corresponding to s_2 is next to the location corresponding to s_1 and the location corresponding to s_1 is labeled with *e*. After reaching the very last location of the chart (the bottom line), the state machine behaves freely so that it puts no further constraint on the system.

In general, a universal chart u is associated with two sets of synchronous barriers, namely, *u:x:y:conVio* and *u:x:y*, where *x* is a counter uniquely identifying an activation of chart u and y is the identifier of a subchart. There could be overlapping activations of the same chart. For instance, a trace $\langle nomotion, nomotion, nomotion \rangle$ will trigger three different overlapping activations of the chart **PeopleOut**. Event *u:x:y* is used to synchronize the entering or exiting of the subchart y in chart u among those participating instances. Event *u:x:y:conVio* is engaged if and only if a cold condition in the subchart *y* is violated in the *x-activation* of *u*. It is the only event that

the system can engage at the state labeled with the negation of the condition. However, every state in state machines for other instances is equipped with transitions labeled with this event.

Before entering the main chart, a universal chart puts no constraint over the system. Thus, the part of state machine synthesized from the prechart will allow all possible behaviors and, at the same time, monitor communication sequences that may match the prechart.

Let $\sum_u^i$ be the set of events associated with instance *i* in chart *u* (composed of prechart *p* and main chart *m*), including forbidden events. $\sum_u \equiv \bigcup \sum_u^i$ is the visible events of chart *u*. Let $M_m^i \equiv (S, S_0, F, \sum, T, I)$ be the state machine synthesized from instance *i* in prechart *p*. There is a transition (s_1, e, s_2) in M_p^i if the location corresponding to s_2 is next to the location corresponding to s_1 and the location corresponding to s_1 is labeled with *e*. In addition, a transition $(s_1, e', s_{\max})$ is constructed for every event *e'* in $\sum_u^i$ but *e,* where s_{max} is the state corresponding to the last location on instance i in the main chart, i.e., the filled one.

Intuitively, the prechart proceeds whenever an expected event is engaged, whereas an unexpected event aborts the activation of the chart. Because hot conditions in precharts have no semantic meaning, all conditions in precharts are treated as cold conditions. Last, the state corresponding to the last location in the prechart is identified with the state corresponding to the first location in the main chart, i.e., if the prechart is completed, the main chart is reached.

Example. Figure 2.1 shows the state machines synthesized from instances in the scenario **PeopleOut**. The alphabet of each state machine includes the forbidden events. The forbidden events are allowed to engage before entering the main chart. Once a communication sequence matches the prechart, the state machine synchronizes entering of the main chart. All states in the prechart are accepting because the state machine will not constrain the system execution before entering the main chart.

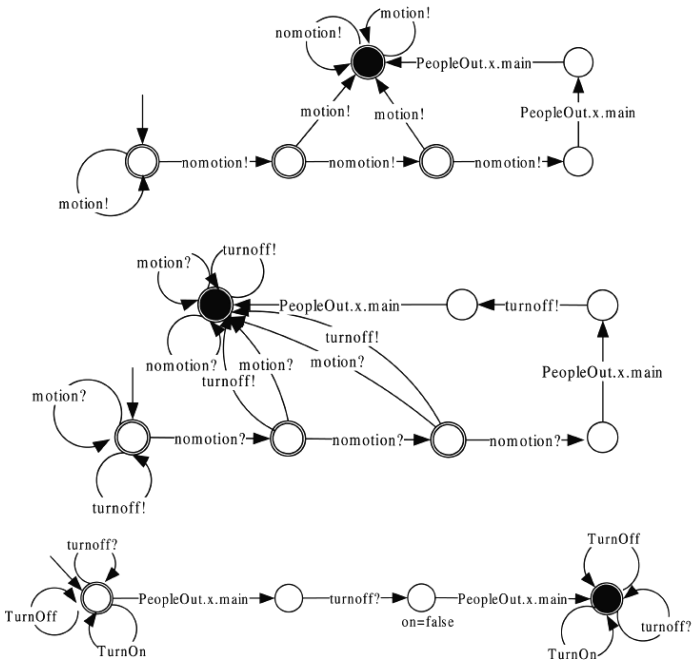


Figure 2.1

Parallel composition of the state machines, e.g., the ones in Figure 2.1, only monitors a single activation of the chart. For instance, the trace $\langle nomotion?, nomotion?, nomotion?, nomotion? \rangle$ is not allowed by the chart in Figure 1.4, yet it is allowed by the state machine for *RoomController* in Fig. 2.1. Though there could be infinite overlapping activations of the same chart, only finite copies of such state machines are required to monitor all the activations. A symmetry reduction shall always make it possible to consider only a finite (and bounded) number of overlapping activations.

Therefore, only a finite copies of the state machines are necessary for monitoring overlapping activations and they can be reused for nonoverlapping activations. The state machine synthesized from instance *i* in chart *u* is written as M_u^i.

In practice, a large number of overlapping activations of the same chart is unlikely because system behaviors are increasingly restricted as the number of overlapping activations increases. For some systems, there is a natural limit on the number of overlapping activations. For instance, there could be at most three overlapping activations of chart **PeopleOut** because the main chart will complete before the fourth *nomotion* event. A simple analysis will tell the maximum number of overlapping activations allowed by a chart.

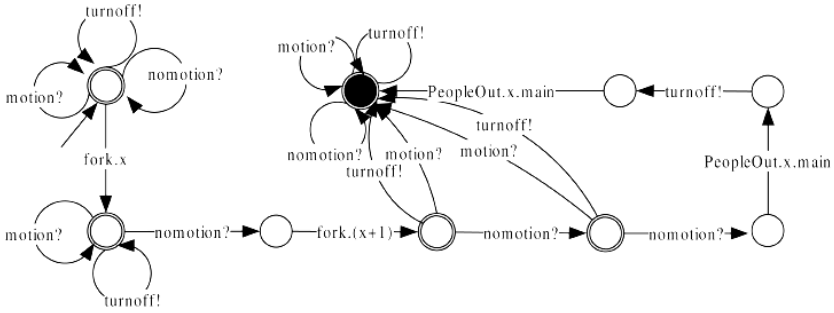
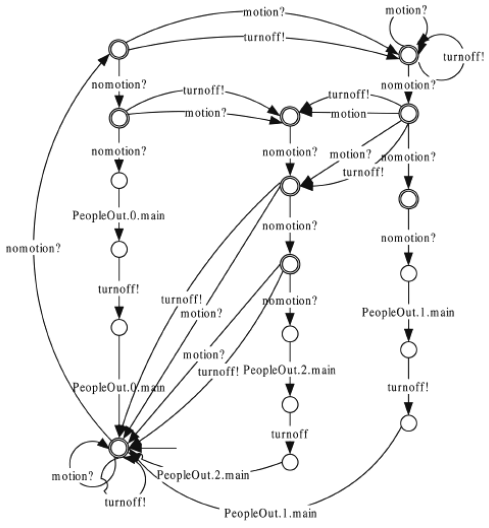


Figure 2.2

Example. Fig. 2.2 shows the state machine synthesized from instance *RoomController* in chart **PeopleOut** to monitor the *x-activation* of the chart. The state machine is augmented with a special synchronization barrier *fork:x*, which is used internally to fork a new copy of the state machine whenever it moves beyond the initial state. Under the assumption that there are at most three overlapping activations of the chart, three copies of the state machine with x ranging from 0 to 2 are constructed.



The copy with $x = 0$ does not have the first state because it is the seed

to fork other copies. The copy with $x = 2$ does not have the state where *fork:3* can be engaged because there is no fourth copy to be forked. The product of the three copies is computed as the final result as shown in Figure 2.3. The very last state (the one composed by three filled state) with the initial state allows nonoverlapping activations. The final state machine can be further reduced using standard technique like bisimulation reduction. For instance, all states labeled with event *fork* are removed as they are irrelevant to system behaviors.

The section presented a systematic way of synthesizing designs from a combination of state-based modeling and interaction-based modeling, namely, Z and LSC. First, an intuitive integration of Z model and LSC model was proposed, integration which is capable of modeling systems with not only complicated data structures but also with complex interactive behaviors. Second, a systematic way of synthesizing distributed finite state designs all the way from the specifications was designed.

3. Extending to Spec#

Introduction

The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This chapter describes the goals and architecture of the Spec# programming system, consisting of the object oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier. The language includes constructs for writing specifications that capture programmer intentions about how methods and data are to be used, the compiler emits run-time checks to enforce these specifications, and the verifier can check the consistency between a program and its specifications.

Software engineering involves the construction of correct and maintainable software. Techniques for reasoning about program correctness have strong roots in the late 1960's (most prominently, Floyd and Hoare). In the subsequent dozen years, several systems were developed to offer mechanical assistance in proving programs correct). To best influence the process by which a software engineer works, one can aim to enhance the engineer's primary thinking and working tool: the programming language. Indeed, a number of programming languages have been designed especially with correctness in mind, via specification and verification, as in, for example, the pioneering languages Gypsy and Euclid. Other languages, perhaps most well-known among them Eiffel, turn embedded specifications into run-time checks, thereby dynamically checking the correctness of each program run.

Despite these visionary underpinnings and numerous victories over technical challenges, current software development practices remain costly and error prone. The most common forms of specification are informal, natural-language documentation, and standardized library interface descriptions. However, numerous programmer assumptions are left unspecified, which complicates program maintenance because the implicit assumptions are easily broken. Furthermore, there's generally no support for making sure that the program works under the assumptions the programmer has in mind and that the programmer has not accidentally overlooked some assumptions. The program development would be improved if more assumptions were recorded and enforced. Realistically, this will not happen unless writing down such specifications is easy and provides not just long-term benefits but also immediate benefits.

The Spec# programming system is a new attempt at a more cost effective way to produce high-quality software. For a programming system to be adopted widely, it must provide a complete infrastructure, including libraries, tools, design support, integrated editing capabilities, and most importantly be easily usable by many programmers.

Therefore, the approach is to integrate into an existing industrial-strength platform, the .NET Framework. The Spec# programming system rests on the Spec# programming language, which is an extension of the existing object-oriented .NET programming language C#. The extensions over C# consist of specification constructs like pre- and postconditions, non-null types, and some facilities for higher-level data abstractions. In addition, C# is enriched with programming constructs whenever doing so supports the Spec# programming methodology. Interoperability with existing .NET code and libraries is allowed, but soundness is guaranteed only as long as the source comes from Spec#. The specifications also become part of program execution, where they are checked dynamically. The Spec# programming system consists not only of a language and compiler, but also an automatic program verifier, called Boogie, which checks specifications statically. The Spec# system is fully integrated into the Microsoft Visual Studio environment.

The main contributions of the Spec# programming system are:

- a small extension to an already popular language,
- a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks,
- tools that enforce the methodology, ranging from easily usable dynamic checking to high-assurance automatic static verification, and
- a smooth adoption path whereby programmers can gradually start taking advantage of the benefits of specification.

In this section, I'll give an overview of the Spec# programming system, its design, and the rationale behind its design. The system is currently under development.

3.1. The Language

The Spec# language is a superset of C#, an object-oriented language targeted for the .NET Platform. C# features single inheritance whose classes can implement multiple interfaces, object references, dynamically dispatched methods, and exceptions, to mention the features most relevant to this paper. Spec# adds to C# type support for distinguishing non-null object references from possibly-null object references, method specifications like pre- and postconditions, a discipline for managing exceptions, and support for constraining the data fields of objects.

3.1.1. Non-Null Types

Many errors in modern programs manifest themselves as null-dereference errors, suggesting the importance of a programming language providing the ability to discriminate between expressions that may evaluate to null and those that are sure not to. In fact, it would be good to eradicate all null dereference errors.

Type support for nullity discrimination is added to Spec#, because types offer the easiest way for programmers to take advantage of nullity distinctions.

Backward compatibility with C# dictates that a C# reference type T denote a possibly-null type in Spec# and that the corresponding non-null type get a new syntax, which in Spec# is T!.

The main complication in a non-null type system arises in addressing non-null fields of partially constructed objects, as illustrated in the following example:

```
class Student : Person
{
     Transcript! t;
     public Student (string name, EnrollmentInfo! ei)
     : base(name)
     {
          t = new Transcript(ei);
     }
}
```

Since the field t is declared of a non-null type, the constructor needs to assign a nonnull value to t . However, note that in this example, the assignment to t occurs after the call to the base-class constructor (as it must in C#). For the duration of that call, t is still null, yet the field is already accessible (for instance, if the base-class constructor makes a dynamically dispatched method call). This violates the type safety guarantees of the non-null type system.

In Spec#, this problem is solved syntactically by allowing constructors to give initializers to fields before the object being constructed becomes reachable by the program.

To correct the example above, one writes:

```
class Student : Person
{
     Transcript! t ;
     public Student (string name, EnrollmentInfo! ei)
     : t (new Transcript(ei)),
     base(name)
     {}
}
```

Spec# borrows this field-initialization syntax from C++, but a crucial point is that Spec#, unlike C++, evaluates field initializers before calling the base-class constructor.

Note that such an initializing expression can use the constructor's parameters, a useful feature that it's vital to any non-null type design. Spec# requires initializers for every non-null field.

Spec# allows non-null types to be used only to specify that fields, local variables, formal parameters, and return types are non-null. Array element types cannot be nonnull types, avoiding both problems with array element initialization and problems with C#'s covariant arrays.

To make the use of non-null types even more palatable for migrating C# programmers, Spec# stipulates the inference of non-nullity for local variables. This inference is performed as a dataflow analysis by the Spec# compiler.

This simple non-null type system was adopted for three reasons. First, problems with null references are endemic in object-oriented programming; providing a solution should be very attractive to a large number of programmers. Second, this simple solution covers a majority of useful non-null programming patterns. Third, for conditions that go beyond the expressiveness of the non-null type system, programmers can use method and class contracts, as described below.

3.1.2. Method Contracts

Every method (including constructors and the properties and indexers of C#) can have a specification that describes its use, outlining a contract between callers and implementations.

As part of that specification, preconditions describe the states in which the method is allowed to be called, and hence are the caller's responsibility. Postconditions describe the states in which the method is allowed to return. The throws set and its associated exceptional postconditions limit which exceptions can be thrown by the method and for each such exception, describe the resulting state. Finally, frame conditions limit the parts of the program state that the method is allowed to modify. The postconditions, throws set, exceptional postconditions, and frame conditions are the implementation's responsibility. Method contracts establish responsibilities, from which one can assign blame in case of a contract-violation error.

Uniform error handling in modern programming languages is often provided by an exception mechanism. Because the exception mechanisms in C# and the .NET Framework are rather unconstrained, Spec# adds support for a more disciplined use of exceptions to improve the understandability and maintenance of programs. As a prelude to explaining method contracts, there is a description of the Spec# view of exceptions.

Exceptions Spec# categorizes exceptions according to the conditions they signal. Looking at exceptions as pertaining to particular methods, Goodenough categorizes exceptions into two kinds of failures, which are in fact client failures and provider failures. A client failure occurs when a method is invoked under an illegal condition, that is, when the method's precondition is not satisfied. Further provider failures are refined into admissible failures and observed program errors. An admissible failure occurs when a method is not able to complete its intended operation, either at all (e.g., the parity of a received network packet is wrong) or after some amount of effort (e.g., after waiting on input from a network socket for some amount of time). The set of admissible failures is part of the contract between callers and implementations. An observed program error is either an intrinsic error in the program (e.g., an array bounds error) or a global failure that's not particularly tied with the method (e.g., an out-of-memory error).

An important consideration among these kinds of exceptions is whether or not one expects a program ever to catch the exception. Admissible failures are part of a program's intended possible behaviors, so it is expected for correct programs to catch and handle admissible failures. In contrast, correct programs never exhibit client failures or observed program errors, and it's not even clear how a program is to react to such errors. If the program handles such failures at all, it would be at the outermost tier of the application or thread.

Because of these considerations, Spec# follows Java by letting programmers declare classes of exceptions as either checked or unchecked. Admissible failures are signaled with checked exceptions, whereas client failures and observed program errors are signaled using unchecked exceptions.

In Spec#, any exception class that implements the interface ICheckedException is considered a checked exception. For more information about the exception design in Spec#, see the paper on exception safety [6].

Preconditions Perhaps the most important programmer assumption is the precondition. Here is a simple example of a method with a precondition:

```
class ArrayList
{
    public virtual void Insert(int index , object value)
    requires 0 <= index && index <= Count;
    requires !IsReadOnly && !IsFixedSize;
    { . . . }
}
```

The precondition specifies that the index into which the object is to be inserted in the array list must be within bounds, and that the list can grow. To enforce these preconditions, the Spec# compiler emits run-time checks that throw a RequiresViolationException, indicating a client failure, if a precondition is not met. If the user invokes Boogie on a call site, then Boogie attempts to verify statically that these preconditions hold at the call site, reporting an error if it cannot.

The .NET Framework documentation for this method is shown in Figure 3.1. There is a subtle difference between the .NET documentation for Insert and the specification of it above. Both specifications state what's expected of the caller; the difference lies in the action taken in the event that preconditions are violated. To support this typical robust programming style of .NET Framework specifications, Spec#'s preconditions can have **otherwise** clauses. These can be used to tell the compiler to use a specified exception, rather than the default RequiresViolationException, in the event that a precondition violation is detected at run time:

ArrayList .*Insert* Method (*Int*32, *Object*)

Inserts an element into the *ArrayList* at the specified index.

```
public virtual void Insert(int index, object value);
```

Parameters

- *index* The zero-based index at which *value* should be inserted.
- *value* The *Object* to insert. The *value* can be a null reference.

Exceptions

Exception Type	Condition
ArgumentOutOfRangeException	*index* is less than zero. *–or–* *index* is greater than *Count* . The *ArrayList* is read-only.
NotSupportedException	*–or–* The *ArrayList* has a fixed size.

Figure 3.1.

```
class ArrayList
{
     void Insert(int index , object value)
     requires 0 <= index && index <= Count
     otherwise ArgumentOutOfRangeException;
     requires !IsReadOnly && !IsFixedSize
     otherwise NotSupportedException;
     { . . . }
}
```

Since it represents a client failure, the exception used in an **otherwise** clause must be an unchecked exception.

Postconditions Method specifications can also include postconditions. For example, one can specify the postconditions of Insert as follows:

```
ensures Count == old(Count) + 1;
ensures value == this[index];
ensures Forall{int i in 0: index; old(this[i]) == this[i]};
ensures Forall{int i in index: old(Count); old(this[i]) == this[i+1]};
```

These postconditions say that the effect of Insert is to increase Count by 1, to insert the given value at the given index, and to keep all other elements in their same relative positions. This example also shows some other Spec# specification features: In the first line, **old**(Count) denotes the value of Count on entry to the method. In the third line, the special function Forall is applied to the comprehension of the boolean expression **old**(**this**[i]) == **this**[i], where i ranges over the integer values in the half-open interval from 0 to less than index . Comprehensions and quantifiers are syntactically restricted in such a way that the compiler can always generate code that computes them.

Boogie attempts to verify each implementation of Insert against these postconditions. When Boogie's verification is successful, then the run-time checks (which would throw an EnsuresViolationException in this case) are not needed since they would never fail.

For run-time checking, Spec# has adopted Eiffel's mechanism for evaluating **old**(E). On entry to a method, the expression E of any **old**(E) occurring in a postcondition is evaluated and the resulting value is saved away. Then, whenever (and if) this value of **old**(E) is needed during the evaluation of the postcondition, the saved value of E is used. Note that the value of **old**(E) may in fact not be needed during the evaluation of the postcondition due to short-circuit boolean expressions or because the method does not terminate normally.

The example above also illustrates a more general point about the differences between checking contracts statically and dynamically. Boogie has knowledge about the program and its built-in data structures. It also has support for quantifiers and can therefore check the postconditions of Insert statically. Contracts that use procedural abstraction, however, can be a problem for static modular checking, since such checking has access only to a limited part of the program. Likewise, contracts that use higher level data structures can be a problem for static checking, because of limitations of decisions procedures and axiomatizations of some theories. Here, dynamic checking is straightforward. On the other hand, the dynamic checking of postconditions can be quite involved when **old** expressions mention quantified variables, as exemplified above.

Though the bulk of specifications is expected to be simple, the more general point is that Spec# supports expressive specifications even when those specifications push the limits of today's checking technology.

Exceptional postconditions As in Java, each method whose invocation may result in a checked exception must account for that exception in the method's throws set. For example, the declaration

```
char Read()
throws SocketClosedException;
{ . . . }
```

where SocketClosedException is a checked exception class, allows the method to throw any checked exception whose allocated type is a subclass of SocketClosedException, but is not allowed to throw any other checked exception. The Spec# compiler holds every implementation to its throws set by a conservative control-flow analysis. A **throws** clause in Spec# can only mention checked exceptions.

Spec# allows a **throws** declaration to be combined with a postcondition that takes effect in the event that the exception is thrown. For example, the exceptional postcondition in

```
void ReadToken(ArrayList a)
throws EndOfFileException
ensures a.Count == old(a.Count);
{ . . . }
```

says that the length of a is unchanged in the event that the method results in an EndOfFileException.

Without further restrictions, it would be possible for a program to foil the compiler's throws-set analysis, which would then undermine Spec#'s guarantee that every checked exception is accounted for. Consider the following example:

```
void ExceptionScam()
{
    Exception e = new MyCheckedException();
    throw e;
}
```

The root of the exception class hierarchy, Exception, is an unchecked exception (because it comes from C#, where all exceptions are unchecked). Since checked exceptions are subtypes of Exception, the **throw** statement in ExceptionScam would have the effect of throwing a checked exception even though the method does not advertise it.

Spec# prevents this: whenever the static type of a thrown expression is an unchecked exception and the static analysis cannot guarantee that the dynamic type is likewise unchecked, then the compiler inserts a run-time check that detects any violation of Spec#'s distinction between checked and unchecked exceptions. For more information about exceptions in Spec#, see the paper on exception safety [6].

Frame conditions Spec# method contracts also include **modifies** clauses (also known as frame conditions), which restrict which pieces of the program state a method implementation is allowed to modify. For example, in the class

```
class C
{
     int x, y;
     void M() modifies x; { . . . }
}
```

method M is permitted to have a net effect on the value of x, whereas the value of y on exit from the method must have the same value as on entry.

Any realistic design of **modifies** clauses includes some facility for abstracting over program state that for reasons of information hiding cannot be mentioned in the method contract. For example, the implementation of ArrayList.Insert is going to modify the private representation of the ArrayList, but private variables are not allowed to be mentioned explicitly in the contract of a public method. Instead, a wildcard can be used.

For example, the specification

```
modifies this^ArrayList;
```

allows any field of **this** declared in class ArrayList to be modified. Spec# also supports other flavors of wildcards, which additionally address the problem of specifying the modification of state in subclasses.

But wildcards are still just a partial solution to the frame problem, because they don't extend to aggregate objects. For example, the ArrayList implementation consists of an array and a count. The **modifies** clause above allows the count and the reference to the array to be changed, but does not give explicit permission to modify the array elements. To deal with aggregate objects, Spec# uses a concept of ownership.

It is said that the ArrayList owns its underlying array, that the array is committed to the ArrayList . Modifications to the state of committed objects do not need to be mentioned explicitly in the **modifies** clause. For more details, see [2], which also describes the connection between ownership and object invariants.

Frame conditions serve as documentation and are used and enforced by Boogie, but they are currently not enforced at run time. There are two reasons for not checking **modifies** clauses at run time. First, they can be prohibitively expensive, since the checking must compare arbitrarily large portions of the heap in a method's pre-state and post-state. Second, the aim is a smooth transition to Spec# from C#; run-time errors in C# programs that otherwise are correct are not wanted.

Inheritance of specifications In Spec#, a method's contract is inherited by the method's overrides. The run-time checks evoked by the method contract are thus also inherited. Not only does this make the specifications more definitive and reliable than today's documentation, but the Spec# specifications also make the code of an implementation easier to read, since today's manually written code for checking preconditions can be rather lengthy.

A method override can add more postconditions by declaring additional **ensures** clauses. The override can add exceptional postconditions only for those exceptions that are already covered by the throws set. The override is not allowed to give any **modifies** clause: enlarging the frame would be unsound, and shrinking the frame can be done with an added postcondition. Spec# does not allow any changes in the precondition, because callers expect the specification at the static resolution of the method to agree with the dynamic checking.

Methods declared in an interface can have specifications, just like the methods declared in a class. Interfaces give rise to a form of multiple inheritance, because a class can inherit a method signature from the superclass and its implemented interfaces. Traditionally, these inherited specifications are combined, which is what Spec# does for postconditions. Spec# also combines exceptional postconditions, but the inherited specifications must have identical throws sets. If a class implements an interface method, then the interface declaration of the method must have a frame condition that is a superset of the class implementation of the method. Spec# does not combine preconditions, unless they are the same, for the reason explained above. Since the obvious definitions of "the same" are either syntactic and brittle, or semantic and require theorem proving, Spec# uses the radical solution of allowing multiple inherited specifications only when these have no **requires** clauses.

Here is an example that shows Spec#'s radical precondition solution not to be too draconian. Consider the following interfaces

```
interface I
{    void M(int x) requires x <= 10; }
interface J
{    void M(int x) requires x >= 10; }
```

Suppose a class C wants to implement both interfaces I and J . In this case, Spec# does not allow C to provide one shared implementation for I .M and J .M . Instead, class C needs to give explicit interface method implementations for M :

```
class C: I, J
{
     void I.M(int x) { . . . }
     void J.M(int x) { . . . }
}
```

(Explicit interface method implementations are a feature of C#.) Because an explicit interface method implementation cannot be accessed other than through the interface, it gets its contract straight from the interface.

3.1.3. Class Contracts

Specifying the rules for using a library or abstraction is done primarily through method contracts, which spell out what's expected of the caller and what the caller can expect in return from the implementation. To specify the design of an implementation, one primarily uses specifications that constrain the value space of the implementation's data.

These specifications are called object invariants and spell out what is expected to hold of each object's data fields in the steady state of the object. For example, the class fragment:

```
class AttendanceRecord
{
     Student[ ]! students;
     bool[ ]! absent ;
     invariant students.Length == absent .Length;
}
```

declares that the lengths of the arrays students and absent are to be the same.

Looking at the simple example above, it is not possible for an object invariant always to hold, because it is not possible in the language to change the lengths of two arrays simultaneously. This is why it is said that the object invariant holds in steady states, which essentially means that the object is not currently being operated on. Following this methodology for object invariants [11], Spec# makes explicit when an object is in its steady state versus when it is exposed, which means the object is vulnerable to modifications. Spec# introduces a block statement **expose** that explicitly indicates when an object's invariant may temporarily be broken: the statement

```
expose (o)
{    S; }
```

exposes the object o for the duration of the sub-statement S , which may then operate on the fields of o . Because field modifications in an object-oriented program tend to be encapsulated in the class that declares the field, the expression o is usually **this**. The object invariant is supposed to hold again at the end of the **expose** statement and Spec# enforces this with a run-time check. Object invariants are also checked at the end of constructors (though there's some flexibility that allows the initial check of an object invariant to be performed elsewhere).

By default, whenever a class or any of its superclasses has a declared invariant, every public method of the class has an implicit

```
expose (this) { . . . }
```

around the method body. The preliminary experience suggests that this default removes most of the need for explicit **expose** statements. In situations where reentrancy is desired, the default can be disabled by a custom attribute on the method.

Exposing an object is not idempotent. That is, it is a checked run-time error if **expose** (o) . . . is reached when o is already exposed. In this way, the expose mechanism is similar to thread-non-reentrant mutexes in concurrent programming, where monitor invariants are the analog of the object invariants. If exposing were idempotent, then one would not be able to rely on the object invariant to hold immediately inside an expose block, in the same way that the idempotence of thread-reentrant mutexes means that one cannot rely on the monitor invariant to hold at the time the mutex is acquired.

For Spec#'s object-invariant methodology to be sound, all modifications of a field o.f must take place while the object o is exposed. Furthermore, the methodology uses an ownership relation to structure objects into a tree-shaped hierarchy. The relation is state dependent, which allows ownership transfer. Such modifications and ownerships are enforced by Boogie, but are not enforced at run time. Object invariants can be declared in any class. To support modular checking of invariants, so that a class does not need to know the invariants of its superclasses and future subclasses, object invariants are partitioned into class frames according to the class that declares each invariant [1]. The **expose** mechanism deals with class frames.

To reduce the programmer's initial cost of adding **expose** statements and to handle non-virtual methods in a more backward compatible way (see [1]), Spec# allows one **expose** statement to expose more than one class frame. To explain this feature, here it is shown the more general form of the **expose** statement in Spec#, which is

```
expose (o upto T)
{ . . . }
```

where T is a superclass of the static type of the expression o . If "**upto** T " is omitted, T defaults to the static type of expression o . More precisely than described it above, the statement exposes all of o 's class frames from above its currently exposed class frame through T (also exposing the class frame T itself). Non-idempotence requires that at least one class frame is exposed as part of the operation. At the end of the **expose** block, the class frames that were exposed on entry are un-exposed, and the object invariant for each of those class frames is checked. This is done at run time using compiler-emitted dynamically dispatched methods that check the invariants.

Exposing an unknown number of class frames, and in particular checking the invariants for class frames whose declarations may not be in scope, poses a problem for modular, static verification. Therefore, a stricter model for **expose** it is used in Boogie.

In particular, whereas the precondition for

expose (o **upto** T) { . . . }

as enforced by run-time checks is that o 's T class frame is un-exposed—that is, that the o's most-derived un-exposed class frame is a subclass of T —Boogie strengthens this precondition by requiring o's most-derived un-exposed class frame to be exactly T . This

way, Boogie is able to find all the object invariants that it needs to check at the end of the **expose** block. In effect, this difference in policy between the run-time behavior and what's enforced by Boogie means that programmers can start writing and running Spec# programs more easily, but then may need to exert additional effort in order to obtain the higher confidence in the program's correctness assured by Boogie (just as additional effort is required to make sure Boogie's modification and ownership rules are satisfied).

Object invariants are allowed to mention only constants, fields, array elements, state independent methods, and confined methods. A method is state independent if it does not depend on mutable state. A confined method may depend on the state of owned objects. The Spec# compiler includes a conservative effect analysis to check that these properties are obeyed.

Spec# also supports class invariants, which are useful to document assumptions about static fields. Methodology and constraints for class invariants are similar to those for object invariants, except that there is no inheritance. The **expose** statement simply takes a class instead of an object as a parameter.

3.1.4. Other Details

Exceptions within contracts If an exception is thrown during the evaluation of a contract in Spec#, then the exception is wrapped in a contract evaluation exception and propagated. This is in contrast to the run-time evaluation of contracts in JML, where such exceptions are caught and the surrounding formula is treated as if it returned boolean value according to certain rules.

Custom attributes on specifications C# provides custom attributes as a way to attach arbitrary data to program structures, such as classes, methods, and fields. A custom attribute is compiled into metadata whose standard format allows various applications to read the custom attributes attached to a particular declaration. Spec# also allows each specification clause to be annotated with custom attributes.

Custom attributes allow users of third-party tools to mark up specifications in tool-specific ways. For instance, the Spec# compiler uses the Conditional custom attribute to control which specifications are emitted as run-time checks in the current build. For example, for the following method

```
int BinarySearch(object[ ]! a, object o, int lo, int hi)
requires 0 <= lo && lo <= hi && hi <= a.Length;
[Conditional ("DEBUG")] requires IsSorted(a, lo, hi);
{ . . . }
```

the compiler emits run-time checks for both preconditions in the debug build, but emits a check only for the first precondition in the non-debug build. This supports the common programming style of debugging assertions.

Purity is the property that a program that runs correctly with all contract checking enabled also runs correctly if some of the contract checking is disabled. Therefore, it is required for all expressions appearing in contracts to be pure, meaning that they have no side effects and do not throw any checked exceptions. The compiler enforces this condition using a conservative effect system.

3.2. System Architecture

Architecturally, the Spec# programming system consists of the compiler, a runtime library, and the Boogie verifier. The compiler has been fully integrated into the Microsoft Visual Studio environment in terms of the project system, build process, design tools, syntax highlighting, and the IntelliSense context-sensitive editing and documentation assistance.

The Spec# compiler differs from an ordinary compiler in that it does not only produce executable code from a program written in the Spec# language, but also preserves all specifications into a language-independent format. Having the specifications available as a separate, compiled unit means program analysis and verification tools can consume the specifications without the need to either modify the Spec# compiler or to write a new source-language compiler.

The Spec# compiler can preserve the specifications in the same binary with the compiled code because it targets the Microsoft .NET Common Language Runtime (CLR). The CLR provides rich metadata facilities for associating many types of information with most elements of the type system (types, methods, fields, *etc.*). The Spec# compiler attaches a specification to each program component for which a specification exists. (Technically, the specifications are preserved as strings in custom attributes. All names are fully resolved; while this renders the format quite verbose, it makes it much easier for any tools consuming it.)

As a result, Boogie consumes compiled code, rather than source code. An additional benefit is that Boogie can be used to verify code written in other languages than Spec#, as long as there is an *out-of-band process* for attaching contracts to such code. Such a process is used to attach specifications to the .NET Framework Base Class Library (BCL).

3.2.1. Run-time Checking

Spec# preconditions and postconditions are turned into inlined code. This is done not only for performance reasons, but also to avoid creating extra methods and fields in the compiled code. All such inlined code is tagged so that code corresponding to the Spec# contracts can be differentiated from the code that comes from the rest of the Spec#

program. Such separation is required by any analysis tool that consumes Spec# contracts from the metadata. For instance, Boogie must be able to determine if the noncontract code in a method meets its postcondition, rather than the combination of the non-contract code followed by the code that checks the postcondition. The inlined code evaluates the conditions and, if violated, throws an appropriate contract exception. To check object invariants, the compiler adds a new method to each class that declares an invariant. Special object fields, such as the invariant level and owner of an object, are added to the super-most class that uses Spec# features within each subtree of the class hierarchy. As mentioned in Section 1, the runtime does not enforce the whole methodology; for instance, run-time checking does not check that an object is exposed before updating a field. This means that an error may go undetected at run time that would be caught by Boogie.

3.2.2. Static Verification

From the intermediate language (including the metadata), Spec#'s static program verifier, Boogie, constructs a program in its own intermediate language, BoogiePL. BoogiePL is a simple language with procedures whose implementations are basic blocks consisting mostly of four kinds of statements: assignments, asserts, assumes, and procedure calls.

An inference system processes the BoogiePL program using interprocedural abstract interpretation to obtain properties such as loop invariants. Any derived properties are added to the program as assert statements or assume statements. The BoogiePL program then goes through several transformations, ending as a verification condition that is fed to an automatic theorem prover. The transformations, such as cutting all loops to derive an acyclic control flow graph by introducing *havoc* statements, are done in a way that preserves the soundness of the analysis. A havoc statement assigns an arbitrary value to a variable; introducing havoc statements for all variables assigned to in a loop causes the theorem prover to consider an arbitrary loop iteration. All feedback from the theorem prover is mapped back onto the source program before it is delivered to the user. The result is that programmers interact with Boogie's prover only by making changes at the program source level, for instance by adding contracts. Currently, Boogie uses the Simplify theorem prover.

3.2.3. Out-of-band Specifications and Other Goodies

All .NET applications use the Base Class Library (BCL) in one form or the other. Thus it is wanted to provide specifications for the entire BCL. This gives any client an immediate benefit even before writing a single contract.

But this raises a problem: how to provide a mechanism for attaching Spec# contracts to code that was written without them? (Note that the BCL can't be modified even if its implementation would be used, since doing so would break versioning.) *Outof-band* specifications, that is, specifications for code external to Spec#, are compiled into a Spec# repository. The repository is consulted in case the Spec# compiler or Boogie encounters a method or class for which it requires a specification (*i.e.*, when the compiler emits run-time checks or when Boogie generates verification conditions), but the method or class in the original code does not have an attached specification. Writing contracts for self-contained examples is easy, but realistic programming is highly dependent on libraries, such as the BCL. A large obstacle then is obtaining contracts for the existing libraries. A companion project is working on semi-automatically generating contracts for existing code. It has automatically extracted almost three thousand preconditions for the current version of the BCL.

3.3. Related Work

A number of programming languages have been designed especially with correctness or verification in mind. These include the pioneering languages Gypsy, Alphard, Euclid, and CLU, which offered different degrees of formality. In Gypsy, which was the first language to include specifications as an integral part of the programming language, the specifications integrated in the source program were aimed directly at program verification via an interactive theorem prover. Alphard was designed around a programming methodology for designing and proving object-like data structures, but the proofs were done by hand. In Euclid, specifications written in the programming language's boolean expressions were checked at run time, with the idea that more complicated specifications, which were supplied in comments, would be used by some external program-verification tool. The CLU programming methodology prominently included specifications, but these were recorded only as stylized comments.

Three modern systems with contracts that have had a direct effect on practical programs are Eiffel, SPARK, and B.

Eiffel is an object-oriented language with almost 20 years of use. The standard library is well documented through contracts, so contracts fall prominently within the purview of programmers. The contracts are enforced dynamically. However, without a full methodology for **modifies** clauses and for object invariants in the presence of callbacks, it would not be possible to obtain modular static verification.

SPARK is a limited subset of Ada, without many dynamic language features like references, memory allocation, and subclassing, yet large enough to be useful for many embedded applications. Praxis Critical Systems has used SPARK in the development of several industrial programs, and their measurements indicate that the rigor provided by SPARK can be cost effective. SPARK offers a selection of static tools, from lightweight sanity checking to full verification with an interactive theorem prover. Compatibility with an existing language has been a high priority in the design of SPARK, just like for Spec#, but their approach is quite different from the proposed one. By ruling out difficult features of Ada, SPARK achieves the property that any SPARK program can be compiled by any standard Ada compiler while retaining its SPARK meaning (all SPARK specifications are placed in stylized Ada comments, and thus they are not used by the compiler). To meet the goal of migrating normally skilled programmers to a higher integrity language, the SPARK's approach of designing a subset of an existing language couldn't be followed. Instead, Spec# is designed to be a superset of an existing language, aiming to support easy and gradual adoption of its new features.

The B approach uses a different methodology for writing programs: starting from full specifications and supporting a machine-aided process for stepwise refining the specifications into compilable programs. The resulting programs are similar in expressiveness to SPARK programs. This methodology, which has been used with success for example in constructing the Paris Metro braking system software, produces only correct programs. However, the skills needed to go through the refinement process make for a steep learning curve for the system and become a barrier for many programmers. It is also not obvious how to extend the methodology to more expressive abstractions, like those in object-oriented programs today.

The Java Modeling Language (JML) is a notation for writing specifications for Java programs. JML specifications, which include rich flavors of method contracts, are recorded in Java source code as stylized comments. An impressive array of tools have been build around JML, including tools for documentation, run-time checking, unit testing, light-weight contract checking, and program verification. Spec# provides a more focused methodology than JML, which for example has yet to adopt a full story for object invariants in the presence of callbacks. The design space of Spec# is somewhat less constrained than JML, since JML does not seek to alter the underlying programming language (which, for example, has let Spec# introduce field initializers and **expose** blocks).

4. Generating Spec# code from a LSC specification – Synthesizer Module

Introduction

Given a set of scenarios specified using live sequence diagrams, the problem of distributed synthesis is of deciding whether there exists a satisfying distributed object system and if so, synthesize one automatically. LSC have limited expressiveness on modeling the data and functional aspects underlying the scenarios, but compared to the classic message sequence chart, is by far more appropriate to describe system behavior.

In the previous chapter, an approach to synthesize distributed state machines from LSCs extended with Z specification was persented. In this chapter, a mechanical support for synthesizing Spec# programs from LSCs enriched with data assertions is proposed. The Spec# programming language is a superset of the C# with additional program-based specification assertions. By targeting a popular programming language like C#, a practical solution to the synthesis problem is provided. For achieving this, the LSC specification is enriched with a set of data assertions: class invariants, preconditions and post-conditions. The assertions capture the data and functional aspects of the system objects. The problem is of synthesizing a distributed Spec# program which complies not only the LSC model, but also the data assertions.

The concrete work for the project includes developing a visual user interface for describing LSCs and a synthesizer module which produce Spec# programs. Considering the fact that Spec# programs had to be generated, implemented the whole application was implemented in C# programming language. The final objective is to be able to simulate the systems execution, system described using LSCs.

4.1. The LSC Specification

A lift system case study for generating executable Java programs from LSCs using CSP as a media language. Reasons are that the lift system is a standard example of reactive systems, and people are familiar with the user requirements of a lift system, and the lift system is not a trivial example because of the complexity caused by inherent concurrent interaction in the system.

A lift system consists of a lift providing transport between *N* floors of the building as dictated by the pressing of a range of service-request buttons. Inside the lift there is a panel of buttons, one for requesting travel to each of the building's

floors. There are two service-request buttons on each floor, for upward and downward travel respectively, though on the first floor and the top floor there is only one button. Pressing an internal button requests a lift to visit the corresponding floor. Pressing an external button requests a lift to visit the floor with the desired direction of travel. For simplicity, only the following four active components in the lift systems are interested, a *Controller* that cooperates the lift system, a *Shaft* through which the controller control the movement of the lift, a *Door* and a set of *Users*.

The following data variables are associated with the *Controller*.

```
final int N;
int [] int_req = new int [N];
int [] ext_req = new int [N];
int dir = 0;
int pos = 0;
int status = 0;
```

The constant *N* represents the number of floors. The integer array *int req* is used to cache the requests from panel of buttons inside, i.e. *int req*[*i*] = 1 if the *i*-th floor is requested from inside. Initially, the array contains all 0s.

Similarly, the array *ext req* caches the external requests, *ext req*[*i*] = 1 (-1) if there is an external request from *i*-th floor for upward (downward) travel. The variable *dir* records the current direction of travel, i.e. 1 for upward and -1 for downward. The variable *pos* records the current position of the lift. Variable *status* is the current status of the lift, either 1 (*Moving*) or 0 (*Holding*). The data variables associated with other objects in the system are skipped since they are all trivial. The following self-containing scenarios of the lift system are interested.

- **InternalRequest, ExternalRequest**: A user requests to go *level* -th floor from inside or requests the service of the lift from outside at *level* -th floor for the direction *dir* . The scenarios are captured by the two charts in Figure 4.1.

```
void UpdateIntReq (int level)
{int_req[level] = 1;}
void UpdateExtReq (int level, int dir)
{ext_req[level] = dir;}
```

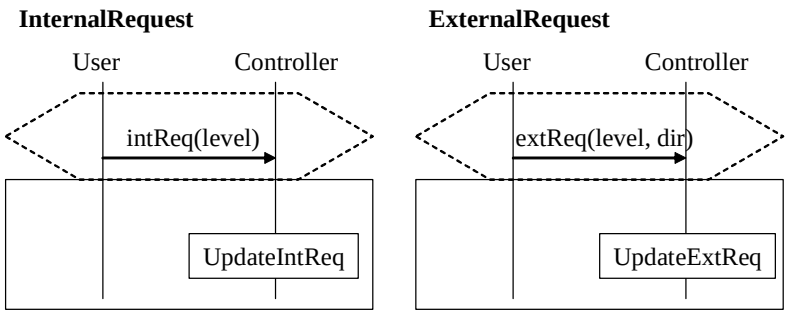


Figure 4.1

- **Arrive**: When the lift arrives a floor, the shaft signal the controller saying the next floor is approaching. The controller checks if the floor is requested from inside or requested from outside with the right direction. If it is, the lift stops and the door opens. Otherwise, the lift keeps moving with the same direction. This scenario is captured in Figure 4.2. Notice that it is required that the *pos* is valid after the local action *UpdatePos* by a hot condition. The conditions and local actions are equivalent to the following:

```
bool ValidatePos(){
return pos >= 0 && pos < N;}

boolean toStop ()
{
return pos == N-1 || pos == 0
||
      int_req[pos] == dir ||
      ext_req[pos] == dir ||
      (ext_req[pos] != 0 &&
      existReqAtSameDirection()
);
}

void SetStatusHolding ()
{status = 0;}

void ClearReq () {
int_req[pos] = 0;
if (ext_req[pos] == dir)
ext_req[pos] = 0;
}
public void ModifyLevel(){
  pos += dir;}
```

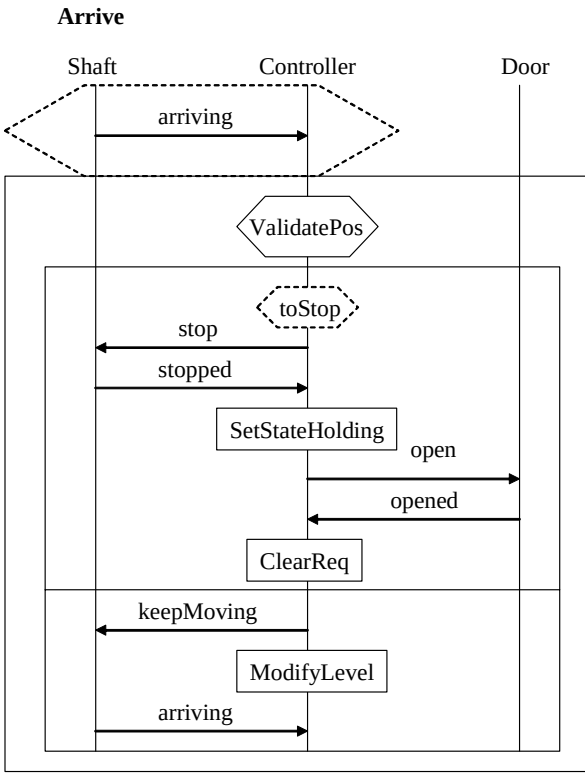


Figure 4.2

- **ServeNextRequest**: After serving a request, the controller checks if there is a request on the current direction. If there is, the lift moves to the requested floor. Otherwise, the controller switches its direction. If the current floor is requested (*existReqAtPos*), the door opens. Otherwise, if there is a request from the opposite direction, the lift changes direction and goes to the desired floor. This scenario is captured by the chart in Figure 4.3. The conditions and local actions are equivalent to the following:

```
boolean existReqAtSameDirection () {
      int sum = 0;
      for (int i = pos+dir; i >= 0 && i < N; i = i + dir)
            sum += int_req[i] + ext_req[i]*ext_req[i];
      return sum > 0;}
```

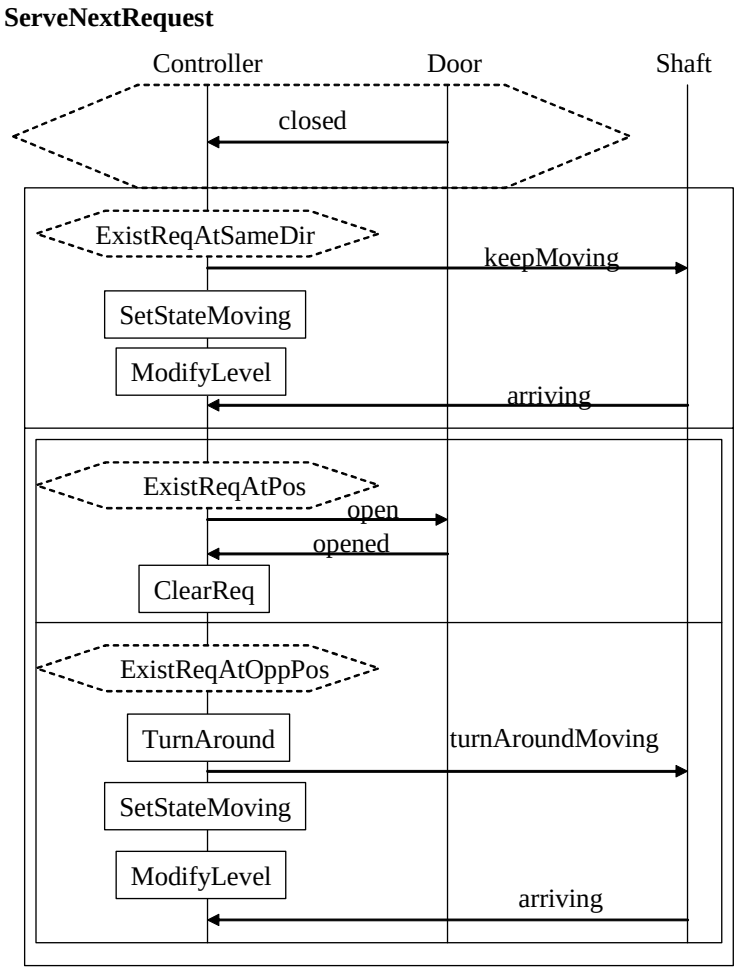


Figure 4.3

```
public bool ExistReqAtOppDir(){
      if (dir == 0) return false;
      int sum = 0;
      for (int i = pos - dir; i >= 0 && i < N; i = i - dir)
            sum += int_req[i] + ext_req[i] * ext_req[i];
      return sum > 0;
}

boolean existReqAtPos () {
      return int_req[pos]==1 || ext_req[pos] != 0;
}

void SetStatusHolding (){
      status = 0;
}
void SetStatusMoving () {
      status = 1;
}
```

```
void ClearReq () {
      int_req[pos] = 0;
      if (ext_req[pos] == dir)
      ext_req[pos] = 0;
}
```

- **OpenCloseDoor**: Once the door is opened, the controller waits for 5 seconds before closing it. This scenario is illustrated in Figure 4.4.

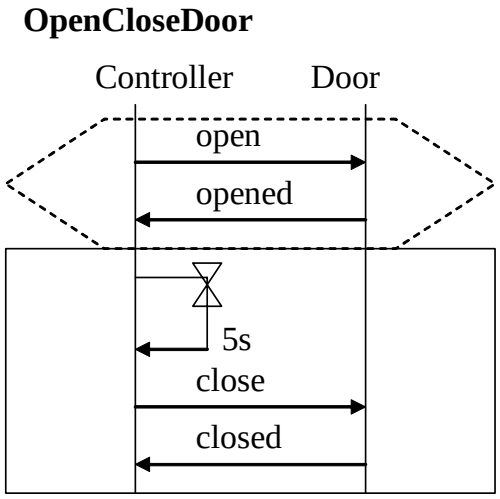


Figure 4.4

- **ReopenDoorFromInside**: If the user presses the button for the current floor when the lift is holding. The door re-opens. This scenario is illustrated in Figure 4.5.

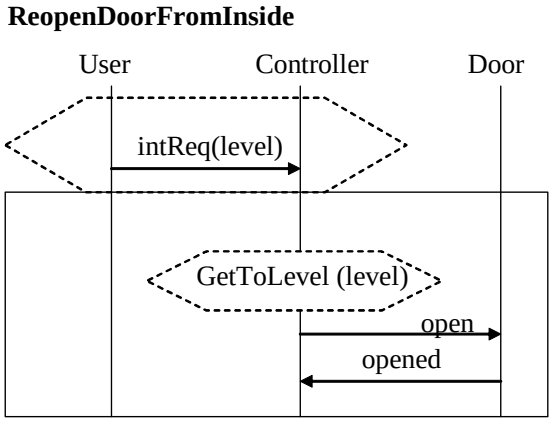


Figure 4.5

```
public bool GetToLevel(int level){
  return level == pos && status == 0;
}
```

- **ReopenDoorFromOutside**:When the lift is holding or stopped at certain floor, a user outside pressing the button (with the right direction if the lift is holding) reopens the door. This scenario is illustrated in Figure 4.6.

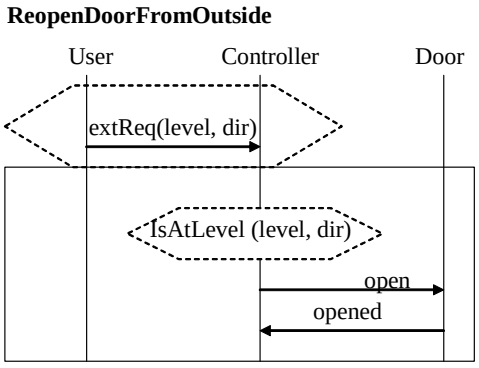


Figure 4.6

```
public bool IsAtLevel(int level, int dir){
      return level == pos &&
      ((status == 0 && dir == 0)
            || (status == 0 && dir == this.dir));
}
```

- **ServeInitIntRequest**: If when an internal request comes, the lift is stopped (holding and with no direction to move), the lift moves to the requested floor. This scenario is illustrated in Figure 4.7.

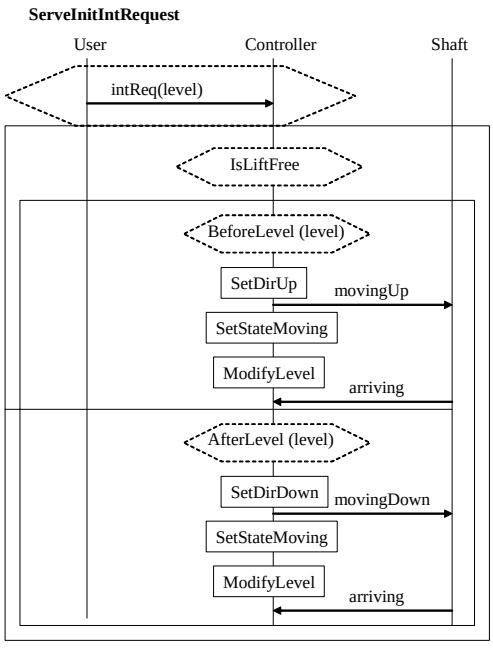


Figure 4.7

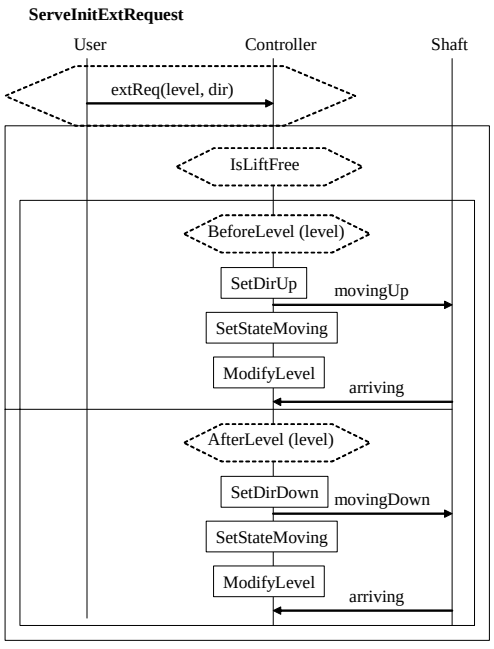


Figure 4.8

```
void SetDirUp () {
      dir = 1;}
void SetDirDown () {
      dir = -1;}
public bool BeforeLevel(int level){
      return level > pos;}
public bool AfterLevel(int level){
      return level < pos;}
```

- **ServeInitExtRequest**: If when an external request comes, the lift is stopped (holding and with no direction to move), the lift moves to the requested floor. This scenario is illustrated in Figure 4.8.

- **NoMoreRequest**: When there is no more request to serve, the lift stays still and sets its state to stopped (holding and with no direction to move). This scenario is illustrated in Figure 4.9.

```
void SetDirZero ()
{
      dir = 0;
}

boolean reqEmpty ()
{
      int sum = 0;
      for (int i = 0; i < N; i++)
      sum += int_req[i] + ext_req[i]*ext_req[i];
      return sum > 0;
}
```

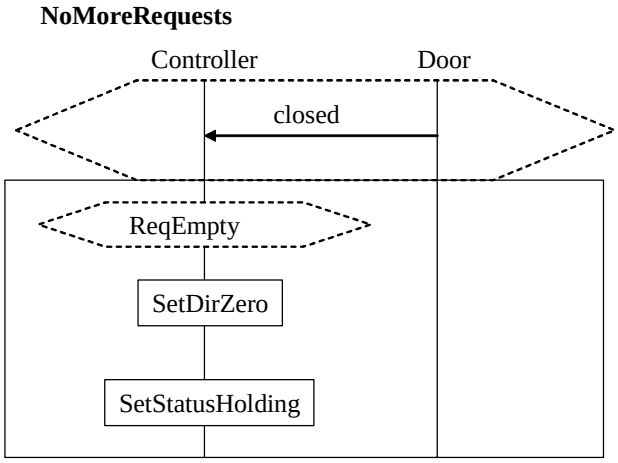


Figure 4.9

4.2. Synthesizer Module – Code Generation

First step in developing this project was to decide on how the generated programs should look like. A thread for every instance in the LSCs is needed because there is a distributed system. First idea would be create a thread in every class constructor, and the operations executed on that thread to be exactly the sequence of events that take place in a LSC. But an instance can exist in more than one LSC – can have different parts in different scenarios, each scenario described by a LSC. So, it is almost impossible to have a function that includes all scenarios in which an instance has a part, and that function to be the *life* of that instance.

The idea that solves this problem is to transform the LSC in state-machines, one corresponding for each instance in the LSC. When an instance has to perform some operations, depending on the activated charts and state in that chart, these operations will be chosen considering the active location (chart, state).

The easiest way of creating the state machine is to consider a new state before each event of receiving a message. A state is waiting for a message. The transitions between states are the operations performed after receiving the message. In this way, the life of an instance can be described like this:

"If that message arrives, a specific group of operations has to be performed."

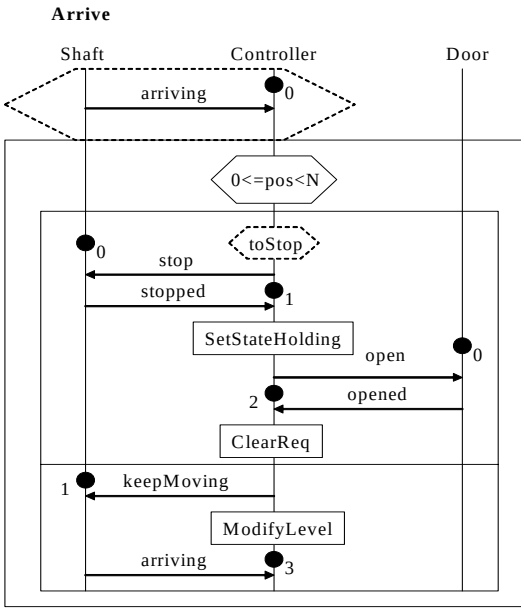


Figure 4.10

Figure 4.10 shows a LSC where are represented the states for each instance, marked on the life line. Figure 4.11 shows an example of how a state-machine should look like for the instance *Controller* (Lift System example).

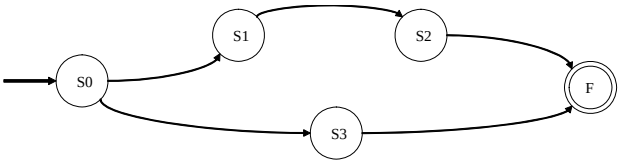


Figure 4.11

When an instance receives a message, the set of operations following this event in the LSC will be performed, and the machine state associated with that instance will pass in the next state, corresponding to the next receiving message. The problem that appears is that an instance may be active in more than one LSC at a time, may run in 2 state machines at a time, for example. Another possibility would be to have a new LSC that starts to run when receiving a message in another LSC.

A situation is presented in figure 4.12:

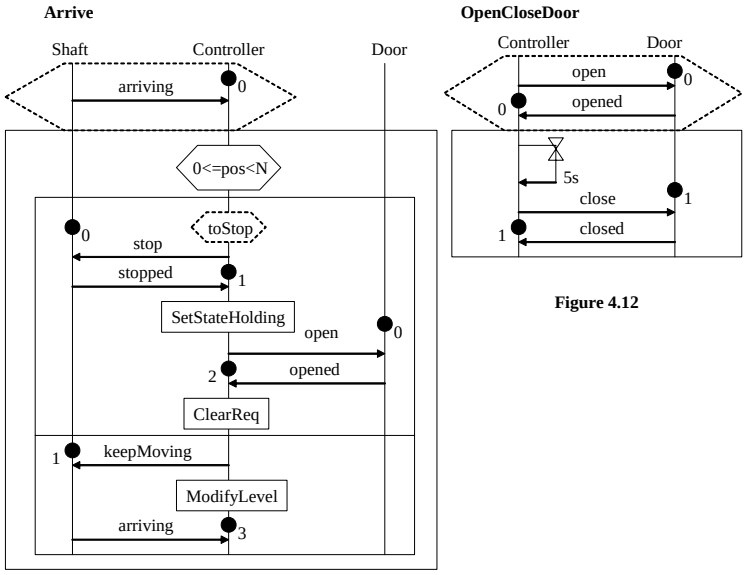


Figure 4.12

Controller sends message *Open* to *Door*. *Door* receives that message, in the LSC *Arrive*, state 0 and sends back message *Opened*. LSC *OpenCloseDoors* will be also activated (events in pre-chart have taken place). Instance *Controller* will be alive in 2 state machines – one for each active LSC. In this situation, the message *Open* was received twice by the instance *Door*: once in the LSC *Arrive* and once in the LSC *OpenCloseDoors*.

4.3. General Message Queue

The events for each instance have to be ordered in time due to the nature of LSC. This means that the messages has to be received in the same order as they have been sent and an instance has to finish all the operations generated by all the messages received before starting to treat a new message. The simple way of keeping the same order of messages is to use message queues.

The first idea used in [8] is to have a queue for every message sent between two instances, in every chart. The number of communicating channels is too big, and is almost impossible to treat the problem of receiving the same message in two LSCs.

The idea is to use a general message queue. The system specified earlier can be described using a general message queue. When an instance sends a message, this message will be put in this queue. There is a message queue for the whole system. All messages transmitted in all scenarios will be enqueued. A separate thread will take messages from the queue and announce every instance interested about the message. The implementation proposed is similar with an event based modeling. Every message has some event handlers registered. When a message is dequeued, the event is fired and a method to handle event will be called. This method is different for every instance (same message). There is one specialized thread for the message queue and one thread for each instance.

Message receiving is an event in the LSCs, when the instance transfers to the next state. For an instance there has to be declared a list of messages that will be handled (the other one will be ignored), in what scenarios (diagram) will a message appear and the state in which a message can be handled. Having the state and the scenario for a message received, the system can go to the next state (waiting for the next message), and running the actions in the LSC.

4.4. System Description

The whole mechanism described before is part of the system generated by the program. These components don't modify when the user wants to develop a new system, so it is a fixed part, that's way it is included in a library (.dll). The library will be copied in the directory that will contain the generated code files. The library is written in Spec# programming language, same as the classes defined in the LSCs.

4.4.1. Fixed Components

In figure 4.13 you can see a diagram of the fixed components:

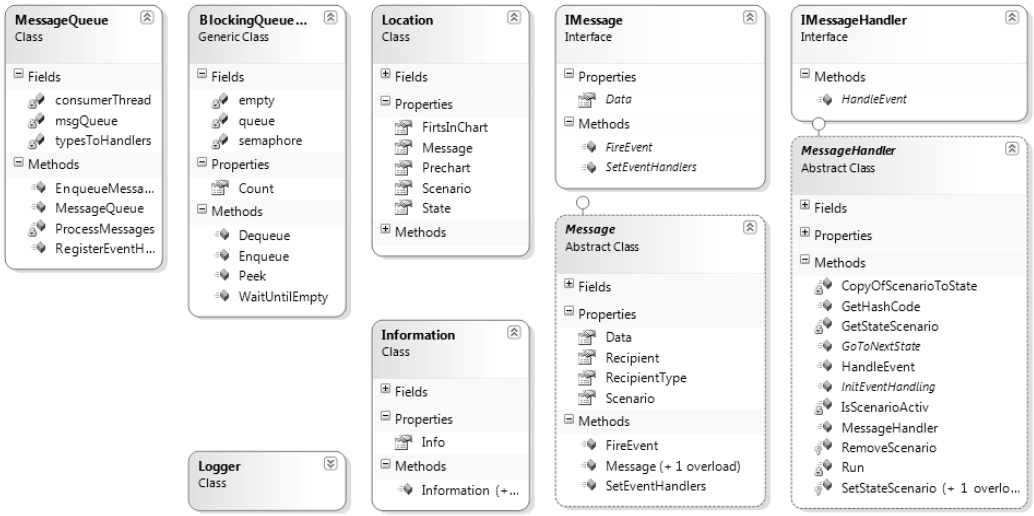


Figure 4.13

The algorithm of processing the messages is powered by the library above. Next it is explained the way this *engine* works.

Each message sent in the system must have a type. There may be more than one message of a specific type in the whole system. Referring to a message means referring to its type.

When an instance sends a message it means that the message is enqueued in the general message queue, which is an instance of the class *MessageQueue*(see figure 4.13). A *MessageQueue* has a thread *(consumerThread)* that consumes the messages in the blocking queue. For each message dequeued, depending on the type, the event handlers will be set. The message handlers are registered in the generated classes: for each instance register the messages that can be received in the LSCs' system. These are particular for each class, so they are not included in the *FixedComponents* part.

```
private void ProcessMessages()
{
   while (true)
   {
      IMessage/*!*/ msg = msgQueue.Dequeue();
      Type type = msg.GetType();
      // Event handling
      if (!typesToHandlers.Contains(type))
         continue;
      Object obj = typesToHandlers[type];
      /*^ assume obj != null; ^*/
      List<IMessageHandler>/*!*/ handlers=(List<IMessageHandler>)obj;
      msg.SetEventHandlers(handlers);
      msg.FireEvent();
   }
}
```

The message is handled in the method *FireEvent*, class *Message.* Each handler registered will treat the message, *HandleEvent* method.

```
public void FireEvent()
{
      foreach (IMessageHandler/*!*/ handler in handlers)
            handler.HandleEvent(this);
}
```

The method *HandleEvent* is contained by the class *MessageHandler*. Here, the message received is treated. There are two possibilities:

- The message is accepted and the operations associated with receiving that message will be performed.
- The message is rejected (ignored). No action will take place and the next message in the general queue will be handled.

How is the distinction between these two categories made? Each instance keeps track of the states and scenarios which are active at a time. The pair formed by the LSC (scenario) and the state of a particular instance it is called *location.* So, considering the *locations* available and activated for an instance, the decision of accepting or rejecting the message will be taken.

Searching through available locations, if one of the locations is active (actual location), then the message will be considered. If the actual state in a scenario is different from the available location, the message will be ignored. If the location is included in the pre-chart or it is the first state in the chart, the message will be handled.

```
public void HandleEvent(IMessage source)
{
 /*^ assume source != null; ^*/
 logger.append(" ");
 logger.append(" Handled event of type " + source.GetType());

 if (GetType() == ((Message)source).RecipientType
 && GetHashCode() != ((Message)source).Recipient)
 {
  logger.append(" Event not accepted - different recipient");
  return;
 }
 ((BlockingQueue<Location>)internalQueue).WaitUntilEmpty();
 copyOfScenarioToState = (Hashtable)CopyOfScenarioToState();

 Object obj = msgTypesToAvailableLocations[source.GetType()];
 /*^ assume obj != null; ^*/
 ArrayList /*!*/availableLocations = (ArrayList)obj;

 foreach (Location /*!*/availableLocation in availableLocations)
 {
  obj = scenarioToInternalQueue[availableLocation.Scenario];
  /*^ assume obj != null; ^*/
  ((BlockingQueue<Location>)obj).WaitUntilEmpty();

  ArrayList /*!*/actualStates=
     GetStateScenario(availableLocation.Scenario);

  int active = 0;
```

```
   for (int i = 0; i < actualStates.Count; i++)
   {
     obj = actualStates[i];
     /*^ assume obj != null; ^*/
     int actualState = (int)obj;
     Location actualLocation =
       new Location(availableLocation.Scenario, actualState, false);

     if (availableLocation.State == actualState
       && (actualState != 0
       || (actualState == 0
         && availableLocation.Scenario == ((Message)source).Scenario)
       || (actualState == 0 && availableLocation.Prechart)))
     {
       logger.append(" Event accepted in " + actualLocation);
       actualLocation.Message = (Message)source;
       internalQueue.Enqueue(actualLocation);
       obj = scenarioToInternalQueue[availableLocation.Scenario];
       /*^ assume obj != null ;^*/
       ((BlockingQueue<Location>)obj).Enqueue(actualLocation);
       active = 1;
     }
     else
       logger.append(" Event not accepted in " + actualLocation);
   }


   if (active == 0 && availableLocation.FirtsInChart
     && (availableLocation.Prechart
     || availableLocation.Scenario == ((Message)source).Scenario))
   {
     logger.append(" Event accepted in " + availableLocation);
     availableLocation.Message = (Message)source;
     internalQueue.Enqueue(availableLocation);
     obj = scenarioToInternalQueue[availableLocation.Scenario];
     /*^ assume obj != null; ^*/
     ((BlockingQueue<Location>)obj).Enqueue(availableLocation);
   }
  }
}
```

The structure Message contains two fields used to determine the recipient of the message: the type of the recipient (class) and the identifier of the recipient. The identifier is a hash code applied to the name of the instance – a string that is unique in the system. These fields are used to distinguish between a message sent to an instance, if in the system there are more than one instance of a specific type (class).

The algorithm described above takes place on the message queue's thread, the consumer thread. But the event handling, the operations that are triggered by a message receiving will take place on another thread: the instance thread. This is done to obtain the distributed system. Each instance is independent and they communicate through messages. Having more than one thread, a new problem appears: thread safety. All the actions need to have this property, and this is achieved using C# methods of synchronization: monitors and semaphores.

There are some internal queues used for facilitating the information transfer between the *consumerThread* and the instances' threads. When a message is accepted, it is put in the internal queue associated with the scenario and the intance. Before enqueueing a message, the queue has to be empty. This way it's for sure that there are

no unhandled messages that may affect the state of that instance (actual locations). Message is dequeued from the internal queue and treated on the particular thread.

Each instance has an internal thread designated to run all operations in the system. This thread is the life of the instance – operations are ordered like in the LSC because of the blocking queues used to keep the messages from sending time to handling time.

As shown in the next paragraph, the message is dequeued only after he was handled (the instance will pass to the naxt state).This is done to be sure that when the queue is empty, the instance's thread finished its job.

```
private void Run()
{
   while (true)
   {
      Location/*!*/ location = internalQueue.Dequeue();
      /*^ assume  location.Scenario != null; ^*/
      BlockingQueue<Location> q = (BlockingQueue<Location>)
            scenarioToInternalQueue[location.Scenario];
      /*^ assume  q!= null; ^*/
      q.Peek();
      GoToNextState(location);
      q = (BlockingQueue<Location>)
            scenarioToInternalQueue[location.Scenario];
      /*^ assume  q!= null; ^*/
      q.Dequeue();
   }
}
```

The library described above will be included in the generated code project. This is the most important part of the system. This engine offers stability and generalization. It is very easy to generate classes for any system because the most important part is already written. Only the details and particularities of the classes have now to be generated.

4.4.2. Generated Components

Instances in the LSC have a class, a type. For each class defined in the LSC specification a Spec# class will be generated. This class will extend the *MessageHandler* class, presented above.

These classes will contain particular details, deduced from LSC system:

- register event handlers – a list that contains the message types received by that class in the system.
- register a list of available locations for each type of message ever received in the LSC system.
- write the method that decides what operations will be executed after accepting a message.

The automated generation will not add the particular fields of a class, because they are not defined in the specification file used to describe the LSC system. Also, the body of the methods called during a LSC will not be generated. Only the declaration of the method will be present. The fields and method bodies will be written after generation, by the programmer.

4.5. Extending LSC with invariants and pre/post-conditions

Spec# is C# extended with contracts. If Spec# facilities are used, the LSC has to be enriched with new parts. One of the first ideas that come to mind is adding invariants to LSC.

Usually a LSC does not contain class definition, but in order to declare invariants, classes should be specified for each instance in the LSC. The XML Schema proposed for describing LSC contains a way of defining classes. A class is represented in the most simply way, just the name; but in the XML file, there is a special part of naming all classes used later in the LSCs. The editor proposed generates the XML and the class definitions inside automatically.

This part of declarations is very useful when generating the Spec# code, because the classes needed for all instances in the system definition have to be known from the very beginning. Having this part of class definition, the job of declaring invariants is easier. When constructing a new class, the invariant definition will be added.

A LSC includes calling different functions and verifying some conditions. As you can see in chapter 1, the XML describing a system contains an enumeration of the methods called for each instance. The class definition contains the function definitions. To make use of the Spec# facilities, the possibility of declaring preconditions and postconditions is included.

The invariants, pre/post-conditions consist of a method call (see chapter 3 for details). Usually, these structures are described by a condition (an expression with Boolean value). It's almost impossible to declare a syntax for LSC that includes Boolean expressions, that's why a call to a Boolean method is used for each condition (invariant, pre/post-condition or typical conditions in the LSCs). The class invariants and the pre/post-conditions are defined using the LSC editor.

For now, the project only covers invariants and pre/post-conditions, but future work may include adding new method contracts:

- exception and exceptional post-conditions
- not-null variable as function parameters
- non-null fields in classes

The LSC specification is quite poor because it does not allow defining class hierarchies. If the project will be extended to include class diagrams, then new Spec# facilities can be added.

4.6. LSC Editor

The editor helps the user to describe the system using LSCs. The output is a XML file, that follows the XML Schema Definition proposed for defining LSCs.

The component parts of the editor are:

- The graphical user interface
- The XSD auto-generated classes and their extensions
- The code generation part

The programs can be found in Appendix A.

4.6.1. Graphical User Interface

This part enhances LSCs definition. The functionalities of this part will be shortly described the next section. The interface displays the XML specification developing after adding the components through a graphical mode. Also, the XML file can be edited.

To define a new LSC system or to use a previous defined one, the user has to create a new project or to open one. If the user wants to change to working project, he can close the current project and open a new one. The files created in the project directory will be automatically included in the project.

After creating or opening an empty file, a new live sequence chart can be added in that file. A file corresponds to a LSC. After defining the whole system (many diagrams in one project), there is an option of creating the general XML file, that contains all the scenarios described. Only after having this XML file, you can proceed to code generation.

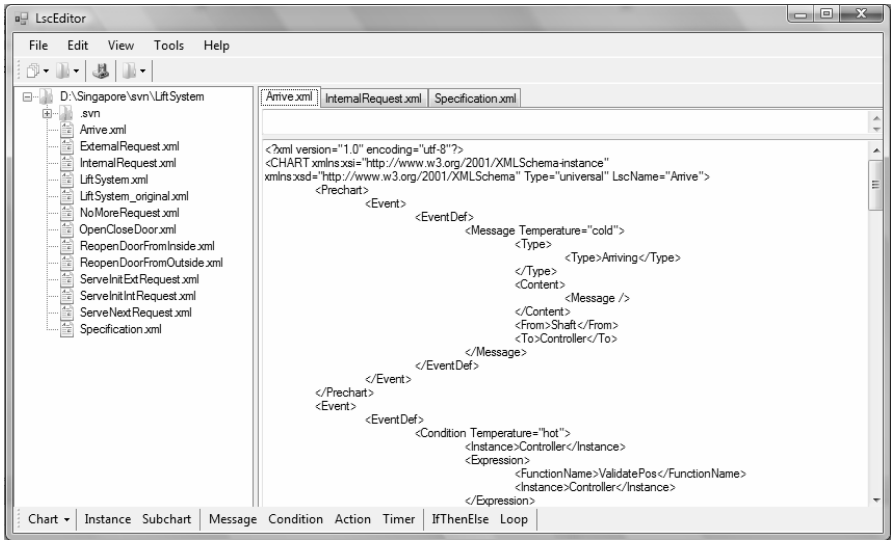


Figure 4.14

Figure 4.14 offers a general image of the Editor. In the left side you can see the opened project and the files contained. As in a classic GUI, there is a standard toolbar, with the general options of opening, closing a file or project. At the bottom of the image you can see the dedicated toolbar. There are the elements of a LSC that can be added in the opened file using a single click. The central part presents the XML file generated by the editor. For now, there is no visual image of the LSC.

For the *Chart* button there are 2 options: universal or existential. The chart will be added only in an empty file. The name of the chart will be the same with the name of the empty file. It can be changed by editing the XML file. In fact, every thing can be changed by editing the XML file.

When pressing *Instance* button the next form will appear:

Figure 4.15

While introducing the name of the instance, the class will be automated completed, but it can be changed. After adding some classes, a new instance may have same type (class) with another instance declared before.

The instances, classes, functions defined for a class, message types will be added in a file called *Specification.xml*. Adding an instance means in fact writing to this file. The modifications in this file are done only if the file is opened. If not, the file will be opened automatically. The changes can be seen immediately on the text box, but they are not automatically saved. The user will be asked if he wants to save the file before a new reading is done.

A new element in the LSC will be added as the last one in the XML file that describes the chart. That means that if the user doesn't want to edit the file, he has to introduce the events in the same order as they appear in the LSC. For example, in the Lift System example, *InternalRequest* chart first it should be added the message and after that the function call.

Next it is shown how a message is introduced. Clicking on the Message button in the editing toolbar causes the form in figure 4.16 to appear.

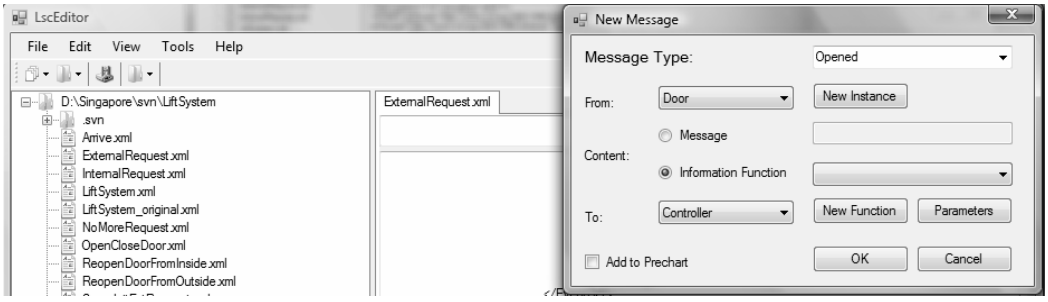


Figure 4.16

The message type will be added in the *Specification* file, but the instance of the message is added in the opened XML file (e.g. ExternalRequest.xml, figure 4.16). For a particular message the user can specify a string content to the message or a function that will be called to obtain the content of the message. Of course, th sender and the receiver of the message must be complited in order to have a correct message. There is an option that can include the message in the pre-chart. The message type may be one of the already defined ones, or may be a new one.

Figure 4.17 shows the form used for adding a condition to the LSC. The condition will be added at the end of the opened XML file. For a condition, the user has to choose or define a Boolean function that will be checked in the generated program. This function can be one of the already defined ones (for all instances). The condition may be added for one or more instances. There is the possibility of choosing between a *Hot Condition* (check box) or *a Cold Condition*.

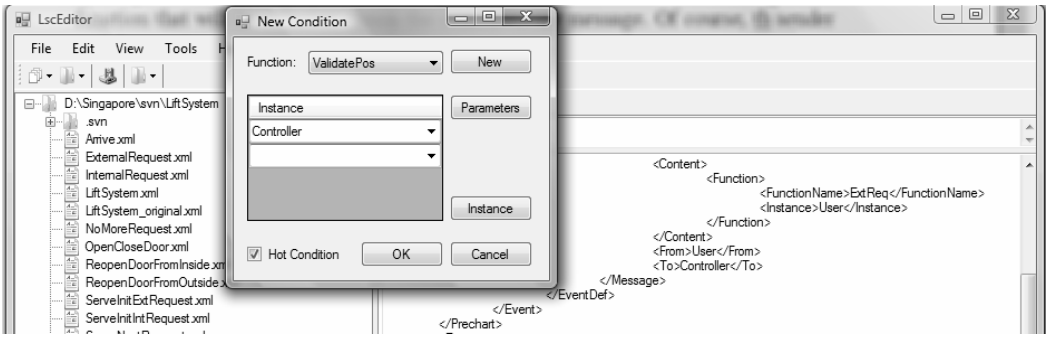


Figure 4.17

The user has to set the effective parameters that will be used when calling the Boolean function. Figure 4.18 shows the method used for specifying the parameters. Depending on how many formal parameters the function has, the number of tab pages will be added to the form. The may be specified a value for the parameter or a variable name that will be included in the instance's class.

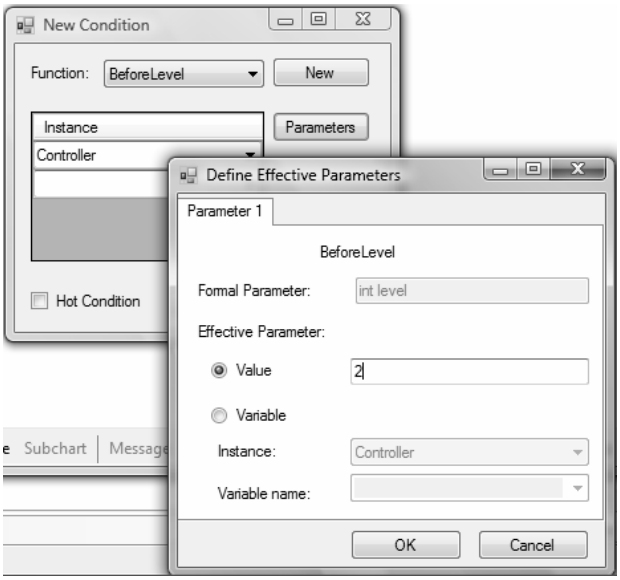


Figure 4.18

When adding a new function, the definition will be written in the *Specification.xml* file, included in the class that contains the respective method. The definition of a function is presented in figure 4.19.

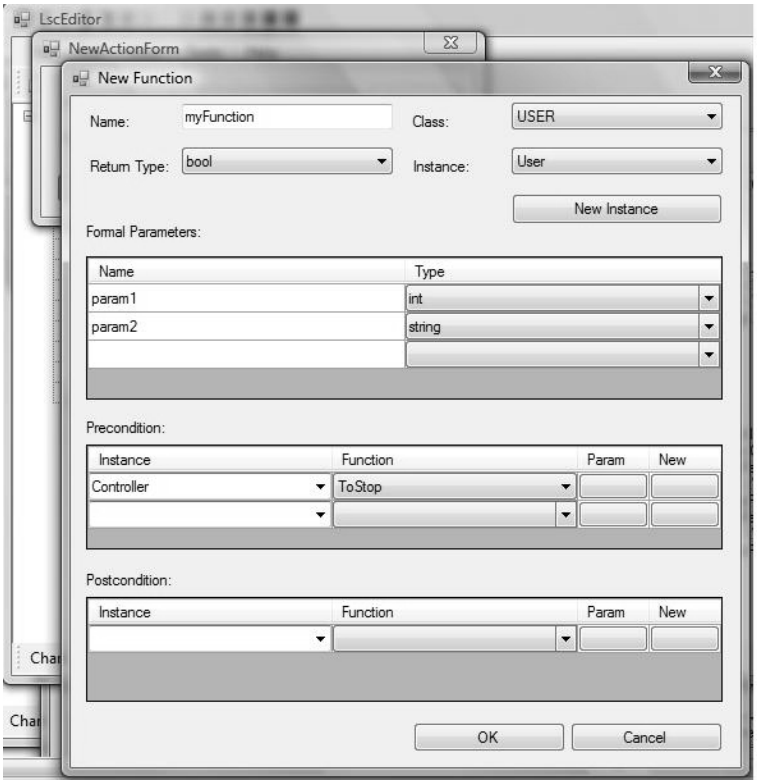


Figure 4.19

The user has to specify the name of the function, the class that contains the new method, the formal parameters and their types. The types of the formal parameters are chosen from a list that contains only the types declared in the XSD for LSC. Pre-and post-conditions can be added for a function. The conditions are given through a Boolean method call. This Boolean function has to be a pure one (see Spec# specification, chapter 3).

Figure 4.20 shows the way of adding a new action in the LSC. By clicking on the button *Action*, the form will appear. Here the user may choose the function that represents the action or define a new one. The parameters for this function are completed next. The action will be added as the last event in the LSC.

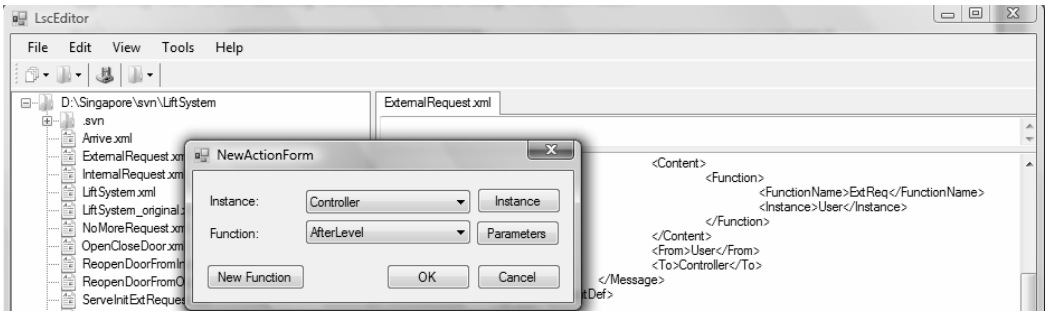


Figure 4.20

One of the simple events that can be added to a LSC is the *Timer*. For defining the timer, the user has to introduce the instances that waits and for how long it waits. In the example shown in figure 4.21 the *Controller* will wait for 5 seconds.

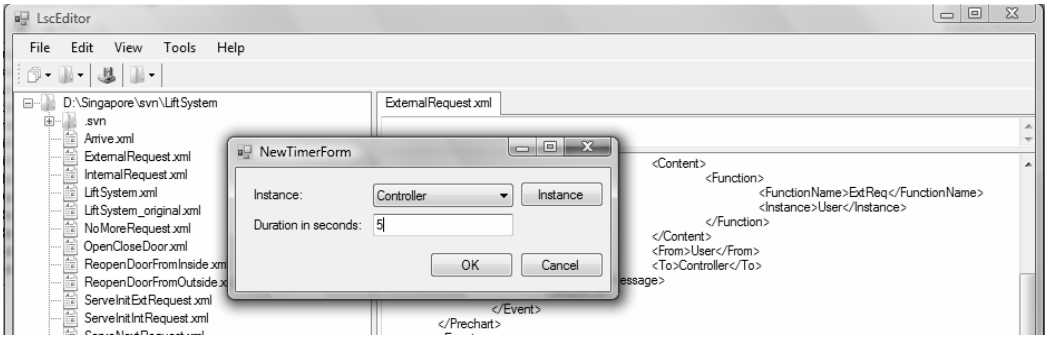


Figure 4.21

Messages, conditions actions and timers are the simple events that can be added on a LSC. There are also some complex events that may be included in a LSC. First structure that comes to mind is the *if-then-else* block or the *loop* block. My editor allows the introduction of these structures.

An *if-then-else* structure is composed of a condition and two subcharts: *then* – block and *else* – block. A subcharts is a group of events defined separately. This events may be simple eventsor complex events.

An *loop* structure is composed of a condition and a subchart, the block of operations the will be performed if the condition is satisfied.

Figure 4.22 presents the form used for defining an if-*then-else* block. It's very similar to the primary editor, but the new text box contains only the subchart definition. First only the *condition* button is active, forcing the user to add a condition. For *then*-block the user can add a subchart or a structure. When the subchart is finished, by clicking *OK* the structure will be placed in the primary editor. The *loop* or *subchart* forms are similar with the one presented.

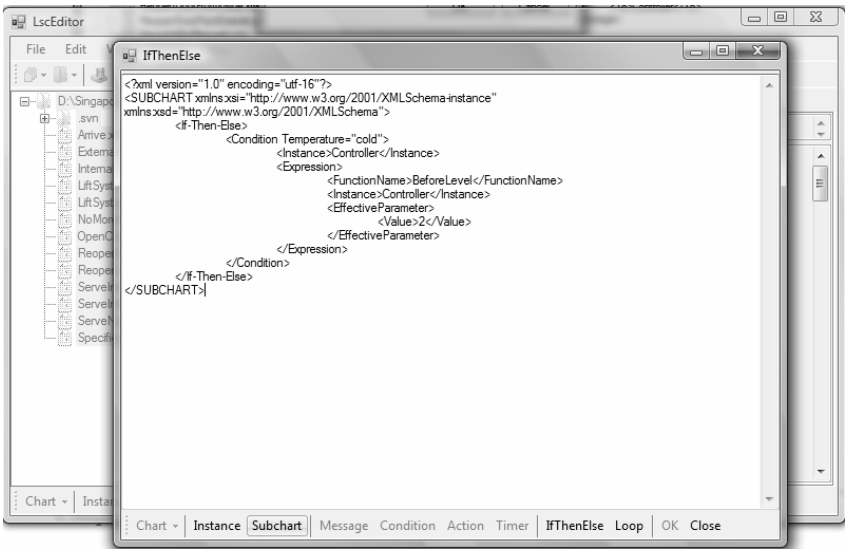


Figure 4.22

4.6.2. The XSD auto-generated Classes and their Extensions

There are different tools that make easier XML file generation. XML Schema Definition Tool (Xsd.exe) is a tool that comes with Visual Studio .NET 2005. The XML Schema Definition tool generates XML schema or common language runtime classes from XDR, XML, and XSD files, or from classes in a runtime assembly. Conversion from schema file to C# source file is done by issuing the following command:

```
xsd /out: LscClasses  /namespace:LscClasses /c LSCXSD.xsd"
```

This tool was useful in generating the classes from the XML Schema Definition (XSD) described in the first chapter. After that the objects associated with the events in the LSCs were created. The XML file generation was solved by writing in a file the objects created. This was done using XML serialization.

The primary purpose of XML serialization in the .NET Framework is to enable the conversion of XML documents and streams to common language runtime objects and vice versa. Serialization of XML to common language runtime objects enables one to convert XML documents into a form where they are easier to process using conventional programming languages. On the other hand, serialization of objects to XML facilitates persisting or transporting the state of such objects in an open, standards compliant and platform agnostic manner.

XML serialization in the .NET Framework supports serializing objects as XML that conforms to a specified. During XML serialization, only the public properties and fields of an object are serialized. Also, type fidelity is not always preserved during XML serialization. This means that if, for instance, you have a Book object that exists in the Library namespace, there is no guarantee that it will be deserialized into an object of the same type. However, this means that objects serialized using the XML serialization in the .NET Framework can be shipped from one machine to the other without requiring that the original type be present on the target machine or that the XML is even processed using the .NET Framework. XML serialization of objects is a useful mechanism for those who want to provide or consume data using platform agnostic technologies such as XML and SOAP.

XML documents converted to objects by the XML serialization process are strongly typed. Data type information is associated with the elements and attributes in an XML document through a schema written in the W3C XML Schema Definition (XSD) Language. The data type information in the schema allows the XmlSerializer to convert XML documents to strongly typed classes.

After generating the classes, they were extended by adding some useful methods: *ToString* – for some of them (when needed) and the most important one: *ToCode. ToCode* is the method called to generate Spec# code.

4.6.3. Code Generation

The code generation is available in the LscEditor in the menu *Tools.* A project has to be opened and the XML file has to be generated before calling the *CodeGeneration* option.

The generator will ask will about the name of the XML specification, the fixed components library and the output directory. For each message type and class in the LSC specification will be defined a Spec# class. Also, it will be generated a file that contains the *void main* function. Here will be defined the instances in the system and will be enqueued the messages that represent the action from outside (the user, for List System example).

The actual code generation is done by calling the methods *ToCode* for different instances, classes or message types:

```
public void GenerateCode()
{
   spec = GetLscSpec();
   if (spec == null)
      return;
   if (spec.MessageTypes != null )
      spec.MessageTypes.ToCode(dir);
   if (spec.Classes != null)
      spec.Classes.ToCodeStart(dir);
   if (spec.Chart != null)
      foreach (CHART c in spec.Chart)
         spec.Classes.ToCode(spec, c);
   if (spec.Classes != null)
      spec.Classes.ToCodeEnd();
   if (spec.Instances != null)
      spec.Instances.ToCode(dir);
}
```

The .NET Framework includes a mechanism called the Code Document Object Model (CodeDOM) that enables developers of programs that emit source code to generate source code in multiple programming languages at run time, based on a single model that represents the code that has to be rendered.

To represent source code, CodeDOM elements are linked to each other to form a data structure known as a CodeDOM graph, which models the structure of some source code.

The **System.CodeDom** namespace defines types that can represent the logical structure of source code, independent of a specific programming language. The **System.CodeDom.Compiler** namespace defines types for generating source code from CodeDOM graphs and managing the compilation of source code in supported languages. Compiler vendors or developers can extend the set of supported languages.

Language-independent source code modeling can be valuable when a program needs to generate source code for a program model in multiple languages or for an uncertain target language. For example, some designers use the CodeDOM as a language abstraction interface to produce source code in the correct programming language. Spec# assertion are generated as comments, using CodeDOM.

The generated code is similar with the description of what was wanted to generate. The code obtained using this generator, for the Lift System specification can be found in appendix B.

Here is an example of how the program generates the code for an element of type *Condition* in the LSC. Code generation is similar to this example:

```
CodeConditionStatement cond;
// new condition statement
cond = new CodeConditionStatement(
   new CodeBinaryOperatorExpression(
      this.Expression.ToCode(),
      CodeBinaryOperatorType.IdentityEquality,
      new CodePrimitiveExpression(false)),
   new CodeStatement[] {
      new CodeExpressionStatement(
         new CodeMethodInvokeExpression(
            new CodeMethodReferenceExpression(
               new CodeThisReferenceExpression(),
               "RemoveScenario"),
            new CodeExpression[] {
               new CodeVariableReferenceExpression("location")})})});

//if the condition is hot, we add an assert; else a return statement
if (this.Temperature == TEMPERATURE.cold)
   cond.TrueStatements.Add(new CodeMethodReturnStatement());
else
   cond.TrueStatements.Add(
      new CodeExpressionStatement(
         new CodeSnippetExpression(
            "Debug.Assert( " + this.Expression.ToString() + ")")));

//add the condition to all the active states
if (events.Contains(chart.classActiveState))
   ((CodeStatementCollection)events[chart.classActiveState]).
      Add(cond);
else
   events.Add(chart.classActiveState,
      new CodeStatementCollection(
      new CodeStatement[]{ cond }));
```

4.7. System Simulation

The purpose of this code generation is to be able to simulate the functioning of the system described through LSCs. This is useful in proving that the system is correct defined. Future work may include constructing a verifier for the generated code.

The exterior actions will be introduced in the simulation using an input file. This file contains the messages that trigger the simulation. The output will be a log file that contains the actions that took place in the fixed part of the generation (the

library of *FixedComponents*). There are no logs from the generated part because is impossible to generate lines that logs messages during code generation.

Let's try to simulate the execution of the Lift system. Take a look over the complete example presented before. If the input file contains two messages:

- An internal request – an user that press the lift button from the inside to go to level 2.
- An external request – an user that press the lift button from level 1, to go up.

The lift is at level 1 in the beginning and it has to go to level 2 and after that return to level 1. The output file is listed and explained (the text beginning with "▪" are additional explanations) next:

```
-------------------------------------------------------------
Thread      Instance     Action
-------------------------------------------------------------

▪ The message IntReq (user's request for level 2) is accepted and handled (dequeue
from internal queue) in 3 scenarios
MessQueue   Controller   Handled event of type GeneratedComponents.IntReq
MessQueue   Controller   Event accepted in location (InternalRequest, 0)
MessQueue   Controller   Event accepted in location (ReopenDoorFromInside, 0)
Controller  Controller   GeneratedComponents.IntReq dequeued from internal queue
MessQueue   Controller   Event accepted in location (ServeInitIntRequest, 0)
Controller  Controller   GeneratedComponents.IntReq dequeued from internal queue
Controller  Controller   GeneratedComponents.IntReq dequeued from internal queue

▪ The condition in scenario ServeInitIntRequest is true (the lift is not moving), so
the lift begins MovingUp. The position of the lift is changed, one level up.
MessQueue   Shaft        Handled event of type GeneratedComponents.Movingup
Controller  Controller   Add location with scenario: location (ServeInitIntRequest, 1)
MessQueue   Shaft        Event accepted in location (ServeInitIntRequest, 0)
Shaft       Shaft        GeneratedComponents.Movingup dequeued from internal queue

▪ The shaft sends the message Arriving in ServeInitIntRequest and the Arrive LSC is
activated (message accepted in state 0)
MessQueue   Controller   Handled event of type GeneratedComponents.Arriving
MessQueue   Controller   Event accepted in location (Arrive, 0)
Controller  Controller   GeneratedComponents.Arriving dequeued from internal queue
Controller  Controller   Add location with scenario: location (Arrive, 1)
MessQueue   Controller   Event accepted in location (ServeInitIntRequest, 1)
Controller  Controller   GeneratedComponents.Arriving dequeued from internal queue
Controller  Controller   Remove location: location (ServeInitIntRequest, 1)
Controller  Controller   Remove location with scenario: ServeInitIntRequest
MessQueue   Controller   Event not accepted in location (ServeInitIntRequest, 1)

▪ Lift arrived at level 2, the condition in chart Arrive are met, so the lift stops
and opens the doors.
MessQueue   Shaft        Handled event of type GeneratedComponents.Stop
MessQueue   Shaft        Event accepted in location (Arrive, 0)
Shaft       Shaft        GeneratedComponents.Stop dequeued from internal queue

▪ Lift stopped.
MessQueue   Controller   Handled event of type GeneratedComponents.Stopped
MessQueue   Controller   Event accepted in location (Arrive, 1)
Controller  Controller   GeneratedComponents.Stopped dequeued from internal queue

▪ The external request comes: level 1, going up. Message accepted in 3 charts. This
request is not yet handled and no other actions are taken for moment. During this
time, the controller handles previous message.
MessQueue   Controller   Handled event of type GeneratedComponents.ExtReq
MessQueue   Controller   Event accepted in location (ExternalRequest, 0)
Controller  Controller   Add location: location (Arrive, 2)
Controller  Controller   Remove location: location (Arrive, 1)
Controller  Controller   GeneratedComponents.ExtReq dequeued from internal queue
MessQueue   Controller   Event accepted in location (ReopenDoorFromOutside, 0)
MessQueue   Controller   Event accepted in location (ServeInitExtRequest, 0)

▪ The door receives the message Open, accepted in 2 scenarios. For chart Arrive, the
response Opened will be sent, but for chart OpenCLoseDoor, no message will be sent
because the actions are part of the pre-chart.
```

```
MessQueue   Door         Handled event of type GeneratedComponents.Open
MessQueue   Door         Event accepted in location (Arrive, 0)
Controller  Controller   GeneratedComponents.ExtReq dequeued from internal queue
Controller  Controller   GeneratedComponents.ExtReq dequeued from internal queue
Door        Door         GeneratedComponents.Open dequeued from internal queue
MessQueue   Door         Event accepted in location (OpenCloseDoor, 0)
Door        Door         GeneratedComponents.Open dequeued from internal queue

▪ Message Opened is handled in 2 scenarios. The OpenCloseDoor chart is now completely
activated.
Door        Door         Add location with scenario: location (OpenCloseDoor, 1)
MessQueue   Controller   Handled event of type GeneratedComponents.Opened
MessQueue   Controller   Event accepted in location (Arrive, 2)
MessQueue   Controller   Event accepted in location (OpenCloseDoor, 0)
Controller  Controller   GeneratedComponents.Opened dequeued from internal queue
Controller  Controller   Remove location: location (Arrive, 2)
Controller  Controller   Remove location with scenario: Arrive
Controller  Controller   GeneratedComponents.Opened dequeued from internal queue

▪ The doors are closed after 5 seconds.
Controller  Controller   Add location with scenario: location (OpenCloseDoor, 1)
MessQueue   Door         Handled event of type GeneratedComponents.Close
MessQueue   Door         Event accepted in location (OpenCloseDoor, 1)
Door        Door         GeneratedComponents.Close dequeued from internal queue
Door        Door         Remove location: location (OpenCloseDoor, 1)
Door        Door         Remove location with scenario: OpenCloseDoor

▪ Message Closed is handled 3 scenarios.
MessQueue   Controller   Handled event of type GeneratedComponents.Closed
MessQueue   Controller   Event accepted in location (NoMoreRequest, 0)
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue
MessQueue   Controller   Event accepted in location (OpenCloseDoor, 1)
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue
Controller  Controller   Remove location: location (OpenCloseDoor, 1)
Controller  Controller   Remove location with scenario: OpenCloseDoor
MessQueue   Controller   Event accepted in location (ServeNextRequest, 0)
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue

▪ The ServeNextRequest chart is activated and a new message is received. The lift
starts moving, going to level 1 (external request). The position is updated (one level
down)
Controller  Controller   Add location with scenario: location (ServeNextRequest, 3)
MessQueue   Shaft        Handled event of type GeneratedComponents.TurnAroundMoving
MessQueue   Shaft        Event accepted in location (ServeNextRequest, 1)
Shaft       Shaft        GeneratedComponents.TurnAroundMoving dequeued from internal
queue

▪ The lift is arriving at level 1, chart Arrive is activated again. The lift stops and
opens the doors.
MessQueue   Controller   Handled event of type GeneratedComponents.Arriving
MessQueue   Controller   Event accepted in location (Arrive, 0)
Controller  Controller   GeneratedComponents.Arriving dequeued from internal queue
Controller  Controller   Add location with scenario: location (Arrive, 1)
MessQueue   Controller   Event not accepted in location (ServeNextRequest, 3)
MessQueue   Controller   Event accepted in location (ServeNextRequest, 3)
Controller  Controller   GeneratedComponents.Arriving dequeued from internal queue

Controller  Controller   Remove location: location (ServeNextRequest, 3)
MessQueue   Shaft        Handled event of type GeneratedComponents.Stop
Controller  Controller   Remove location with scenario: ServeNextRequest
MessQueue   Shaft        Event accepted in location (Arrive, 0)
Shaft       Shaft        GeneratedComponents.Stop dequeued from internal queue

MessQueue   Controller   Handled event of type GeneratedComponents.Stopped
MessQueue   Controller   Event accepted in location (Arrive, 1)
Controller  Controller   GeneratedComponents.Stopped dequeued from internal queue

Controller  Controller   Add location: location (Arrive, 2)
MessQueue   Door         Handled event of type GeneratedComponents.Open
Controller  Controller   Remove location: location (Arrive, 1)
MessQueue   Door         Event accepted in location (Arrive, 0)
Door        Door         GeneratedComponents.Open dequeued from internal queue
MessQueue   Door         Event accepted in location (OpenCloseDoor, 0)
Door        Door         GeneratedComponents.Open dequeued from internal queue
Door        Door         Add location with scenario: location (OpenCloseDoor, 1)
```

```
MessQueue   Controller   Handled event of type GeneratedComponents.Opened
MessQueue   Controller   Event accepted in location (Arrive, 2)
Controller  Controller   GeneratedComponents.Opened dequeued from internal queue
Controller  Controller   Remove location: location (Arrive, 2)
Controller  Controller   Remove location with scenario: Arrive
MessQueue   Controller   Event accepted in location (OpenCloseDoor, 0)
Controller  Controller   GeneratedComponents.Opened dequeued from internal queue
Controller  Controller   Add location with scenario: location (OpenCloseDoor, 1)

MessQueue   Door         Handled event of type GeneratedComponents.Close
MessQueue   Door         Event accepted in location (OpenCloseDoor, 1)
Door        Door         GeneratedComponents.Close dequeued from internal queue
Door        Door         Remove location: location (OpenCloseDoor, 1)
Door        Door         Remove location with scenario: OpenCloseDoor
```

▪ After closing the doors, no more requests are available, so the lift waits.

```
MessQueue   Controller   Handled event of type GeneratedComponents.Closed
MessQueue   Controller   Event accepted in location (NoMoreRequest, 0)
MessQueue   Controller   Event accepted in location (OpenCloseDoor, 1)
MessQueue   Controller   Event accepted in location (ServeNextRequest, 0)
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue
Controller  Controller   Remove location: location (OpenCloseDoor, 1)
Controller  Controller   Remove location with scenario: OpenCloseDoor
Controller  Controller   GeneratedComponents.Closed dequeued from internal queue
```

Conclusions

Live Sequence Charts (LSC), proposed by Damm and Harel [4], has been rapidly recognized as a rather rich and useful extension of MSC. A large set of constructs has been provided for specifying not only possible behaviors, but also mandatory behaviors. For instance, a universal chart, typically preceded with a pre-chart, specifies global mandatory behaviors, i.e., once a system run matches the pre-chart, the subsequent behavior must follow the main chart. On the level of a single chart, events and conditions and locations are also labeled with modalities. LSC provides structuring constructs as well, for example sub-chart, branching and iteration, to build scenarios hierarchically.

In a nutshell, LSC offers a far more powerful means for stating requirements for complex systems than MSC. Because it distinguishes mandatory behaviors from possible ones (in contrast to MSC), it serves as an excellent basis of mechanized analysis of scenarios, for instance, the study of the distributed synthesis and verification problem.

Mechanized generation of programs from high-level specification is an important yet challenging approach to correct programs. In this work, we developed a method to synthesize correct-by-construction prototypes from scenarios. The key idea is of transforming the life of an instance in a state machine.

The uniqueness of the approach is that system *engine* is not generated, but is part of the fixed components. This way the system is less exposed to mistakes that may appear during code generation. The representation in based on the tool support that Spec# offers. Spec# provides a rich set of compositional constructs. A systematic way of synthesizing distributed processes directly from LSC is proposed. By using a bounded set of special synchronization events, the behaviors of each object can be decided locally without constructing the global state machine. Therefore, our work preserves the structure of the LSC specification.

There are a number of possible future works. The first is to extend the synthesis to systems with complex data structures (like class hierarchies). LSC lacks expressiveness to capture complicated data structures and functional requirements. It is ineffective on specifying systems with not only intensive interactions but also complicated data requirements. Therefore, it is necessary to extend LSC with data-related constructs. For now, the project only covers invariants and pre/post-conditions, but future work may include adding new method contracts: exception and exceptional post-conditions, not-null variable as function parameters and non-null fields in classes.

Bibliography

1. Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, andWolfram Schulte: ***Verification of object-oriented programs with invariants***. Journal of Object Technology, 2004.

2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte: ***The Spec# Programming System: An Overview***

3. Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke: ***Live Sequence Charts - an introduction to lines, arrows and strange boxes in the context of formal verification***.

4. W. Damm and D. Harel: ***LSCs: Breathing life into message sequence charts. Formal Methods in System Design***, 2001.

5. ITU-TS (International Telecommunication Union - Telecommunication Standardization Sector) Recommendation Z.120: ***Message Sequence Chart (MSC)***, ITU-TS, Geneva, 1996.

6. K. Rustan M. Leino and Wolfram Schulte: ***Exception safety for C#.*** In SEFM 2004—Second International Conference on Software Engineering and Formal Methods, pages 218–227. IEEE, September 2004.

7. J. Sun: ***Complementary Formalisms – Synthesis, verification and Visualisation***, PhD Thesis, 2006.

8. J. Sun and Jin Song Dong: ***Design Synthesis from Interaction and State-Based Specification***, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 32, No. 6, 2006.

9. J. Sun and J. S. Dong: ***From Live Sequence Charts to Distributed Implementation***

10. J. Sun and J. S. Dong: ***Synthesis of Distributed Processes from Scenario-based Specifications***. Formal Methods 2005 (FM'05), University of Newcastle upon Tyne, UK.

11. J. Sun and J. S. Dong: ***Model Checking Live Sequence Charts***. The 10th International Conference on Engineering of Complex Computer Systems (ICECCS'05), Shanghai, China, 2005.

*** http://research.microsoft.com/specsharp/

*** http://msdn2.microsoft.com/en-us/library/microsoft.csharp.aspx

*** http://msdn2.microsoft.com/en-us/library/system.codedom.aspx

*** http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.71).aspx

*** http://msdn2.microsoft.com/en-us/library/ms950721.aspx

*** http://msdn2.microsoft.com/en-us/library/650ax5cx.aspx

Appendix A

LSC2Code.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.IO;
using LscClasses;
using System.Xml.Serialization;
using System.Xml;
namespace LSC2Code
{
  public class LSC2Code
  {
  string dir;

  string xmlPath;
  string dllPath;
  LscSpec spec;
  public LSC2Code(string dir, string xmlPath, string dllPath)
  {
    this.dir = dir;
    this.xmlPath = xmlPath;
    this.dllPath = dllPath;
  }
  public LscSpec GetLscSpec()
  {
    Process process = new Process();
     process.StartInfo.FileName = "cmd";
     process.StartInfo.Arguments =
         "/c copy \"" + dllPath+ "\" \""+ dir+"\"";
     process.StartInfo.UseShellExecute = false;
     process.Start();
     process.WaitForExit();
    StreamReader reader = new StreamReader(xmlPath);
    spec = null;
    try
    {
    XmlSerializer serializer = new XmlSerializer(typeof (LscSpec));
    spec = (LscSpec)serializer.Deserialize(new XmlTextReader(reader));
    }
    catch (Exception e)
    {
    Console.WriteLine(e.Message);
    return null;
    }
    reader.Close();
    return spec;
  }
  public void GenerateCode()
  {
    spec = GetLscSpec();
    if (spec == null)
      return;
    if (spec.MessageTypes != null )
      spec.MessageTypes.ToCode(dir);

    if (spec.Classes != null)
    spec.Classes.ToCodeStart(dir);

    if (spec.Chart != null)
     foreach (CHART c in spec.Chart)
       spec.Classes.ToCode(spec, c);

    if (spec.Classes != null)
     spec.Classes.ToCodeEnd();

    if (spec.Instances != null)
     spec.Instances.ToCode(dir);
  }
  }
}
```

LSCXSD.cs

```
//------------------------------------------------------------------------------
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:2.0.50727.312
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
//
// This source code was auto-generated by xsd, Version=2.0.50727.42.
//
namespace LscClasses {
  using System.Xml.Serialization;
/// <remarks/>
  [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
  [System.SerializableAttribute()]
  [System.Diagnostics.DebuggerStepThroughAttribute()]
  [System.ComponentModel.DesignerCategoryAttribute("code")]
  [System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
  [System.Xml.Serialization.XmlRootAttribute(Namespace="", IsNullable=false)]
  public partial class LscSpec {
    private TYPEDEF[] typeDefField;
    private CLASSDEFS classesField;
    private MESSAGETYPEDEFS messageTypesField;
    private INSTDEFS instancesField;
    private CHART[] chartField;
    private string specNameField;
    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("TypeDef")]
    public TYPEDEF[] TypeDef {
      get {
        return this.typeDefField;
      }
      set {
        this.typeDefField = value;
      }
    }
    /// <remarks/>
    public CLASSDEFS Classes {
      get {
        return this.classesField;
      }
      set {
        this.classesField = value;
      }
    }
    /// <remarks/>
    public MESSAGETYPEDEFS MessageTypes {
      get {
        return this.messageTypesField;
      }
      set {
        this.messageTypesField = value;
      }
    }

    /// <remarks/>
    public INSTDEFS Instances {
      get {
        return this.instancesField;
      }
      set {
        this.instancesField = value;
      }
    }
    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("Chart")]
    public CHART[] Chart {
      get {
        return this.chartField;
      }
      set {
        this.chartField = value;
      }
```

```
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string SpecName {
    get {
      return this.specNameField;
    }
    set {
      this.specNameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class TYPEDEF {
  private object itemField;
  private string nameField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute(
     "BoundedInteger", typeof(BOUNDEDINT))]
  [System.Xml.Serialization.XmlElementAttribute("Enumeration", typeof(ENUMERATION))]
  public object Item {
    get {
      return this.itemField;
    }
    set {
      this.itemField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class BOUNDEDINT {
  private int lowerBoundField;
  private int upperBoundField;
  /// <remarks/>
  public int LowerBound {
    get {
      return this.lowerBoundField;
    }
    set {
      this.lowerBoundField = value;
    }
  }
  /// <remarks/>
  public int UpperBound {
    get {
      return this.upperBoundField;
    }
    set {
      this.upperBoundField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class COREGION {
  private EVENTDEF[] eventDefField;
```

```
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("EventDef")]
  public EVENTDEF[] EventDef {
    get {
      return this.eventDefField;
    }
    set {
      this.eventDefField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class EVENTDEF {
  private object itemField;
  private ItemChoiceType itemElementNameField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Action", typeof(FUNCTIONCALL))]
  [System.Xml.Serialization.XmlElementAttribute("Condition", typeof(CONDITION))]
  [System.Xml.Serialization.XmlElementAttribute("Message", typeof(MESSAGE))]
  [System.Xml.Serialization.XmlElementAttribute("ResetTimer", typeof(string))]
  [System.Xml.Serialization.XmlElementAttribute("SetTimer", typeof(SETTIMER))]
  [System.Xml.Serialization.XmlElementAttribute("Subchart", typeof(SUBCHART))]
  [System.Xml.Serialization.XmlElementAttribute("TimeOut", typeof(string))]
  [System.Xml.Serialization.XmlChoiceIdentifierAttribute("ItemElementName")]
  public object Item {
    get {
      return this.itemField;
    }
    set {
      this.itemField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlIgnoreAttribute()]
  public ItemChoiceType ItemElementName {
    get {
      return this.itemElementNameField;
    }
    set {
      this.itemElementNameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class FUNCTIONCALL {
  private string functionNameField;
  private string instanceField;
  private EFFECTIVEPARAM[] effectiveParameterField;
  /// <remarks/>
  public string FunctionName {
    get {
      return this.functionNameField;
    }
    set {
      this.functionNameField = value;
    }
  }
  /// <remarks/>
  public string Instance {
    get {
      return this.instanceField;
    }
    set {
      this.instanceField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("EffectiveParameter")]
  public EFFECTIVEPARAM[] EffectiveParameter {
```

```
    get {
      return this.effectiveParameterField;
    }
    set {
      this.effectiveParameterField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class EFFECTIVEPARAM {
  private object itemField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Value", typeof(string))]
  [System.Xml.Serialization.XmlElementAttribute("Variable", typeof(VARIABLE))]
  public object Item {
    get {
      return this.itemField;
    }
    set {
      this.itemField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class VARIABLE {
  private string variableNameField;
  private string instanceField;
  /// <remarks/>
  public string VariableName {
    get {
      return this.variableNameField;
    }
    set {
      this.variableNameField = value;
    }
  }
  /// <remarks/>
  public string Instance {
    get {
      return this.instanceField;
    }
    set {
      this.instanceField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class CONDITION {
  private string[] instanceField;
  private FUNCTIONCALL expressionField;
  private TEMPERATURE temperatureField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Instance")]
  public string[] Instance {
    get {
      return this.instanceField;
    }
    set {
      this.instanceField = value;
    }
  }
  /// <remarks/>
  public FUNCTIONCALL Expression {
    get {
      return this.expressionField;
```

```
    }
    set {
      this.expressionField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public TEMPERATURE Temperature {
    get {
      return this.temperatureField;
    }
    set {
      this.temperatureField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
public enum TEMPERATURE {

  /// <remarks/>
  cold,

  /// <remarks/>
  hot,
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class MESSAGE {
  private MESSAGETYPE typeField;
  private CONTENT contentField;
  private string fromField;
  private string toField;
  private TEMPERATURE temperatureField;
  /// <remarks/>
  public MESSAGETYPE Type {
    get {
      return this.typeField;
    }
    set {
      this.typeField = value;
    }
  }

  /// <remarks/>
  public CONTENT Content {
    get {
      return this.contentField;
    }
    set {
      this.contentField = value;
    }
  }
  /// <remarks/>
  public string From {
    get {
      return this.fromField;
    }
    set {
      this.fromField = value;
    }
  }
  /// <remarks/>
  public string To {
    get {
      return this.toField;
    }
    set {
      this.toField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
```

```
  public TEMPERATURE Temperature {
    get {
      return this.temperatureField;
    }
    set {
      this.temperatureField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class MESSAGETYPE {
  private string typeField;
  /// <remarks/>
  public string Type {
    get {
      return this.typeField;
    }
    set {
      this.typeField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class CONTENT {
  private object itemField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Function", typeof(FUNCTIONCALL))]
  [System.Xml.Serialization.XmlElementAttribute("Message", typeof(string))]
  public object Item {
    get {
      return this.itemField;
    }
    set {
      this.itemField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class SETTIMER {
  private string durationField;
  private string instanceField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute(DataType="integer")]
  public string Duration {
    get {
      return this.durationField;
    }
    set {
      this.durationField = value;
    }
  }
  /// <remarks/>
  public string Instance {
    get {
      return this.instanceField;
    }
    set {
      this.instanceField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
```

```
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class SUBCHART {
  private object[] itemsField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Event", typeof(EVENT))]
  [System.Xml.Serialization.XmlElementAttribute("If-Then-Else", typeof(IFTHENELSE))]
  [System.Xml.Serialization.XmlElementAttribute("Loop", typeof(LOOP))]
  public object[] Items {
    get {
      return this.itemsField;
    }
    set {
      this.itemsField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class EVENT {
  private object itemField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Coregion", typeof(COREGION))]
  [System.Xml.Serialization.XmlElementAttribute("EventDef", typeof(EVENTDEF))]
  public object Item {
    get {
      return this.itemField;
    }
    set {
      this.itemField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class IFTHENELSE {
  private CONDITION conditionField;
  private SUBCHART ifThenField;
  private SUBCHART elseField;
  /// <remarks/>
  public CONDITION Condition {
    get {
      return this.conditionField;
    }
    set {
      this.conditionField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("If-Then")]
  public SUBCHART IfThen {
    get {
      return this.ifThenField;
    }
    set {
      this.ifThenField = value;
    }
  }
  /// <remarks/>
  public SUBCHART Else {
    get {
      return this.elseField;
    }
    set {
      this.elseField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
```

```
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class LOOP {
  private CONDITION conditionField;
  private SUBCHART subchartField;
  /// <remarks/>
  public CONDITION Condition {
    get {
      return this.conditionField;
    }
    set {
      this.conditionField = value;
    }
  }
  /// <remarks/>
  public SUBCHART Subchart {
    get {
      return this.subchartField;
    }
    set {
      this.subchartField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(IncludeInSchema=false)]
public enum ItemChoiceType {
  /// <remarks/>
  Action,
  /// <remarks/>
  Condition,
  /// <remarks/>
  Message,
  /// <remarks/>
  ResetTimer,
  /// <remarks/>
  SetTimer,
  /// <remarks/>
  Subchart,
  /// <remarks/>
  TimeOut,
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class CHART {
  private EVENT[] forbiddenEventField;
  private EVENT[] prechartField;
  private EVENT[] eventField;
  private CHARTTYPE typeField;
  private string lscNameField;
  private ACTMODE activationModeField;
  private bool activationModeFieldSpecified;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("ForbiddenEvent")]
  public EVENT[] ForbiddenEvent {
    get {
      return this.forbiddenEventField;
    }
    set {
      this.forbiddenEventField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlArrayItemAttribute("Event", IsNullable=false)]
  public EVENT[] Prechart {
    get {
      return this.prechartField;
    }
    set {
      this.prechartField = value;
    }
  }
  /// <remarks/>
```

```
  [System.Xml.Serialization.XmlElementAttribute("Event")]
  public EVENT[] Event {
    get {
      return this.eventField;
    }
    set {
      this.eventField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public CHARTTYPE Type {
    get {
      return this.typeField;
    }
    set {
      this.typeField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string LscName {
    get {
      return this.lscNameField;
    }
    set {
      this.lscNameField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public ACTMODE ActivationMode {
    get {
      return this.activationModeField;
    }
    set {
      this.activationModeField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlIgnoreAttribute()]
  public bool ActivationModeSpecified {
    get {
      return this.activationModeFieldSpecified;
    }
    set {
      this.activationModeFieldSpecified = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
public enum CHARTTYPE {
  /// <remarks/>
  existential,
  /// <remarks/>
  universal,
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
public enum ACTMODE {
  /// <remarks/>
  initial,
  /// <remarks/>
  invariant,
  /// <remarks/>
  iterative,
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class INSTDEF {
  private string classField;
```

```
  private string nameField;
  /// <remarks/>
  public string Class {
    get {
      return this.classField;
    }
    set {
      this.classField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class INSTDEFS {
  private INSTDEF[] instanceField;
  private string nameField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Instance")]
  public INSTDEF[] Instance {
    get {
      return this.instanceField;
    }
    set {
      this.instanceField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class MESSAGETYPEDEF {
  private string nameField;
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class MESSAGETYPEDEFS {
  private MESSAGETYPEDEF[] messageTypeField;
  private string nameField;
  /// <remarks/>
```

```
  [System.Xml.Serialization.XmlElementAttribute("MessageType")]
  public MESSAGETYPEDEF[] MessageType {
    get {
      return this.messageTypeField;
    }
    set {
      this.messageTypeField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class VARIDEF {
  private string initialValueField;
  private string classField;
  private string variNameField;
  private VARITYPE typeField;
  /// <remarks/>
  public string InitialValue {
    get {
      return this.initialValueField;
    }
    set {
      this.initialValueField = value;
    }
  }
  /// <remarks/>
  public string Class {
    get {
      return this.classField;
    }
    set {
      this.classField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string VariName {
    get {
      return this.variNameField;
    }
    set {
      this.variNameField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public VARITYPE Type {
    get {
      return this.typeField;
    }
    set {
      this.typeField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
public enum VARITYPE {
  /// <remarks/>
  @void,
  /// <remarks/>
  @int,
```

```
  /// <remarks/>
  @bool,
  /// <remarks/>
  @string,
  /// <remarks/>
  @double,
  /// <remarks/>
  Information,
}

/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class FORMALPARAM {
  private string nameField;
  private VARITYPE typeField;
  /// <remarks/>
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
  /// <remarks/>
  public VARITYPE Type {
    get {
      return this.typeField;
    }
    set {
      this.typeField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class FUNCTIONDEF {
  private string classField;
  private VARITYPE returnTypeField;
  private bool returnTypeFieldSpecified;
  private FORMALPARAM[] formalParameterField;
  private CONDITION[] preconditionField;
  private CONDITION[] postconditionField;
  private string functionNameField;
  /// <remarks/>
  public string Class {
    get {
      return this.classField;
    }
    set {
      this.classField = value;
    }
  }
  /// <remarks/>
  public VARITYPE ReturnType {
    get {
      return this.returnTypeField;
    }
    set {
      this.returnTypeField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlIgnoreAttribute()]
  public bool ReturnTypeSpecified {
    get {
      return this.returnTypeFieldSpecified;
    }
    set {
      this.returnTypeFieldSpecified = value;
    }
```

```
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("FormalParameter")]
  public FORMALPARAM[] FormalParameter {
    get {
      return this.formalParameterField;
    }
    set {
      this.formalParameterField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Precondition")]
  public CONDITION[] Precondition {
    get {
      return this.preconditionField;
    }
    set {
      this.preconditionField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Postcondition")]
  public CONDITION[] Postcondition {
    get {
      return this.postconditionField;
    }
    set {
      this.postconditionField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string FunctionName {
    get {
      return this.functionNameField;
    }
    set {
      this.functionNameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class INVARIANT {
  private CONDITION conditionField;
  /// <remarks/>
  public CONDITION Condition {
    get {
      return this.conditionField;
    }
    set {
      this.conditionField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class CLASSDEF {
  private INVARIANT[] invariantField;
  private FUNCTIONDEF[] functionField;
  private VARIDEF[] variableField;
  private string nameField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Invariant")]
  public INVARIANT[] Invariant {
    get {
      return this.invariantField;
    }
    set {
      this.invariantField = value;
```

```
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Function")]
  public FUNCTIONDEF[] Function {
    get {
      return this.functionField;
    }
    set {
      this.functionField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Variable")]
  public VARIDEF[] Variable {
    get {
      return this.variableField;
    }
    set {
      this.variableField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class CLASSDEFS {

  private CLASSDEF[] classField;

  private string nameField;

  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Class")]
  public CLASSDEF[] Class {
    get {
      return this.classField;
    }
    set {
      this.classField = value;
    }
  }
  /// <remarks/>
  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string Name {
    get {
      return this.nameField;
    }
    set {
      this.nameField = value;
    }
  }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
public partial class ENUMERATION {
  private string[] elementField;
  /// <remarks/>
  [System.Xml.Serialization.XmlElementAttribute("Element")]
  public string[] Element {
    get {
      return this.elementField;
```

```
      }
      set {
        this.elementField = value;
      }
    }
  }
}
```

CHART.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
using System.Collections;
namespace LscClasses
{
  partial class CHART
  {
    internal static Hashtable Concatenate2(Hashtable events, Hashtable ev)
    {
      if (ev == null || ev.Keys == null)
        return events;
      if (events == null)
        return ev;
      foreach (CLASSDEF cd in ev.Keys)
      {
        if (events.Contains(cd))
        {
          Hashtable e1 = (Hashtable)events[cd];
          Hashtable e2 = (Hashtable)ev[cd];
          e1 =  Concatenate2(e1, e2);
        }
        else
         events.Add(cd, ev[cd]);
      }
      return events;
    }
    internal static Hashtable Concatenate(Hashtable events, Hashtable ev)
    {
      if (ev == null || ev.Keys == null)
        return events;
      if (events == null)
        return ev;
      foreach (CodeConditionStatement cd in ev.Keys)
      {
        if (events.Contains(cd))
        {
          if ((CodeStatementCollection)events[cd] == null)
            events[cd] = new CodeStatementCollection();
          if ((CodeStatementCollection)ev[cd] != null)
            ((CodeStatementCollection)events[cd]).
                  AddRange((CodeStatementCollection)ev[cd]);
        }
        else
        {
          if ((CodeStatementCollection)ev[cd] != null)
            events.Add(cd, ev[cd]);
        }
      }
      return events;
    }
    internal CodeConditionStatement classActiveState;
    internal int classActiveStateNumber;
    internal CodeConditionStatement classCondition;
    public void ToCode(LscSpec spec, CLASSDEF cls)
    {
       CodeConditionStatement newScenario = new CodeConditionStatement();
       newScenario.Condition = new CodeSnippetExpression("\"" + this.LscName
            + "\".Equals(location.Scenario)");
       cls.GoToNextStateMethod.Statements.Add(newScenario);
       classCondition = newScenario;
       classActiveState = null;
       classActiveStateNumber = 0;
       if (this.Prechart != null)
        foreach (EVENT e in this.Prechart)
        {
          e.prechart = true;
```

```
          Hashtable events = e.ToCode(spec, this, cls );
          if (events != null && events.Keys != null)
            foreach(CodeConditionStatement ccs in events.Keys)
              if ((CodeStatementCollection)events[ccs] != null)
                  ccs.TrueStatements.AddRange((CodeStatementCollection)events[ccs]);
        }
      if (this.Event != null)
        foreach (EVENT e in this.Event)
        {
          Hashtable events = e.ToCode(spec, this, cls );
          if (events != null && events.Keys != null)
            foreach(CodeConditionStatement ccs in events.Keys)
              if ((CodeStatementCollection)events[ccs] != null)
                  ccs.TrueStatements.AddRange((CodeStatementCollection)events[ccs]);



        }
      if (classActiveStateNumber != 0)
      {
        cls.charts.Add(this.LscName, null);
      }
    }
  }
}
```

CLASSDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.CodeDom;
using Microsoft.CSharp;
using System.CodeDom.Compiler;
using System.Reflection;
using System.Collections;
namespace LscClasses
{
  partial class CLASSDEF
  {
    CodeMemberMethod initEventHandlingMethod;
    CodeMemberMethod populateAvailableScenariosMethod;
    CodeMemberMethod goToNextStateMethod;
    internal Hashtable messages = new Hashtable();
    internal Hashtable charts = new Hashtable();
    StreamWriter sw;
    ICodeGenerator codeGenerator;
    CodeTypeDeclaration clsDecl;
    CodeGeneratorOptions cgo;
    CodeNamespace cnsCodeNamespace;
    Stream codeFile;
    internal CodeMemberMethod InitEventHandlingMethod
    {
      get
      {
        return initEventHandlingMethod;
      }
      set
      {
        initEventHandlingMethod = value;
      }
    }
    internal CodeMemberMethod PopulateAvailableScenariosMethod
    {
      get
      {
        return populateAvailableScenariosMethod;
      }
      set
      {
        populateAvailableScenariosMethod = value;
      }
    }
    internal CodeMemberMethod GoToNextStateMethod
    {
      get
      {
```

```
    return goToNextStateMethod;
  }
  set
  {
    goToNextStateMethod = value;
  }
}
public override string ToString()
{
  return Name;
}
public override bool Equals(object obj)
{
  if (obj.GetType() == typeof(CLASSDEF))
    return Name.Equals(((CLASSDEF)obj).Name);
  return base.Equals(obj);
}
public override int GetHashCode()
{
  return base.GetHashCode();
}

public void ToCodeStart(string dir)
{
  codeFile = File.Open(Path.Combine(dir, this.Name+".cs"), FileMode.Create);
  sw = new StreamWriter(codeFile);
  CSharpCodeProvider cscp = new CSharpCodeProvider();
  codeGenerator = cscp.CreateGenerator(sw);
  cgo = new CodeGeneratorOptions();
  cgo.BracingStyle = "C";
  cgo.IndentString = "  ";
  CodeSnippetCompileUnit cscu = new CodeSnippetCompileUnit("using System;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cscu = new CodeSnippetCompileUnit("using FixedComponents;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cscu = new CodeSnippetCompileUnit("using System.Collections;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cscu = new CodeSnippetCompileUnit("using System.IO;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cscu = new CodeSnippetCompileUnit("using System.Threading;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cscu = new CodeSnippetCompileUnit("using System.Diagnostics;");
  codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
  cnsCodeNamespace = new CodeNamespace("GeneratedComponents");
  clsDecl = new CodeTypeDeclaration();
  clsDecl.Name = this.Name;
  clsDecl.IsClass = true;
  clsDecl.TypeAttributes = TypeAttributes.Public;
  clsDecl.BaseTypes.Add(new CodeTypeReference("MessageHandler"));
  cnsCodeNamespace.Types.Add(clsDecl);
  CodeConstructor clsConstructor = new CodeConstructor();
  clsConstructor.Attributes = MemberAttributes.Public;
  clsConstructor.Parameters.Add(
    new CodeParameterDeclarationExpression("MessageQueue", "queue"));
  clsConstructor.Parameters.Add(
    new CodeParameterDeclarationExpression(typeof(string), "name"));
  clsConstructor.BaseConstructorArgs.Add(
    new CodeVariableReferenceExpression("queue"));
  clsConstructor.Statements.Add(new CodeSnippetExpression("this.name = name"));
  clsConstructor.Statements.Add(new CodeSnippetExpression("thread.Name = name"));
  clsConstructor.Statements.Add(
    new CodeSnippetExpression("logger = new Logger(name, Logger.ALL)"));
  clsDecl.Members.Add(clsConstructor);
  if (this.Variable != null)
    foreach (VARIDEF vd in Variable)
    {
      vd.ToCode(clsDecl);
    }
  if (this.Function != null)
    foreach (FUNCTIONDEF fd in Function)
    {
      fd.ToCode(clsDecl);
    }
  if (this.Invariant != null)
    foreach (FUNCTIONCALL fc in Invariant)
      clsDecl.Comments.Add(
        new CodeCommentStatement("^invariant "+ fc.ToString()+";"));
```

```
      initEventHandlingMethod = new CodeMemberMethod();
      initEventHandlingMethod.Attributes =
         MemberAttributes.Public | MemberAttributes.Override;
      initEventHandlingMethod.Name = "InitEventHandling";
      initEventHandlingMethod.ReturnType = new CodeTypeReference(typeof(void));
      clsDecl.Members.Add(initEventHandlingMethod);
      populateAvailableScenariosMethod = new CodeMemberMethod();
      populateAvailableScenariosMethod.Attributes =
        MemberAttributes.Public|MemberAttributes.Final;
      populateAvailableScenariosMethod.Name = "PopulateAvailableScenarios";
      populateAvailableScenariosMethod.ReturnType =
        new CodeTypeReference(typeof(void));
      clsDecl.Members.Add(populateAvailableScenariosMethod);
      goToNextStateMethod = new CodeMemberMethod();
      goToNextStateMethod.Attributes =
        MemberAttributes.Public|MemberAttributes.Override;
      goToNextStateMethod.Name = "GoToNextState";
      goToNextStateMethod.ReturnType = new CodeTypeReference(typeof(void));
      goToNextStateMethod.Parameters.Add(
         new CodeParameterDeclarationExpression("Location", "location"));
      goToNextStateMethod.Statements.Add(
         new CodeSnippetStatement("//^assume location != null;"));
      clsDecl.Members.Add(goToNextStateMethod);
      populateAvailableScenariosMethod.Statements.Add(
        new CodeSnippetExpression("ArrayList availableLocations"));
      initEventHandlingMethod.Statements.Add(
        new CodeSnippetExpression("ArrayList types = new ArrayList()"));
    }
    public void ToCodeEnd()
    {
      if (messages != null && messages.Keys != null)
        foreach (string s in messages.Keys)
        {
          InitEventHandlingMethod.Statements.Add(
           new CodeSnippetExpression("types.Add(
            Type.GetType(\"GeneratedComponents." + s + "\"))"));
          CodeStatementCollection c = (CodeStatementCollection)messages[s];
          c.Insert(0,new CodeExpressionStatement(
            new CodeSnippetExpression(" availableLocations = new ArrayList()")));
          c.Add(new CodeExpressionStatement(new CodeSnippetExpression(
            "msgTypesToAvailableLocations.Add(Type.GetType(\"GeneratedComponents."
            +s + "\"), availableLocations)")));
          PopulateAvailableScenariosMethod.Statements.AddRange(c);
        }
      if (charts != null && charts.Keys != null)
        foreach (string s in charts.Keys)
          InitEventHandlingMethod.Statements.Add(
            new CodeSnippetExpression("scenarioToInternalQueue.Add(\"" + s + "\",
             new BlockingQueue<Location>())"));
      GoToNextStateMethod.Statements.Add(
        new CodeSnippetExpression("RemoveScenario(location)"));
      initEventHandlingMethod.Statements.Add(
        new CodeSnippetExpression("queue.RegisterEventHandler(this, types)"));
      initEventHandlingMethod.Statements.Add(
        new CodeSnippetExpression("PopulateAvailableScenarios()"));
      codeGenerator.GenerateCodeFromNamespace(cnsCodeNamespace, sw, cgo);
      sw.Close();
      codeFile.Close();
    }
  }
}
```

CLASSDEFS.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using LscClasses;
namespace LscClasses
{
  partial class CLASSDEFS
  {
    public CLASSDEF GetClass(string name)
    {
      if (Class != null)
        foreach (CLASSDEF cd in Class)
        {
          if (name.Equals(cd.Name))
```

```
            return cd;
          }
        return null;
      }
      public void ToCodeStart(string dir)
      {
        if (Class != null)
          foreach (CLASSDEF c in Class)
          {
            c.ToCodeStart(dir);
          }
      }
      public void ToCodeEnd()
      {
        if (Class != null)
          foreach (CLASSDEF c in Class)
          {
            c.ToCodeEnd();
          }
      }
      public void ToCode(LscSpec spec, CHART c)
      {
        if (Class != null)
          foreach (CLASSDEF cd in Class)
          {
            c.ToCode(spec, cd);
          }
      }
    }
}
```

CONDITION.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
using System.Collections;

namespace LscClasses
{
  partial class CONDITION
  {
    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls)
    {
      Hashtable events = new Hashtable();
      if (this.Instance != null)
        foreach (string s in this.Instance)
        {
          INSTDEF id = spec.Instances.GetInstance(s);
          CLASSDEF cd = spec.Classes.GetClass(id.Class);
          if (cd.Equals(cls))
          {
            CodeConditionStatement cond;
            cond = new CodeConditionStatement(
                new CodeBinaryOperatorExpression(
                  this.Expression.ToCode(),
                  CodeBinaryOperatorType.IdentityEquality,
                  new CodePrimitiveExpression(false)),
                new CodeStatement[] {
                    new CodeExpressionStatement(
                      new CodeMethodInvokeExpression(
                        new CodeMethodReferenceExpression(
                          new CodeThisReferenceExpression(),
                          "RemoveScenario"),
                        new CodeExpression[] {
                          new CodeVariableReferenceExpression("location")}))});
            if (this.Temperature == TEMPERATURE.cold)
            {
              cond.TrueStatements.Add(new CodeMethodReturnStatement());
            }
            else
            {
              cond.TrueStatements.Add(
                new CodeExpressionStatement(
                  new CodeSnippetExpression(
                     "Debug.Assert( " + this.Expression.ToString() + ")")));
```

```
            }
            if (events.Contains(chart.classActiveState))
              ((CodeStatementCollection)events[chart.classActiveState]).Add(cond);
            else
              events.Add(chart.classActiveState,
                new CodeStatementCollection(
                  new CodeStatement[]{
                    cond }));
          }
        }
      return events;
    }
  }
}
```

EVENT.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.CodeDom;
namespace LscClasses
{
  partial class EVENT
  {
    internal bool prechart = false;
    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls)
    {
      EVENTDEF ed = Item as EVENTDEF;
      if (ed == null)
      {
        Hashtable events = new Hashtable();
        EVENTDEF[] coreg = Item as EVENTDEF[];
        foreach (EVENTDEF evd in coreg)
        {
          Hashtable ev = evd.ToCode(spec, chart, cls, prechart);
          events = CHART.Concatenate(events, ev);
          if (evd.ItemElementName == ItemChoiceType.Message)
            return events;
        }
        return events;
      }
      return ed.ToCode(spec, chart, cls, prechart);

    }
  }
}
```

EVENTDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.CodeDom;
namespace LscClasses
{
  partial class EVENTDEF
  {

    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls, bool prechart)
    {
      Hashtable events = new Hashtable();
      if (ItemElementName == ItemChoiceType.Message)
      {
        MESSAGE m = Item as MESSAGE;
        return m.ToCode(spec, chart, cls, prechart);
      }
      if (ItemElementName == ItemChoiceType.Action)
      {
        FUNCTIONCALL fc = Item as FUNCTIONCALL;
        INSTDEF id = spec.Instances.GetInstance(fc.Instance);
        CLASSDEF cd = spec.Classes.GetClass(id.Class);
        if (cd.Equals(cls))
        {

          events.Add(chart.classActiveState,
            new CodeStatementCollection(new CodeStatement[] {
```

```
            new CodeExpressionStatement(fc.ToCode()) }));
        }
        return events;
      }
      if (ItemElementName == ItemChoiceType.Condition)
      {
        CONDITION c = Item as CONDITION;
        events = c.ToCode(spec, chart, cls );
        return events;
      }
      if (ItemElementName == ItemChoiceType.SetTimer)
      {
        SETTIMER st = Item as SETTIMER;
        INSTDEF id = spec.Instances.GetInstance(st.Instance);
        CLASSDEF cd = spec.Classes.GetClass(id.Class);
        if (cd.Equals(cls))
        {
          string s = "Thread.Sleep(" + st.Duration + "000)";
          events.Add(chart.classActiveState,
            new CodeStatementCollection(
              new CodeStatement[]{
                new CodeExpressionStatement(new CodeSnippetExpression(s))}));
        }
        return events;
      }
      if (ItemElementName == ItemChoiceType.Subchart)
      {
        SUBCHART sc = Item as SUBCHART;

        return sc.ToCode(spec, chart, cls);
      }
      return null;

    }
  }
}
```

FORMALPARAMETER.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
namespace LscClasses
{
  partial class FORMALPARAM
  {
    public override string ToString()
    {
      return this.Type + " " + this.Name;
    }
    public void ToCode(CodeMemberMethod method)
    {

      method.Parameters.Add(
        new CodeParameterDeclarationExpression(this.Type.ToString(), this.Name));
    }
  }
}
```

FUNCTIONCALL.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using LscClasses;
using System.CodeDom;
namespace LscClasses
{
  partial class FUNCTIONCALL
  {
    public override string ToString()
    {

      string expr = this.FunctionName + "(";
      if (this.EffectiveParameter != null)
```

```
      {
        int i = 0;
        foreach (EFFECTIVEPARAM ep in this.EffectiveParameter)
        {
          if (i > 0)
            expr += ", ";
          i++;
          VARIABLE var = ep.Item as VARIABLE;
          if (var == null)
            expr += ep.Item as string;
          else
          {
            expr += var.Instance + "." + var.VariableName;
          }
        }
      }
      expr += ")";
      return expr;
    }
    public CodeMethodInvokeExpression ToCode()
    {
      CodeMethodInvokeExpression m = new CodeMethodInvokeExpression();
      m.Method = new CodeMethodReferenceExpression(
              new CodeThisReferenceExpression(),
              this.FunctionName);
      if (this.EffectiveParameter != null)
      {
        foreach (EFFECTIVEPARAM ep in this.EffectiveParameter)
        {
          VARIABLE var = ep.Item as VARIABLE;
          if (var == null)
          {
            m.Parameters.Add(new CodeSnippetExpression(ep.Item as string));
          }
          else
          {
            m.Parameters.Add(new CodeFieldReferenceExpression(
              new CodeSnippetExpression(var.Instance), var.VariableName));
          }
        }
      }
      return m;
    }
  }
}
```

FUNCTIONDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
namespace LscClasses
{
  partial class FUNCTIONDEF
  {
    public override string ToString()
    {
      return FunctionName;
    }
    public override bool Equals(object obj)
    {
      if (obj.GetType() == typeof(FUNCTIONDEF))
        return this.FunctionName.Equals(((FUNCTIONDEF)obj).FunctionName) &&
          this.Class.Equals(((FUNCTIONDEF)obj).Class);
      return base.Equals(obj);
    }
    public override int GetHashCode()
    {
      return base.GetHashCode();
    }
    public void ToCode(CodeTypeDeclaration cls)
    {
      CodeMemberMethod method = new CodeMemberMethod();
      method.Attributes = MemberAttributes.Public | MemberAttributes.Final;
      method.Name = this.FunctionName;
      method.ReturnType = new CodeTypeReference(this.ReturnType.ToString());
```

```
      if (this.FormalParameter != null)
        foreach (FORMALPARAM fp in this.FormalParameter)
          fp.ToCode(method);
      if (this.Precondition != null)
        foreach (FUNCTIONCALL c in this.Precondition)
         method.Statements.Add(
             new CodeSnippetExpression("//^requires " + c.ToString()));
      if (this.Postcondition != null)
        foreach (FUNCTIONCALL c in this.Postcondition)
            method.Statements.Add(
               new CodeSnippetExpression("//^requires " + c.ToString()));
      if (this.ReturnType != VARITYPE.@void)
        method.Statements.Add(new CodeSnippetExpression("return null"));
      cls.Members.Add(method);
    }
  }
}
```

IFTHENELSE.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.CodeDom;
namespace LscClasses
{
  partial class IFTHENELSE
  {
    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls)
    {
      INSTDEF id = spec.Instances.GetInstance(Condition.Instance[0]);
      CLASSDEF clsCond = spec.Classes.GetClass(id.Class);
      Hashtable ifHash;
      Hashtable events = new Hashtable();
      CodeConditionStatement state = null;
      CodeConditionStatement ccs = null;
      state = chart.classActiveState;
      if (cls.Equals(clsCond))
      {
        CodeExpression condExpr = Condition.Expression.ToCode();
        ccs = new CodeConditionStatement();
        ccs.Condition = condExpr;
        events.Add(state, new CodeStatementCollection(new CodeStatement[] { ccs }));
        ifHash = IfThen.ToCode(spec, chart, cls);
        if (ifHash != null && ifHash.Contains(state))
        {
           ccs.TrueStatements.AddRange((CodeStatementCollection)(
             (Hashtable)ifHash)[state]);
        }
        if (events == null || events.Keys == null)
          events = ifHash;
        if (ifHash != null && ifHash.Keys != null)
          foreach (CodeConditionStatement cd in ifHash.Keys)
          {
            if (!cd.Equals(state))
            {
              if (events.Contains(cd))
              {
                if ((CodeStatementCollection)events[cd] == null)
                  events[cd] = new CodeStatementCollection();
                if ((CodeStatementCollection)ifHash[cd] != null)
                  ((CodeStatementCollection)events[cd]).
                      AddRange((CodeStatementCollection)ifHash[cd]);
              }
              else
              {
                if ((CodeStatementCollection)ifHash[cd] != null)
                  events.Add(cd, ifHash[cd]);
              }
            }
          }
      }
      ifHash = IfThen.ToCodeEnd(spec, chart, cls);
      if (ifHash != null && ifHash.Keys != null)
        events = CHART.Concatenate(events, ifHash);
```

```
      if (Else == null)
        return events;
      chart.classActiveState = state;
      if (cls.Equals(clsCond))
      {
        ifHash = Else.ToCode(spec, chart, cls);
        if (ifHash != null && ifHash.Contains(state))
        {
          ccs.FalseStatements.
            AddRange((CodeStatementCollection)((Hashtable)ifHash)[state]);
        }
        if (events == null || events.Keys == null)
          events = ifHash;
        if (ifHash != null && ifHash.Keys != null)
          foreach (CodeConditionStatement cd in ifHash.Keys)
          {
            if (!cd.Equals(state))
            {
              if (events.Contains(cd))
              {
                if ((CodeStatementCollection)events[cd] == null)
                  events[cd] = new CodeStatementCollection();
                if ((CodeStatementCollection)ifHash[cd] != null)
                  ((CodeStatementCollection)events[cd]).
                    AddRange((CodeStatementCollection)ifHash[cd]);
              }
              else
              {
                if ((CodeStatementCollection)ifHash[cd] != null)
                  events.Add(cd, ifHash[cd]);
              }
            }
          }
      }
      ifHash = Else.ToCodeEnd(spec, chart, cls);
      if (ifHash != null && ifHash.Keys != null)
        events = CHART.Concatenate(events, ifHash);
      return events;
    }
  }
}
```

INSTDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
namespace LscClasses
{
  partial class INSTDEF
  {
    public override string ToString()
    {
      return Name;
    }
    public override bool Equals(object obj)
    {
      if (obj.GetType() == typeof(INSTDEF))
        return Name.Equals(((INSTDEF)obj).Name);
      return base.Equals(obj);
    }
    public override int GetHashCode()
    {
      return base.GetHashCode();
    }
    public CodeStatement ToCode()
    {
      return new CodeExpressionStatement(new CodeSnippetExpression(
        Class +" "+ Name + " = new "+Class+"(queue, \""+Name+"\")"));
    }
  }
}
```

INSTDEFS.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.CodeDom.Compiler;
using System.CodeDom;
using Microsoft.CSharp;
using System.Reflection;
namespace LscClasses
{
  partial class INSTDEFS
  {
    public INSTDEF GetInstance(string name)
    {
      if (this.Instance != null)
        foreach (INSTDEF id in Instance)
        {
          if (name.Equals(id.Name))
            return id;
        }
      return null;
    }
    public void ToCode(string dir)
    {
      StreamWriter sw;
      ICodeGenerator codeGenerator;
      CodeTypeDeclaration clsDecl;
      CodeGeneratorOptions cgo;
      CodeNamespace cnsCodeNamespace;
      Stream codeFile;
      codeFile = File.Open(Path.Combine(dir, "Program.cs"), FileMode.Create);
      sw = new StreamWriter(codeFile);
      CSharpCodeProvider cscp = new CSharpCodeProvider();
      codeGenerator = cscp.CreateGenerator(sw);
      cgo = new CodeGeneratorOptions();
      cgo.BracingStyle = "C";
      cgo.IndentString = "  ";
      CodeSnippetCompileUnit cscu = new CodeSnippetCompileUnit("using System;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cscu = new CodeSnippetCompileUnit("using FixedComponents;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cscu = new CodeSnippetCompileUnit("using System.Collections;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cscu = new CodeSnippetCompileUnit("using System.IO;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cscu = new CodeSnippetCompileUnit("using System.Threading;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cnsCodeNamespace = new CodeNamespace("GeneratedComponents");
      clsDecl = new CodeTypeDeclaration();
      clsDecl.Name = "Program";
      clsDecl.IsClass = true;
      clsDecl.TypeAttributes = TypeAttributes.Public;
      cnsCodeNamespace.Types.Add(clsDecl);
      CodeMemberMethod m = new CodeMemberMethod();
      m.Name = "Main";
      m.Attributes = MemberAttributes.Static | MemberAttributes.Public;
      m.Parameters.Add(new CodeParameterDeclarationExpression("string []", "args"));
      m.Statements.Add (
        new CodeSnippetExpression("MessageQueue queue = new MessageQueue()"));
      if (this.Instance != null)
        foreach (INSTDEF id in this.Instance)
          m.Statements.Add(id.ToCode());
      clsDecl.Members.Add(m);
      codeGenerator.GenerateCodeFromNamespace(cnsCodeNamespace, sw, cgo);
      sw.Close();
      codeFile.Close();
    }
  }
}
```

MESSAGE.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
```

```
using System.CodeDom;
namespace LscClasses
{
  partial class MESSAGE
  {

    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls, bool prechart)
    {
      Hashtable events = new Hashtable();
      INSTDEF idTo = spec.Instances.GetInstance(this.To);
      CLASSDEF cdTo = spec.Classes.GetClass(idTo.Class);
      INSTDEF idFrom = spec.Instances.GetInstance(this.From);
      CLASSDEF cdFrom = spec.Classes.GetClass(idFrom.Class);
      if (cdFrom.Equals(cls))
      {
        if (chart.classActiveState != null && !prechart)
        {
          events.Add(chart.classActiveState,
            new CodeStatementCollection(
              new CodeStatement[]{
                new CodeExpressionStatement(
                  new CodeSnippetExpression(
                    "queue.EnqueueMessage(new " +
                    this.Type.Type +  "(\"" +
                    idTo.Name +
                    "\".GetHashCode(), Type.GetType(\"GeneratedComponents."+
                    cdTo.Name +"\"), location.Scenario))"))}));
        }
      }
      if (cls.Equals(cdTo))
      {
        int state = ((int)chart.classActiveStateNumber);
        CodeConditionStatement newState = new CodeConditionStatement();
        newState.Condition = new CodeSnippetExpression("location.State == " + state);
        if (!cls.messages.Contains(this.Type.Type))
        {
          cls.messages.Add(this.Type.Type,  new CodeStatementCollection());
        }
        bool firstInChart = (chart.classActiveState == null);
        ((CodeStatementCollection)cls.messages[this.Type.Type]).Add(
          new CodeExpressionStatement(
            new CodeSnippetExpression(
              "availableLocations.Add(new Location(\""+
              chart.LscName+"\", "+
              chart.classActiveStateNumber+ ", "+
              prechart.ToString().ToLower() + ", " +
              firstInChart.ToString().ToLower() +"))")));
        if (chart.classActiveState != null)
        {
          events.Add(chart.classActiveState,
            new CodeStatementCollection(
              new CodeStatement[]{
                new CodeExpressionStatement(
                  new CodeSnippetExpression("SetStateScenario(location, " +
                   ((int)chart.classActiveStateNumber ) + ")"))}));
        }
        chart.classActiveState = newState;
        chart.classActiveStateNumber = state + 1;
        chart.classCondition.TrueStatements.Add(newState);
      }
      return events;
    }
  }
}
```

MESSAGETYPEDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.CodeDom;
using Microsoft.CSharp;
using System.CodeDom.Compiler;
using System.Reflection;
namespace LscClasses
```

```
{
  partial class MESSAGETYPEDEF
  {
    public override string ToString()
    {
      return Name;
    }
    public void ToCode(string dir)
    {
      Stream codeFile = File.Open(Path.Combine(dir, this.Name+".cs"),
FileMode.Create);
      StreamWriter sw = new StreamWriter(codeFile);
      CSharpCodeProvider cscp = new CSharpCodeProvider();
      ICodeGenerator codeGenerator = cscp.CreateGenerator(sw);
      CodeGeneratorOptions cgo = new CodeGeneratorOptions();
      cgo.BracingStyle = "C";
      CodeSnippetCompileUnit cscu = new CodeSnippetCompileUnit("using System;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      cscu = new CodeSnippetCompileUnit("using FixedComponents;");
      codeGenerator.GenerateCodeFromCompileUnit(cscu, sw, cgo);
      CodeNamespace cnsCodeNamespace = new CodeNamespace("GeneratedComponents");
      CodeTypeDeclaration clsDecl = new CodeTypeDeclaration();
      clsDecl.Name = this.Name;
      clsDecl.IsClass = true;
      clsDecl.TypeAttributes = TypeAttributes.Public;
      clsDecl.BaseTypes.Add(new CodeTypeReference("Message"));
      cnsCodeNamespace.Types.Add(clsDecl);
      CodeConstructor clsConstructor = new CodeConstructor();
      clsConstructor.Attributes = MemberAttributes.Public;
      clsConstructor.Parameters.Add(
         new CodeParameterDeclarationExpression(typeof(int), "recipient"));
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(Type), "recipientType"));
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(string), "scenario"));
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression("Information", "info") );
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("recipient"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("recipientType"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("scenario"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("info"));
      clsDecl.Members.Add(clsConstructor);
      clsConstructor = new CodeConstructor();
      clsConstructor.Attributes = MemberAttributes.Public;
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(int), "recipient"));
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(Type), "recipientType"));
      clsConstructor.Parameters.Add(
        new CodeParameterDeclarationExpression(typeof(string), "scenario"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("recipient"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("recipientType"));
      clsConstructor.BaseConstructorArgs.Add(
        new CodeVariableReferenceExpression("scenario"));
      clsDecl.Members.Add(clsConstructor);
      codeGenerator.GenerateCodeFromNamespace(cnsCodeNamespace,sw,cgo);
      sw.Close();
      codeFile.Close();
    }
  }
}
```

MESSAGETYPEDEFS.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace LscClasses
{
  partial class MESSAGETYPEDEFS
```

```
  {
    public MESSAGETYPEDEF GetMessageType(string name)
    {
      if (MessageType != null)
        foreach (MESSAGETYPEDEF mtd in this.MessageType)
        {
          if (name.Equals(mtd.Name))
            return mtd;
        }
      return null;
    }
    public void ToCode(string dir)
    {
      if (MessageType != null)
        foreach (MESSAGETYPEDEF m in MessageType)
        {
          m.ToCode(dir);
        }
    }
  }
}
```

SUBCHART.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.CodeDom;
namespace LscClasses
{
  partial class SUBCHART
  {
    int i = 0;
    public Hashtable ToCode(LscSpec spec, CHART chart, CLASSDEF cls)
    {
      i = 0;
      Hashtable events = new Hashtable();
      int message = 0;
      if (Items != null)
        foreach (object obj in Items)
        {
          i++;
          Hashtable ev = null;
          EVENT e = obj as EVENT;
          if (e != null)
          {
            ev = e.ToCode(spec, chart, cls);
            EVENTDEF ed = e.Item as EVENTDEF;
            if (ed != null && ed.ItemElementName == ItemChoiceType.Message)
            {
              INSTDEF idTo = spec.Instances.GetInstance(((MESSAGE)ed.Item).To);
              CLASSDEF cdTo = spec.Classes.GetClass(idTo.Class);
              if (cdTo.Equals(cls))
                message = 1;
            }
          }
          IFTHENELSE ite = obj as IFTHENELSE;
          if (ite != null)
          {

            ev =  ite.ToCode(spec, chart, cls );

          }
          LOOP loop = obj as LOOP;
          if (loop != null)
          {
             ev = loop.ToCode(spec, cls);
          }
          events = CHART.Concatenate(events, ev);
          if (message == 1)
            return events;
        }
      return events;
    }
    public Hashtable ToCodeEnd(LscSpec spec, CHART chart, CLASSDEF cls)
```

```
    {
      Hashtable events = new Hashtable();
      object obj;
      if (Items != null)
        for (; i < Items.Length; i++ )
        {
          obj = Items[i];
          Hashtable ev = null;
          EVENT e = obj as EVENT;
          if (e != null)
          {
            ev = e.ToCode(spec, chart, cls);
          }
          IFTHENELSE ite = obj as IFTHENELSE;
          if (ite != null)
          {
            ev = ite.ToCode(spec, chart, cls);
          }
          LOOP loop = obj as LOOP;
          if (loop != null)
          {
            ev = loop.ToCode(spec, cls);
          }
          events = CHART.Concatenate(events, ev);

        }
      i = 0;
      return events;
    }

  }
}
```

VARIDEF.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom;
namespace LscClasses
{
  partial class VARIDEF
  {
    public override string ToString()
    {
      return this.Type + " " + this.VariName;
    }
    public override bool Equals(object obj)
    {
      if (obj.GetType() == typeof(VARIDEF))
        return this.VariName.Equals(((VARIDEF)obj).VariName) &&
          this.Class.Equals(((VARIDEF)obj).Class);
      return base.Equals(obj);
    }
    public override int GetHashCode()
    {
      return base.GetHashCode();
    }
    public void ToCode(CodeTypeDeclaration cls)
    {
      CodeMemberField vari = new CodeMemberField(this.Type.ToString(), this.VariName);
      vari.Attributes = MemberAttributes.Private;
      cls.Members.Add(vari);
      CodeMemberProperty variProp = new CodeMemberProperty();
      variProp.Attributes = MemberAttributes.Public | MemberAttributes.Final;
      variProp.Type = new CodeTypeReference(this.Type.ToString());
      variProp.Name = this.VariName.ToUpper();
      variProp.HasGet = true; //the getter
      variProp.GetStatements.Add(new CodeSnippetExpression("return "+VariName));
      variProp.HasSet = true; //the setter
      variProp.SetStatements.Add(new CodeSnippetExpression(VariName+" = value"));
      cls.Members.Add(variProp);
    }
  }
}
```

Appendix B

Fixed Components

BlockingQueue.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
namespace FixedComponents
{
  public class BlockingQueue<T>
  {
    private Queue<T> queue = new Queue<T>();
    private Semaphore semaphore = new Semaphore(0, int.MaxValue);
    private Semaphore empty = new Semaphore(1, 1);

    public void Enqueue(T data)
    {
      if (data == null) throw new ArgumentNullException("data");
      lock(queue)
      {
        queue.Enqueue(data);
        if (queue.Count == 1)
        {
          empty.WaitOne();
        }
      }
      semaphore.Release();
    }
    public T Dequeue()
    {
      semaphore.WaitOne();
      lock (queue)
      {
        if (queue.Count == 1)
          empty.Release();
        return queue.Dequeue();
      }
    }
    public T Peek()
    {
      semaphore.WaitOne();
      semaphore.Release();
      lock (queue)
      {
        return queue.Peek();
      }
    }
    public void WaitUntilEmpty()
    {
      empty.WaitOne();
      empty.Release();
    }
    public int Count
    {
      get
      {
        lock (queue)
          return queue.Count;
      }
    }
  }
}
```

IMessage.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace FixedComponents
{
  public interface IMessage
  {
    void SetEventHandlers(List<IMessageHandler> receptors);
```

```
    void FireEvent();
    Information Data
    {
      get;
      set;
    }

  }
}
```

IMessageHandler.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace FixedComponents
{
  public interface IMessageHandler
  {
    void HandleEvent(IMessage source);
  }
}
```

Information.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace FixedComponents
{
  public class Information
  {
    object[] info;
    public Information(int size)
    {
      info = new object[size];
    }
    public Information(object[] infos)
    {
      this.info = infos;
    }
    public object[] Info
    {
      get
      {
        return info;
      }
      set
      {
        info = value;
      }
    }
  }
}
```

Location.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace FixedComponents
{
  public class Location
  {
    string scenario;
    int state;
    Message msg;
    bool prechart;
    bool firstInChart;
    public Location(string scenario, int state, bool prechart)
    {
      this.scenario = scenario;
      this.state = state;
      this.prechart = prechart;
      if (state == 0)
        firstInChart = true;
      else
        firstInChart = false;
```

```
    }
    public Location(string scenario, int state, bool prechart, bool firstInChart)
    {
      this.scenario = scenario;
      this.state = state;
      this.prechart = prechart;
      this.firstInChart = firstInChart;
    }
    public override string ToString()
    {
      return "location (" + Scenario + ", " + State + ") ";
    }
    public string Scenario
    {
      get
      {
        return scenario;
      }
      set
      {
        scenario = value;
      }
    }
    public int State
    {
      get
      {
        return state;
      }
      set
      {
        state = value;
      }
    }
    public Message Message
    {
      get
      {
        return msg;
      }
      set
      {
        msg = value;
      }
    }
    public bool Prechart
    {
      get
      {
        return prechart;
      }
      set
      {
        prechart = value;
      }
    }
    public bool FirtsInChart
    {
      get
      {
        return firstInChart;
      }
      set
      {
        firstInChart = value;
      }
    }
  }
}
```

Message.cs

```
using System;
using System.Collections.Generic;
using System.Text;
namespace FixedComponents
{
```

```
  public abstract class Message : IMessage
  {
    List<IMessageHandler> handlers;
    protected Information data;
    protected int recipient;
    protected Type recipientType;
    string scenario;
    public Message(int recipient, Type recipientType, string scenario)
    {
      this.recipient = recipient;
      this.recipientType = recipientType;
      this.scenario = scenario;
    }
    public Message(
         int recipient, Type recipientType, string scenario, Information info)
    {
      this.recipient = recipient;
      this.recipientType = recipientType;
      this.scenario = scenario;
      this.data = info;
    }
    public void SetEventHandlers(List<IMessageHandler> handlers)
    {
      this.handlers = handlers;
    }
    public void FireEvent()
    {
      foreach (IMessageHandler handler in handlers)
      {
        handler.HandleEvent(this);
      }
    }
    public Information Data
    {
      get
      {
        return data;
      }
      set
      {
        data = value;
      }
    }
    public int Recipient
    {
      get
      {
        return recipient;
      }
    }
    public Type RecipientType
    {
      get
      {
        return recipientType;
      }
    }
    public string Scenario
    {
      get
      {
        return scenario;
      }
    }
  }
}
```

MessageQueue.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Reflection;
using System.Collections;
namespace FixedComponents
{
```

```
  public class MessageQueue
  {
    BlockingQueue<IMessage> msgQueue;
    Thread consumerThread = null;
    Hashtable typesToHandlers = new Hashtable();
    public MessageQueue()
    {
      msgQueue = new BlockingQueue<IMessage>();
      ThreadStart starter = new ThreadStart(ProcessMessages);
      consumerThread = new Thread(starter);
      consumerThread.Name = "MessQueue";
      consumerThread.Start();
    }
    public void RegisterEventHandler(IMessageHandler handler, ArrayList msgTypes)
    {
      foreach (Type crtType in msgTypes)
      {
        if (typesToHandlers.Contains(crtType))
        {
          // Getting the event-handlers that already registered this event
          List<IMessageHandler> eventHandlers =
                      (List<IMessageHandler>)typesToHandlers[crtType];
          // Adding the new handler to the list
          eventHandlers.Add(handler);
        }
        else
        {
          List<IMessageHandler> eventHandlers = new List<IMessageHandler>();
          eventHandlers.Add(handler);
          typesToHandlers.Add(crtType, eventHandlers);
        }
      }
    }
    public void EnqueueMessage(IMessage msg)
    {
      //Console.WriteLine("Enqueued message of type " + msg.GetType());
      msgQueue.Enqueue(msg);
    }
    private void ProcessMessages()
    {
      while (true)
      {
        IMessage msg = msgQueue.Dequeue();
        Type type = msg.GetType();
        // Event handling
        if (!typesToHandlers.Contains(type))
          continue;
        List<IMessageHandler> handlers =
         (List<IMessageHandler>)typesToHandlers[type];
        msg.SetEventHandlers(handlers);
        msg.FireEvent();
      }
    }
  }
}
```

MessageHandler.cs

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using System.Threading;
using System.IO;
namespace FixedComponents
{
  public abstract class MessageHandler :IMessageHandler
  {
    protected Hashtable scenarioToState = new Hashtable();
    private Hashtable copyOfScenarioToState;
    protected MessageQueue queue = null;
    protected Hashtable msgTypesToAvailableLocations = new Hashtable();
    protected Hashtable scenarioToActivateMode = new Hashtable();
    protected Thread thread;
    protected BlockingQueue<Location> internalQueue;
    protected Hashtable scenarioToInternalQueue = new Hashtable();
    protected Logger logger;
```

```
protected string name;
public MessageHandler(MessageQueue queue)
{
  this.queue = queue;
  this.internalQueue = new BlockingQueue<Location>();
  this.thread = new Thread(new ThreadStart(Run));
  thread.Start();
  InitEventHandling();
}
public void HandleEvent(IMessage msg)
{
  logger.append(" ");
  logger.append(" Handled event of type " + msg.GetType());
  if (GetType() == ((Message)msg).RecipientType
    && GetHashCode() != ((Message)msg).Recipient)
  {
    logger.append(" Event not accepted - different recipient");
    return;
  }
  ((BlockingQueue<Location>)internalQueue).WaitUntilEmpty();
  copyOfScenarioToState = (Hashtable)CopyOfScenarioToState();
  ArrayList availableLocations =
   (ArrayList)msgTypesToAvailableLocations[msg.GetType()];
  foreach (Location availableLocation in availableLocations)
  {
    ((BlockingQueue<Location>)
             scenarioToInternalQueue[availableLocation.Scenario]).
      WaitUntilEmpty();
    ArrayList actualStates = GetStateScenario(availableLocation.Scenario);
    int activ = 0;
    for (int i = 0; i < actualStates.Count; i++)
    {
      int actualState = (int)actualStates[i];
      Location actualLocation = new Location(
                  availableLocation.Scenario, actualState, false);
      if (availableLocation.State == actualState
        && (actualState != 0
          || (actualState == 0
                  && availableLocation.Scenario == ((Message)msg).Scenario)
          || (actualState == 0 && availableLocation.Prechart)))
      {
        logger.append(" Event accepted in " + actualLocation);
        actualLocation.Message = (Message)msg;
        internalQueue.Enqueue(actualLocation);
        ((BlockingQueue<Location>)
                        scenarioToInternalQueue[availableLocation.Scenario]).
                        Enqueue(actualLocation);
        activ = 1;
      }
      else
      {
        logger.append(" Event not accepted in " + actualLocation);
      }
    }
    if (activ == 0 && availableLocation.FirtsInChart
      && (availableLocation.Prechart
        || availableLocation.Scenario == ((Message)msg).Scenario)
      )
    {
      logger.append(" Event accepted in " + availableLocation);
      availableLocation.Message = (Message)msg;
      internalQueue.Enqueue(availableLocation);
      ((BlockingQueue<Location>)
       scenarioToInternalQueue[availableLocation.Scenario]).
       Enqueue(availableLocation);
    }
  }
}
private void Run()
{
  while (true)
  {
    Location location = internalQueue.Dequeue();
    ((BlockingQueue<Location>)scenarioToInternalQueue[location.Scenario]).Peek();
    logger.append(location.Message.GetType()+" dequeued from internal queue ");
    GoToNextState(location);
```

```
   ((BlockingQueue<Location>)
       scenarioToInternalQueue[location.Scenario]).
       Dequeue();
  }
}
protected void SetStateScenario(Location loc)
{
  SetStateScenario(loc, loc.State + 1);
}
protected void SetStateScenario(Location loc, int newState)
{
  lock (queue)
  {
    if (scenarioToState.Contains(loc.Scenario))
    {
      ArrayList states = (ArrayList)scenarioToState[loc.Scenario];
        states.Add(newState);
    }
    else
    {
      scenarioToState.Add(loc.Scenario, new ArrayList(new int[] { newState }));
    }
  }
}
protected void RemoveScenario(Location loc)
{
  lock (queue)
  {
    if (scenarioToState.Contains(loc.Scenario))
    {
      ArrayList states = (ArrayList)scenarioToState[loc.Scenario];
      for (int i = 0; i < states.Count; i++)
      {
        if ((int)states[i] == loc.State)
        {
          states.RemoveAt(i);
          logger.append(" Remove location: " + loc);
          break;
        }
      }
      if (states.Count == 0)
      {
        scenarioToState.Remove(loc.Scenario);
        logger.append(" Remove location with scenario: " + loc.Scenario);
      }
    }
  }
}
private ArrayList GetStateScenario(string scenario)
{
  ArrayList availableStates;
    if (copyOfScenarioToState.Contains(scenario))
    {
      availableStates = (ArrayList)copyOfScenarioToState[scenario];
    }
    else
    {
      availableStates = new ArrayList();
    }
    return availableStates;
}
private Hashtable CopyOfScenarioToState()
{
  copyOfScenarioToState = new Hashtable();
  foreach (string scenario in scenarioToState.Keys)
  {
    ArrayList states = (ArrayList) scenarioToState[scenario];
    copyOfScenarioToState.Add(scenario, new ArrayList(states));
  }
  return copyOfScenarioToState;
}
private int IsScenarioActiv(string scenario)
{
  lock (queue)
  {
    if (scenarioToState.Contains(scenario))
```

```
        {
          return 1;
        }
        return 0;
      }
    }
    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;

      }
    }
    public override int GetHashCode()
    {
      return Name.GetHashCode();
    }
    abstract public void InitEventHandling();
    abstract public void GoToNextState(Location location);
  }
}
```

Generated Components

CONTROLLER.ssc

```
using System;
using FixedComponents;
using System.Collections;
using System.IO;
using System.Threading;
using System.Diagnostics;
//^using Microsoft.Contracts;
namespace GeneratedComponents
{
    public class CONTROLLER : MessageHandler
    {
        //begin not generated
        public int N = 6;
        public int[]/*!*/ int_req =  new int[0];
        public int[]/*!*/ ext_req = new int[0];
        private int dir;
        private int pos = 1;
        private int status;
        //end
        // ^ invariant this.ValidatePos();

       //^[Microsoft.Contracts.NotDelayed]
       public CONTROLLER(MessageQueue/*!*/ queue, string name) :base(queue)
       {
           this.name = name;
           thread.Name = name;
           logger = new Logger(name, Logger.ALL);
            //begin not generated
            int_req = new int[N];
            ext_req = new int[N];
            dir = 0;
            status = 0;
            //end
       }

       //^[Microsoft.Contracts.Confined]
       public bool NotNullRequests()
       {
           return this.int_req != null && this.ext_req!= null;
       }

       //^[Microsoft.Contracts.Confined]
        public bool ReqEmpty()
           //^requires NotNullRequests();
        {
```

```
    int sum = 0;
    for (int i = 0; i < N; i++)
    {
        sum += int_req[i] + ext_req[i] * ext_req[i];
    }
    return sum == 0;
}
public void UpdateIntReq(int level)
   //^requires NotNullRequests();
{
    int_req[level] = 1;
}
public void UpdateExtReq(int level, int dir)
   //^requires NotNullRequests();
{
    ext_req[level] = dir;
}
public void SetStatusHolding()
{
    status = 0;
}
public void SetStatusMoving()
{
    status = 1;
}
public void TurnAround()
{
    dir = dir * (-1);
}
public void ClearReq()
   //^requires NotNullRequests();
{
    int_req[pos] = 0;
    if (ext_req[pos] == dir)
        ext_req[pos] = 0;
}
public void SetDirUp()
{
    dir = 1;
}
public void SetDirDown()
{
    dir = -1;
}

public void SetDirZero()
{
    dir = 0;
}
//^[Microsoft.Contracts.Confined]
public bool ExistReqAtSameDir()
   //^requires NotNullRequests();
{
    if (dir == 0)
        return false;
    int sum = 0;
    for (int i = pos + dir; i >= 0 && i < N; i = i + dir)
    {
        sum += int_req[i] + ext_req[i] * ext_req[i];
    }
    return sum > 0;
}
//^[Microsoft.Contracts.Confined]
public bool ExistReqAtOppDir()
   //^requires NotNullRequests();
{
    if (dir == 0)
        return false;
    int sum = 0;
    for (int i = pos - dir; i >= 0 && i < N; i = i - dir)
    {
        sum += int_req[i] + ext_req[i] * ext_req[i];
    }
    return sum > 0;
}
//^[Microsoft.Contracts.Confined]
```

```
public bool ValidatePos()
{
    return pos >= 0 && pos < N;
}
//^[Microsoft.Contracts.Confined]
public bool ToStop()
   //^requires NotNullRequests();
{
    return pos == N - 1 || pos == 0 || int_req[pos] == dir
   || ext_req[pos] == dir || (ext_req[pos] != 0 && !ExistReqAtSameDir());
}
//^[Microsoft.Contracts.Confined]
public bool ExistReqAtPos()
   //^requires NotNullRequests();
{
    return int_req[pos] == 1 || ext_req[pos] != 0;
}
//^[Microsoft.Contracts.Confined]
public bool GetToLevel(int level)
{
    return level == pos && status == 0;
}
//^[Microsoft.Contracts.Confined]
public bool IsAtLevel(int level, int dir)
{
    return level == pos && ((status == 0 && dir == 0)
       || (status == 0 && dir == this.dir));
}
//^[Microsoft.Contracts.Confined]
public bool IsLiftFree(int level)
{
    return status == 0 && dir == 0;
}
//^[Microsoft.Contracts.Confined]
public bool BeforeLevel(int level)
{
    return level > pos;
}
//^[Microsoft.Contracts.Confined]
public bool AfterLevel(int level)
{
    return level < pos;
}
public void ModifyLevel()
{
    pos += dir;
}
public override void InitEventHandling()
{
    ArrayList types = new ArrayList();
    types.Add(Type.GetType("GeneratedComponents.IntReq"));
    types.Add(Type.GetType("GeneratedComponents.Opened"));
    types.Add(Type.GetType("GeneratedComponents.Arriving"));
    types.Add(Type.GetType("GeneratedComponents.Stopped"));
    types.Add(Type.GetType("GeneratedComponents.ExtReq"));
    types.Add(Type.GetType("GeneratedComponents.Closed"));
    scenarioToInternalQueue.Add("ServeInitIntRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ReopenDoorFromInside",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("NoMoreRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ExternalRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("OpenCloseDoor",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ServeInitExtRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("Arrive", new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ReopenDoorFromOutside",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("InternalRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ServeNextRequest",
        new BlockingQueue<Location>());
    queue.RegisterEventHandler(this, types);
```

```
    PopulateAvailableScenarios();
}

public void PopulateAvailableScenarios()
{
    ArrayList availableLocations;
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("InternalRequest", 0, true, true));
    availableLocations.Add(new Location("ReopenDoorFromInside",0, true, true));
    availableLocations.Add(new Location("ServeInitIntRequest",0, true, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.IntReq"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("Arrive", 2, false, false));
    availableLocations.Add(new Location("OpenCloseDoor", 0, true, true));
    availableLocations.Add(new Location("ReopenDoorFromInside",1,false,false));
    availableLocations.Add(new Location("ReopenDoorFromOutside",1,false,false);
    availableLocations.Add(new Location("ServeNextRequest", 2, false, false));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Opened"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("Arrive", 0, true, true));
    availableLocations.Add(new Location("Arrive", 3, false, false));
    availableLocations.Add(new Location("ServeInitExtRequest",1,false, false));
    availableLocations.Add(new Location("ServeInitExtRequest",2, false,false));
    availableLocations.Add(new Location("ServeInitIntRequest", 1,false false));
    availableLocations.Add(new Location("ServeInitIntRequest",2,false, false));
    availableLocations.Add(new Location("ServeNextRequest", 1, false, false));
    availableLocations.Add(new Location("ServeNextRequest", 3, false, false));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Arriving"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("Arrive", 1, false, false));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Stopped"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("ExternalRequest", 0, true, true));
    availableLocations.Add(new Location("ReopenDoorFromOutside",0,true, true));
    availableLocations.Add(new Location("ServeInitExtRequest", 0, true, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.ExtReq"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("NoMoreRequest", 0, true, true));
    availableLocations.Add(new Location("OpenCloseDoor", 1, false, false));
    availableLocations.Add(new Location("ServeNextRequest", 0, true, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Closed"), availableLocations);
}

public override void GoToNextState(Location location)
{
    //^assume location != null;
    if ("Arrive".Equals(location.Scenario))
    {
        if (location.State == 0)
        {
            if ((this.ValidatePos() == false))
            {
                this.RemoveScenario(location);
                Debug.Assert( ValidatePos());
            }
            if (this.ToStop())
            {
                queue.EnqueueMessage(new Stop("Shaft".GetHashCode(),
                Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                SetStateScenario(location, 1);
            }
            else
            {
                queue.EnqueueMessage(new KeepMoving("Shaft".GetHashCode(),
                Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                this.ModifyLevel();
                SetStateScenario(location, 3);
            }
        }
        if (location.State == 1)
```

```
    {
        this.SetStatusHolding();
        queue.EnqueueMessage(new Open("Door".GetHashCode(),
        Type.GetType("GeneratedComponents.DOOR"), location.Scenario));
        SetStateScenario(location, 2);
    }
    if (location.State == 2)
    {
        this.ClearReq();
    }
    if (location.State == 3)
    {
    }
}
if ("ExternalRequest".Equals(location.Scenario))
{
    if (location.State == 0)
    {
        Information info = location.Message.Data;
        //^assume info!=null && info.Info != null;
        this.UpdateExtReq((int)info.Info[0], (int)info.Info[1]);
    }
}
if ("InternalRequest".Equals(location.Scenario))
{
    if (location.State == 0)
    {
        Information info = location.Message.Data;
        //^assume info!=null && info.Info != null;
        this.UpdateIntReq((int)info.Info[0]);
    }
}
if ("NoMoreRequest".Equals(location.Scenario))
{
    if (location.State == 0)
    {
        if ((this.ReqEmpty() == false))
        {
            this.RemoveScenario(location);
            return;
        }
        this.SetDirZero();
        this.SetStatusHolding();
    }
}
if ("OpenCloseDoor".Equals(location.Scenario))
{
    if (location.State == 0)
    {
        Thread.Sleep(1000);
        Console.WriteLine(this + ": Waited for 5 s");
        queue.EnqueueMessage(new Close("Door".GetHashCode(),
            Type.GetType("GeneratedComponents.DOOR"), location.Scenario));
        SetStateScenario(location, 1);
    }
    if (location.State == 1)
    {
    }
}
if ("ReopenDoorFromInside".Equals(location.Scenario))
{
    if (location.State == 0)
    {
        Information info = location.Message.Data;
        //^assume info!=null && info.Info != null;
        if ((this.GetToLevel((int)info.Info[0]) == false))
        {
            this.RemoveScenario(location);
            return;
        }
        queue.EnqueueMessage(new Open("Door".GetHashCode(),
            Type.GetType("GeneratedComponents.DOOR"), location.Scenario));
        SetStateScenario(location, 1);
    }
    if (location.State == 1)
    {
```

```
            }
        }
        if ("ReopenDoorFromOutside".Equals(location.Scenario))
        {
            if (location.State == 0)
            {
                Information info = location.Message.Data;
                //^assume info!=null && info.Info != null;
                if ((this.IsAtLevel((int)info.Info[0],(int)info.Info[1]) == false))
                {
                    this.RemoveScenario(location);
                    return;
                }
                queue.EnqueueMessage(new Open("Door".GetHashCode(),
Type.GetType("GeneratedComponents.DOOR"), location.Scenario));
                SetStateScenario(location, 1);
            }
        }
        if ("ServeInitExtRequest".Equals(location.Scenario))
        {
            if (location.State == 0)
            {
                Information info = location.Message.Data;
                //^assume info!=null && info.Info != null;
                if ((this.IsLiftFree((int)info.Info[0]) == false))
                {
                    this.RemoveScenario(location);
                    return;
                }
                if (this.BeforeLevel((int)location.Message.Data.Info[0]))
                {
                    this.SetDirUp();
                    this.SetStatusMoving();
                    this.ModifyLevel();
                    queue.EnqueueMessage(new Movingup("Shaft".GetHashCode(),
                    Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                    SetStateScenario(location, 1);
                }
                else
                {
                    if ((AfterLevel((int)location.Message.Data.Info[0])==false))
                    {
                        this.RemoveScenario(location);
                        return;
                    }
                    this.SetDirDown();
                    this.SetStatusMoving();
                    this.ModifyLevel();
                    queue.EnqueueMessage(new MovingDown("Shaft".GetHashCode(),
                    Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                    SetStateScenario(location, 2);
                }
            }
        }
        if ("ServeInitIntRequest".Equals(location.Scenario))
        {
            if (location.State == 0)
            {
                Information info = location.Message.Data;
                //^assume info!=null;
                if ((this.IsLiftFree((int)info.Info[0]) == false))
                {
                    this.RemoveScenario(location);
                    return;
                }
                if (this.BeforeLevel((int)location.Message.Data.Info[0]))
                {
                    this.SetDirUp();
                    this.SetStatusMoving();
                    this.ModifyLevel();
                    queue.EnqueueMessage(new Movingup("Shaft".GetHashCode(),
                    Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                    SetStateScenario(location, 1);
                }
                else
                {
```

```
                    if ((AfterLevel((int)location.Message.Data.Info[0]) == false))
                    {
                        this.RemoveScenario(location);
                        return;
                    }
                    this.SetDirDown();
                    this.SetStatusMoving();
                    this.ModifyLevel();
                    queue.EnqueueMessage(new MovingDown("Shaft".GetHashCode(),
                    Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                    SetStateScenario(location, 2);
                }
            }
            if (location.State == 1)
            {
            }
            if (location.State == 2)
            {
            }
        }
        if ("ServeNextRequest".Equals(location.Scenario))
        {
            if (location.State == 0)
            {
                if (this.ExistReqAtSameDir())
                {
                    queue.EnqueueMessage(new KeepMoving("Shaft".GetHashCode(),
                    Type.GetType("GeneratedComponents.SHAFT"), location.Scenario));
                    this.SetStatusMoving();
                    this.ModifyLevel();
                    SetStateScenario(location, 1);
                }
                else
                {
                    if (this.ExistReqAtPos())
                    {
                        queue.EnqueueMessage(new Open("Door".GetHashCode(),
                        Type.GetType("DOOR"), location.Scenario));
                        SetStateScenario(location, 2);
                    }
                    else
                    {
                        if ((this.ExistReqAtOppDir() == false))
                        {
                            this.RemoveScenario(location);
                            return;
                        }
                        this.TurnAround();
                        queue.EnqueueMessage(new TurnAround ("Shaft".GetHashCode(),
                        Type.GetType("SHAFT"), location.Scenario));
                        this.SetStatusMoving();
                        this.ModifyLevel();
                        SetStateScenario(location, 3);
                    }
                }
            }
            if (location.State == 1)
            {
            }
            if (location.State == 2)
            {
                this.ClearReq();
            }
            if (location.State == 3)
            {
            }
        }
        RemoveScenario(location);
    }
  }
}
```

DOOR.ssc

```
using System;
using FixedComponents;
using System.Collections;
using System.IO;
using System.Threading;
using System.Diagnostics;
namespace GeneratedComponents
{
    public class DOOR : MessageHandler
    {

        public DOOR(MessageQueue queue, string name) :
                base(queue)
        {
            this.name = name;
            thread.Name = name;
            logger = new Logger(name, Logger.ALL);
        }

        public override void InitEventHandling()
        {
            ArrayList types = new ArrayList();
            types.Add(Type.GetType("GeneratedComponents.Open"));
            types.Add(Type.GetType("GeneratedComponents.Close"));
            scenarioToInternalQueue.Add("ReopenDoorFromInside",
                new BlockingQueue<Location>());
            scenarioToInternalQueue.Add("ServeNextRequest",
                new BlockingQueue<Location>());
            scenarioToInternalQueue.Add("ReopenDoorFromOutside",
                new BlockingQueue<Location>());
            scenarioToInternalQueue.Add("OpenCloseDoor",
                new BlockingQueue<Location>());
            scenarioToInternalQueue.Add("Arrive", new BlockingQueue<Location>());
            queue.RegisterEventHandler(this, types);
            PopulateAvailableScenarios();
        }

        public void PopulateAvailableScenarios()
        {
            ArrayList availableLocations;
             availableLocations = new ArrayList();
            availableLocations.Add(new Location("Arrive", 0, false, true));
            availableLocations.Add(new Location("OpenCloseDoor", 0, true, true));
            availableLocations.Add(
                new Location("ReopenDoorFromInside", 0, false, true));
            availableLocations.Add(
                new Location("ReopenDoorFromOutside", 0, false, true));
            availableLocations.Add(
                new Location("ServeNextRequest", 0, false, true));
            msgTypesToAvailableLocations.Add(
                Type.GetType("GeneratedComponents.Open"), availableLocations);
             availableLocations = new ArrayList();
            availableLocations.Add(
                new Location("OpenCloseDoor", 1, false, false));
            msgTypesToAvailableLocations.Add(
                Type.GetType("GeneratedComponents.Close"), availableLocations);
        }

        public override void GoToNextState(Location location)
        {
            if ("Arrive".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(
                        new Opened("Controller".GetHashCode(),
                        Type.GetType("GeneratedComponents.CONTROLLER"),
                        location.Scenario));
                }
            }
            if ("ExternalRequest".Equals(location.Scenario))
            {
            }
            if ("InternalRequest".Equals(location.Scenario))
            {
```

```
            }
            if ("NoMoreRequest".Equals(location.Scenario))
            {
            }
            if ("OpenCloseDoor".Equals(location.Scenario))
            {
                if (location.State == 0)
                    SetStateScenario(location, 1);
                if (location.State == 1)
                    queue.EnqueueMessage(
                    new Closed("Controller".GetHashCode(),
                    Type.GetType("CONTROLLER"), location.Scenario));
            }
            if ("ReopenDoorFromInside".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(
                    new Opened("Controller".GetHashCode(),
                    Type.GetType("CONTROLLER"), location.Scenario));
                }
            }
            if ("ReopenDoorFromOutside".Equals(location.Scenario))
            {
                if (location.State == 0)
                    queue.EnqueueMessage(
                    new Opened("Controller".GetHashCode(),
                    Type.GetType("CONTROLLER"), location.Scenario));
            }
            if ("ServeInitExtRequest".Equals(location.Scenario))
            {
            }
            if ("ServeInitIntRequest".Equals(location.Scenario))
            {
            }
            if ("ServeNextRequest".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(
                    new Opened("Controller".GetHashCode(),
                    Type.GetType("CONTROLLER"), location.Scenario));
                }
            }
            RemoveScenario(location);
        }
    }
}
```

SHAFT.ssc

```
using System;
using FixedComponents;
using System.Collections;
using System.IO;
using System.Threading;
using System.Diagnostics;
namespace GeneratedComponents
{
    public class SHAFT : MessageHandler
    {

        public SHAFT(MessageQueue queue, string name) :
                base(queue)
        {
            this.name = name;
            thread.Name = name;
            logger = new Logger(name, Logger.ALL);
        }

        public override void InitEventHandling()
        {
            ArrayList types = new ArrayList();
            types.Add(Type.GetType("GeneratedComponents.TurnAroundMoving"));
            types.Add(Type.GetType("GeneratedComponents.Movingup"));
            types.Add(Type.GetType("GeneratedComponents.MovingDown"));
            types.Add(Type.GetType("GeneratedComponents.KeepMoving"));
```

```
    types.Add(Type.GetType("GeneratedComponents.Stop"));
    scenarioToInternalQueue.Add("ServeNextRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ServeInitExtRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("ServeInitIntRequest",
        new BlockingQueue<Location>());
    scenarioToInternalQueue.Add("Arrive", new BlockingQueue<Location>());
    queue.RegisterEventHandler(this, types);
    PopulateAvailableScenarios();
}

public void PopulateAvailableScenarios()
{
    ArrayList availableLocations;
    availableLocations = new ArrayList();
    availableLocations.Add(
        new Location("ServeNextRequest", 1, false, true));
    msgTypesToAvailableLocations.Add(
    Type.GetType("GeneratedComponents.TurnAroundMoving"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(
        new Location("ServeInitExtRequest", 0, false, true));
    availableLocations.Add(
        new Location("ServeInitIntRequest", 0, false, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Movingup"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(
        new Location("ServeInitExtRequest", 1, false, true));
    availableLocations.Add(
        new Location("ServeInitIntRequest", 1, false, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.MovingDown"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("Arrive", 1, false, true));
    availableLocations.Add(new Location("ServeNextRequest", 0, false, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.KeepMoving"), availableLocations);
    availableLocations = new ArrayList();
    availableLocations.Add(new Location("Arrive", 0, false, true));
    msgTypesToAvailableLocations.Add(
        Type.GetType("GeneratedComponents.Stop"), availableLocations);
}

public override void GoToNextState(Location location)
{
    if ("Arrive".Equals(location.Scenario))
    {
        if (location.State == 0)
        {
            queue.EnqueueMessage(new Stopped("Controller".GetHashCode(),
                Type.GetType("CONTROLLER"), location.Scenario));
        }
        if (location.State == 1)
        {
            queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                Type.GetType("CONTROLLER"), location.Scenario));
        }
    }
    if ("ExternalRequest".Equals(location.Scenario))
    {
    }
    if ("InternalRequest".Equals(location.Scenario))
    {
    }
    if ("NoMoreRequest".Equals(location.Scenario))
    {
    }
    if ("OpenCloseDoor".Equals(location.Scenario))
    {
    }
    if ("ReopenDoorFromInside".Equals(location.Scenario))
    {
    }
    if ("ReopenDoorFromOutside".Equals(location.Scenario))
```

```
            {
            }
            if ("ServeInitExtRequest".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
                if (location.State == 1)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
            }
            if ("ServeInitIntRequest".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
                if (location.State == 1)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
            }
            if ("ServeNextRequest".Equals(location.Scenario))
            {
                if (location.State == 0)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
                if (location.State == 1)
                {
                    queue.EnqueueMessage(new Arriving("Controller".GetHashCode(),
                        Type.GetType("CONTROLLER"), location.Scenario));
                }
            }
            RemoveScenario(location);
        }
    }
}
```

Program.ssc

```
using System;
using FixedComponents;
using System.Collections;
using System.IO;
using System.Threading;
namespace GeneratedComponents
{
    public class Program
    {
        public static void Main(string [] args)
        {
            MessageQueue queue = new MessageQueue();
            CONTROLLER Controller = new CONTROLLER(queue, "Controller");
            DOOR Door = new DOOR(queue, "Door");
            USER User = new USER(queue, "User");
            SHAFT Shaft = new SHAFT(queue, "Shaft");
        }
    }
}
```