Design Synthesis from Interaction and State-Based Specifications

Jun Sun and Jin Song Dong

Abstract—Interaction-based and state-based modeling are two complementary approaches of behavior modeling. The former focuses on global interactions between system components. The latter concentrates on the internal states of individual components. Both approaches have been proven useful in practice. One challenging and important research objective is to combine the modeling power of both effectively and then use the combination as the basis for automatic design synthesis. We present a combination of interaction-based and state-based modeling, namely, Live Sequence Charts and Z, for system specification. We then propose a way of generating distributed design from the combinations. Our approach handles systems with intensive interactive behaviors as well as complex state structures.

Index Terms—Z language, live sequence charts, specification, synthesis.

1 Introduction

Behavior modeling plays an important role in the engineering of software-based systems. It is the basis for systematic approaches for specification, design, code generation, testing, and verification. Two complementary approaches for modeling behavior have been proven useful in practice. One is interaction-based, which focuses on global interactions between system components, e.g., Message Sequence Charts (MSC) [46] and Live Sequence Charts (LSC) [9]. The other is state-based modeling, which concentrates on the internal states of individual components, e.g., Z [49], VDM [23], and Statechart [14]. Large scale systems often have not only complex data structures but also intensive interactive behaviors relying on the underlying data objects. In this work, we propose a combination of interaction and state-based modeling, i.e., a complete system specification consisting of two separate parts: an LSC part for capturing interactions between system components and a Z part for modeling the data and functional aspects. The significant and novel aspect is that the combination combines the modeling power of both so that it can be used to specify systems beyond the capability of either one. Moreover, such combined specifications contain sufficient information for mechanized synthesis of distributed system designs, which may lead directly to implementation.

The first contribution of this work is the integration of LSC and Z. LSC is a rather rich extension to MSC. It allows specification of not only possible behaviors, but also mandatory ones. We choose Z because it is widely known and accepted and well-developed in terms of specification, refinement, etc. The second contribution is our approach to solving the synthesis problem of the combined specifications. Synthesis from specifications like scenario-based diagrams or various automata is extremely hard [34], [35], [15], [4]. To the best of our knowledge, our work is the first attempt to synthesize low-level implementations from combined formal system specification of interactive-based and state-based modeling. Our aim is to discover a practical way of synthesizing sound implementations. Due to the complexity of the problem, the synthesized implementation may not exhibit all the runs allowed by the specification.

We take a step-by-step approach. First, a distributed object system is synthesized from the LSC part of the specification by treating local actions as abstract events. The global state machine is never constructed during the step so as to avoid state space explosion. Meanwhile, an abstract finite state machine is constructed from the Z part using automated predicate abstraction [1], [29], which allows us to grasp the behaviors of the objects based on a finite set of data assertions. Second, the distributed object system is refined on an object basis to guarantee that the preconditions of the local actions (Z operations) and hot conditions in the LSC model will not be violated. Additional crucial properties for open systems, like nonblocking and uncontrollability of the environment, are also taken into account. Finally, we may synthesize executable implementation by generating code from the refined finite state machines (the design). Our method is implemented as a Java application.

The remainder of the paper is organized as follows: Section 1.1 discusses the related works. Section 2 briefly introduces the relevant features of Z and LSC and proposes an intuitive integration. Section 3 presents a systematic way of synthesizing a distributed object system from the LSC model. Section 4 refines the distributed object system based on abstract views of the Z model. Section 5 discusses our prototype. Section 6 concludes the paper.

1.1 Related Works

The part of our work on synthesizing distributed object systems from LSC is related to attempts at synthesizing

• *The authors are with the Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543, Republic of Singapore. E-mail: {sunj, dongjs}@compu.nus.edu.sg.*

Manuscript received 21 Apr. 2005; revised 10 Apr. 2006; accepted 1 June 2006; published online 23 June 2006.
Recommended for acceptance by S. Uchitel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0095-0405.

state machines from interaction-based modeling. The synthesis problem of LSC has been discussed by Harel and Kugler. In [15], they tackled the problem by defining the notion of consistency of an LSC model. Their approach starts with constructing a *global system automaton* and decomposing it by different means. Their approach suffers from the state explosion problem due to the construction of the *global system automaton*, which is often of huge size because of the distributed nature of LSC and the underlying weak partial ordering semantics. In our work, a distributed object system is synthesized directly from the LSCs. We avoid constructing the global state machine by using a bounded set of synchronization barriers for monitoring the completion of universal charts locally. In [5], Bontemps et al. discussed the synthesis problem for a small subset of LSC (LSC without conditions, structuring constructs, modalities on locations, and messages). They proposed a game-based semantics for LSC, which leads to the notion of the consistency of their LSC. Their work is later extended to handle all LSC constructs but unbounded loop in [2]. In our approach, almost all LSC constructs are supported except complex time-related ones, which deserve a complicated discussion and are thus left to future work. More importantly, we present a way of synthesizing distributed implementations from LSCs. In [4], Bontemps and Schobbens investigated the complexity of various problems of LSC. The results are fairly negative, i.e., they showed the distributed realization problem is undecidable. Later, a number of lightweight approaches to the synthesis problem were proposed. For instance, Harel et al. reinvestigated the problem in [16]. They generate Statecharts [14] from LSC and then verify them for correctness, thus avoiding undecidability. In our work, we use a set of special events (bounded by the maximum number of overlapping activations of the universal charts and the number of the universal charts) to avoid undecidability. Thus, our work can be viewed as a lightweight approach to the problem as well. Another lightweight approach is evidenced in [3], where Bontemps et al. proposed a technique coupling translation and verification to cope with undecidability. We remark that such an approach certainly works in our context. Additionally, the work described in [24] synthesizes a timed Büchi Automaton from a single universal chart. What makes our goal both harder and more interesting is the treatment of a set of charts, not just a single one. There have been a number works on formalizing and then synthesizing from the classic MSC [48], [25], [44], [18], [45], [28], [26]. Compared to synthesis from MSC, synthesis from LSC is considerably more complicated as LSC offers more features than MSC. Nevertheless, the fact that LSC can be used to specify complete system behaviors also makes synthesis from LSC more useful.

Our approach offers a promising method to apply synthesis techniques to system specifications containing scenarios as well as complex data requirements. As far as the authors know, this work is the first on integrating state-based formalism with scenario-based formalisms for system specification and development. The part of our work on linking LSC and Z is remotely related to works on integrating interaction-based (or event-based) formalisms with state-based formalisms [38], [6], [39], [31]. Most of the works on integrating formalisms focus on specifying and reasoning about the integrated specifications. Our work goes beyond to synthesizing finite state designs.

2 Background

2.1 The Z Specification Language

Z [49] 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 [21]. 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.

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

$$
\begin{array}{|l}
\hline
\mathit{Light} \\
\hline
dim : 0 \,..\, 100 \\
on : \mathbb{B} \\
\hline
dim > 0 \Leftrightarrow on = true \\
\hline
\end{array}
$$

The declaration part contains the declaration of two variables. The variable dim represents the illumination of the light object, which is of value from 0 to 100 (in percent) and on is a Boolean variable indicating whether the light object is on or not. The predicate part, referred to as the state invariant, places a constraint on the values of the two variables, i.e., the dim is nonzero if and only if the light object is on.

A Z specification typically consists of a number of state and operation schemas. A state schema groups together state variables, e.g., the $Light$ schema. An operation schema defines the relationship between the "before" and "after" valuations of one or more state schemas. Z is capable of specifying open system, i.e., systems which constantly interact with their environment. External inputs to an operation schema are denoted as variables followed by a question mark.

Example. The operation schema defines the operation $Adjust$ by how the state variables of the $Light$ schema is updated.

Adjust

$\Delta Light$

$dim? : 0..100$

$on = true \land dim' = dim?$

The schema name after Δ indicates the state schemas to be updated by the operation. The variable $dim?$ is an input from the environment. The state-update is expressed using a predicate involving both primed and unprimed state variables. In particular, the operation can only be applied when the light is on and, after the operation, the light level is set to be $dim?$. If $dim?$ is zero, the state variable on will be set to false because of the state invariant.

We shall partition a large Z specification into packages. A Z package contains one state schema, one initial schema (e.g., $LightInit$ below) which identifies the initial valuation of the state schema, and a number of operation schemas that may update the state schema. In other words, a Z package identifies the state space of a set of objects.

Example.

TurnOn

$\Delta Light$

$on = false \land dim' = 100 \land on' = true$

TurnOff

$\Delta Light$

$on = true \land dim' = 0 \land on' = false$

LightInit

$Light'$

$dim' = 0 \land on' = false$

The schemas in the above and schema $Light$, $Adjust$ constitute a Z package of the $Light$ objects. The initial schema $LightInit$ states that, initially, the light is off. Operation schemas are defined to turn on or turn off the light or set the light level to a specific level.

The $Light$ model is a part of the Light Control System (LCS) [12], which is an intelligent control system. It detects the occupation of a building, then turns off the light automatically or tunes the illumination in the building according to the outside light level. We use the LCS as a running example to demonstrate our work. The system contains three active components, a *light*, a *motion detector* which detects movements in the room periodically, and a *room controller* that coordinates the light and the motion detector. The data structure of the motion detector is trivial.

Example. The Z package of the *room controller* contains the schemas shown in Fig. 1. The variable dim represents the light level (in the *room controller*'s knowledge). Initially, it is of value 0. The operation $Tune$ computes the desired light level according to the outside light level.

Operation schemas in a Z package are given a standard Z semantics, which is used to develop a transition-system semantics [49]. The Z operation semantics is best viewed as describing a relation between initial and final states of an operation. The Z precondition of an operation schema describes the initial states for which the outcome of the operation is properly defined, i.e., the domain of the operation. If an operation is invoked outside its domain, the system diverges.

If $Operation$ is an operation schema, we write $\mathrm{pre}(Operation)$ to denote its precondition. If the state schema is denoted as $State$, and $inputs$ ($outputs$) is the list of inputs (outputs) associated with the operation, then the precondition is defined as:

$$\mathrm{pre}(Operation) \mathrel{\widehat{=}} \exists State'; outputs \bullet Operation \setminus outputs,$$

where $\exists$ is an existential quantifier, the predicate after $\bullet$ is quantified by variables defined before $\bullet$. The schema $Operation \setminus outputs$ may be obtained by existentially quantifying each component in $outputs$ within $Operation$. The precondition hides any component that corresponds to the state after the operation and any output that happens to be present. If a state $(State_a)$[1] satisfies the precondition of $Operation$, we write $\mathrm{post}(Operation, State_a)$ to denote the final state that can be reached from $State_a$ by engaging $Operation$.

$$\mathrm{post}(Operation, State_a) \mathrel{\widehat{=}} \exists State; inputs \bullet State_a \land Operation.$$

Example. The precondition of the operation $Adjust$ in the $Light$ package is:

$$\mathrm{pre}(Adjust) \mathrel{\widehat{=}} \exists dim' : 0..100; on' : \mathbb{B} \mid dim' > 0 \iff on' = true \bullet on = true \land dim' = dim?$$

Informally, it reads as "there exists a poststate (an instance of the $Light$ schema) such that the operation $Adjust$ can be engaged." Given a state where $dim > 0 \land on = true$, the postcondition of the operation $Adjust$ is:

$$\mathrm{post}(Adjust, dim > 0 \land on = true) \mathrel{\widehat{=}} \exists dim : 0..100; on : \mathbb{B}; dim? : 0..100 \mid dim > 0 \iff on = true \bullet (dim > 0 \land on = true) \land (on = true \land dim' = dim?).$$

The operation schemas of a package thus form a named collection of relations which determine a (finite or infinite) transition system in which an operation may fire exactly when its Z precondition is satisfied. The semantic model for the system consists of all the sequences of operations/events which can be performed by the objects.

Z is not intended for description of nonfunctional properties, such as usability, performance, size, and reliability.

1. In this paper, states and predicates are used interchangeably.

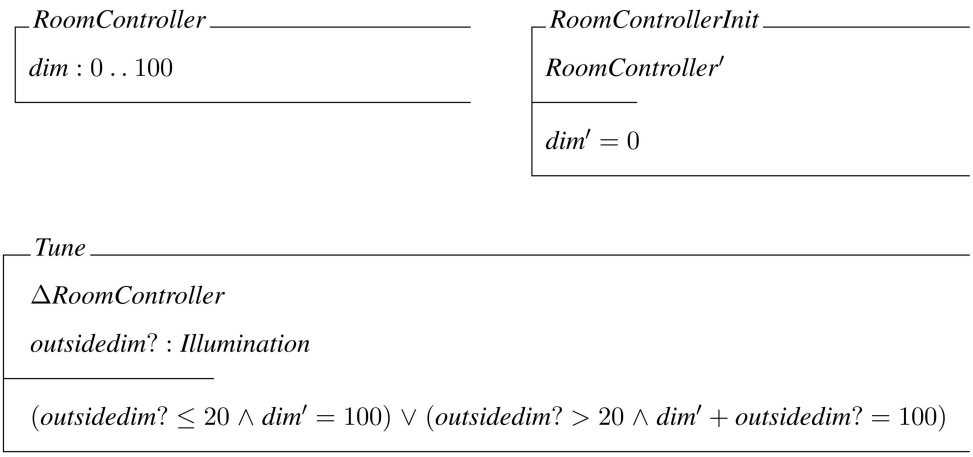

Fig. 1. Z package of the *room controller* schemas.

Neither is it intended for timed or concurrent behaviors. There are other formal methods that are well suited for these purposes. We may use these methods in combination with Z to relate state and state-change information to complementary aspects of the systems.

2.2 Live Sequence Charts

The notion of MSC [46] is widely used to describe scenarios of interaction between processes or objects. However, MSC suffers from the rather weak partial ordering semantics that make it incapable of capturing many kinds of behavioral requirements [6], [7], [9]. LSC was introduced in [9] to overcome the shortcomings of MSC by adding liveness. LSC extends MSC with various constructs to distinguish scenarios that must happen from ones that may happen, conditions that must be fulfilled from ones that may be fulfilled, etc.

There are two kinds of charts in LSC. Existential charts are mainly used to describe possible scenarios of a system in the early stages of system development, i.e., the same role played by the classic MSC. In later stages, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. A universal chart is typically preceded by a prechart, which serves as the activation condition for executing the main chart. Whenever a communication sequence matches a prechart, the system must proceed as specified by the main chart. Due to precharts, a system run may activate a universal chart more than once and some of the activations might overlap [32]. In this work, we assume that an LSC specification consists of a set of universal charts, whereas existential charts are used to specify test cases. An implementation of the LSC model should only exhibit behaviors allowed by the universal charts and should always be able to exhibit at least one of the behaviors specified by any of the existential charts.

Each universal chart is associated with a set of visible events. Only visible events are constrained by the chart. An optional set of forbidden events is associated with a chart. A chart typically consists of multiple instances, which are represented as vertical lines. Along with each line, there are a finite number of locations. A location carries the temperature annotation for progress within an instance. Locations may be either cold or hot. A hot location means that the system has to move beyond, whereas the system may stay at a cold location forever. Globally, a system run is accepted by the chart if no object is stuck at a hot location. Messages and conditions are also labeled. A hot message must be received, whereas a cold one may get lost. A hot condition must be met, whereas a cold condition terminates the chart or subchart if it is evaluated to false. However, we remark that the ability to specify hot and cold messages, i.e., whether a message is required to be received or may get lost, is redundant because of the facility for describing hot and cold locations. Essentially, the temperature of the locations takes precedence over the temperature of messages, so whether or not the message is received is determined entirely by the temperature of the message input. This questionable feature of LSC is recognized by Harel and Marelly, who list the possible cases and conclude that the temperature of the message has no semantic meaning [17]. Thus, in the following discussion, the temperature of all messages is discarded.

Example. Fig. 2 shows two typical scenarios of the LCS. 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. Other scenarios of the LCS include the charts in Fig. 3, where the occupant may turn the light on/off by pushing the button or the system regularly adjusts the illumination of the light. The LCS in this paper is a simplified version of the one presented in [12].

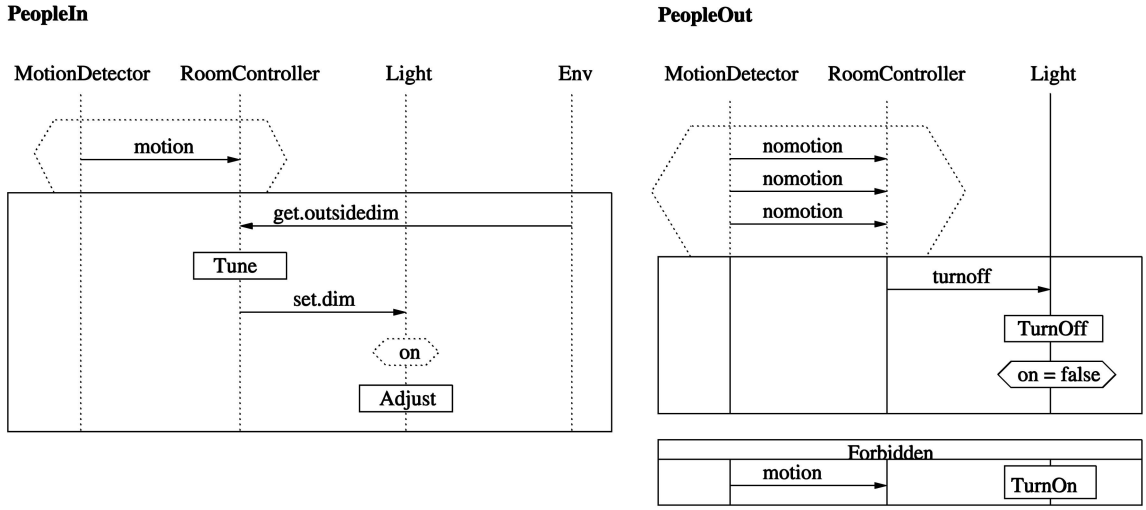

Fig. 2. Scenarios of the LCS: $\mathbf{PeopleIn}$ and $\mathbf{PeopleOut}$.

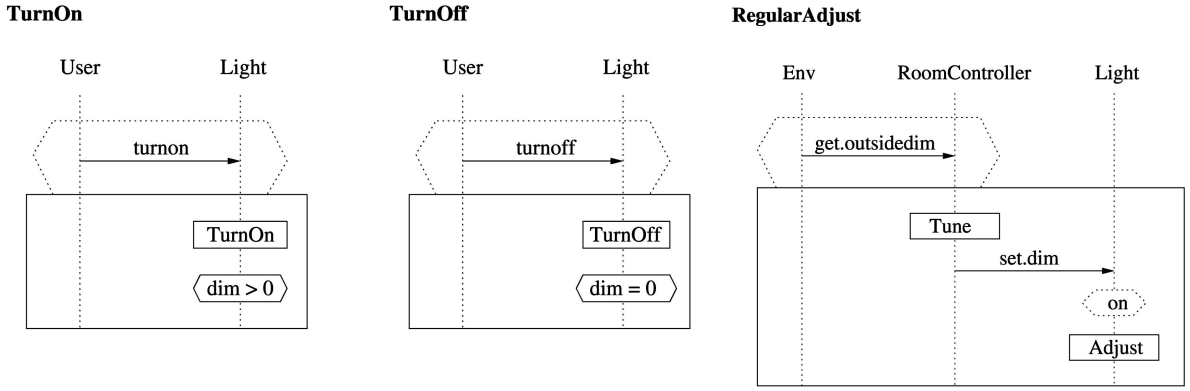

Fig. 3. Scenarios of the LCS: $\mathbf{TurnOn}$ and $\mathbf{TurnOff}$ and $\mathbf{RegularAdjust}$.

LSC also supports advanced features like hierarchy, symbolic instances and messages, etc. For a detailed introduction on a complete list of features of LSC, refer to [17]. Our 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, etc. [17]. We also assume that all conditions are distributed because we are only interested in local control strategies and we believe shared conditions are a problematic feature of LSC. A condition is a Boolean expression over the visible variables of the chart. Therefore, some form of global variables is presupposed. This does not match the reality of distributed systems. Objects in distributed systems have their own state space (local variables) and all communication between objects would be via messages. This assumption makes our localized synthesis possible. We do support distributed conditions that can be rewritten as a set of local conditions with proper synchronization.

2.3 An Integration of Z and LSC

We remark that 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 [15], [5]. 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 we think 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, we propose a simple yet effective integration of LSC and Z which allows specification of systems with not only complicated interactive behaviors, but also complex data structures.

We require that a combined system specification consist 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. 2 and Fig. 3 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.

3 Synthesis of a Distributed Object System

In this section, we explain 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 [49] 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. We discuss our construction using the notion of finite state machine.

3.1 Terminology

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

$$M \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I),$$

where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of accepting states, Σ is the alphabet, and $T : S \times \Sigma \rightarrow S$ 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, $\langle s_1, e_1, s_2, e_2, \cdots, s_i, e_i, s_{i+1}, \cdots \rangle$, is an alternating sequence of states and events subject to the following: $\forall i : \mathbb{N} | i \geq 1 \bullet (s_i, e_i, s_{i+1}) \in T$ and $s_1 \in S_0$. 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.

Definition 2. *Given two state machines $M_i \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$, where $i \in \{1, 2\}$, a state machine $M \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$ is the product, written as $M_1 \| M_2$ if $M.S \mathrel{\widehat{=}} M_1.S \times M_2.S$ and $M.S_0 \mathrel{\widehat{=}} M_1.S_0 \times M_2.S_0$ and $M.F \mathrel{\widehat{=}} M_1.F \times M_2.F$ and $M.\Sigma \mathrel{\widehat{=}} M_1.\Sigma \cup M_2.\Sigma$ and $M.I \mathrel{\widehat{=}} \{((s_1, s_2) \mapsto M_1.I(s_1) \wedge M_2.I(s_2))\}$ and T is the least subset of $S \times \Sigma \times S$ satisfying the following:*

$$\begin{aligned}
&(s_1, s_2) \in M.S \wedge (s_1, e, s_1') \in M_1.T \wedge e \notin M_2.\Sigma \\
&\Rightarrow ((s_1, s_2), e, (s_1', s_2)) \in M.T \\
&(s_1, s_2) \in M.S \wedge (s_2, e, s_2') \in M_2.T \wedge e \notin M_1.\Sigma \\
&\Rightarrow ((s_1, s_2), e, (s_1, s_2')) \in M.T \\
&(s_1, s_2) \in M.S \wedge (s_1, e, s_1') \in M_1.T \wedge (s_2, e, s_2') \in M_2.T \\
&\Rightarrow ((s_1, s_2), e, (s_1', s_2')) \in M.T.
\end{aligned}$$

The parallel composition is symmetric and associative. The indexed product of multiple state machines is written as $\|_i M_i$, where i is the index.

Definition 3. *Let $M_i \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$, where $i \in \{1, 2\}$, be two state machines. A total relation $\mathcal{R} : M_1.S \rightarrow M_2.S$ is a fair simulation from M_1 to M_2 if it satisfies the following:*

$$\begin{aligned}
&D_1 : \forall s : M_1.S_0 \bullet \mathcal{R}(s) \in M_2.S_0 \\
&D_2 : \forall (s_1, e, s_2) \in M_1.T; s_1' : M_2.S | \mathcal{R}(s_1) = s_1' \\
&\qquad \bullet \exists s_2' : M_2.S \bullet (s_1', e, s_2') \in M_2.T \wedge \mathcal{R}(s_2) = s_2' \\
&D_3 : \forall s : M_1.F \bullet \mathcal{R}(s) \in M_2.F.
\end{aligned}$$

D_1 states that there is an initial state in M_2 corresponding to every initial state in M_1. D_2 states if M_1 can engage an event at a certain state, M_2 should be able to simulate the transition at the corresponding state. D_3 guarantees that all final states in M_1 are simulated in M_2. A similar definition appeared in [11] and later development can be found in [19]. If there is a fair simulation relation from M_1 to M_2, then M_2 fair trace contains M_1, i.e., it is possible to generate by M_2 every fair sequence of operations that can be generated by M_1. The notion of fair trace-containment is robust with respect to LTL [19].

3.2 Synthesizing Local State Machines

We start with constructing a state machine for each instance in a single chart. Given a basic chart m (a main chart or a subchart of a main chart without hierarchy), let $M_m^i \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$ 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

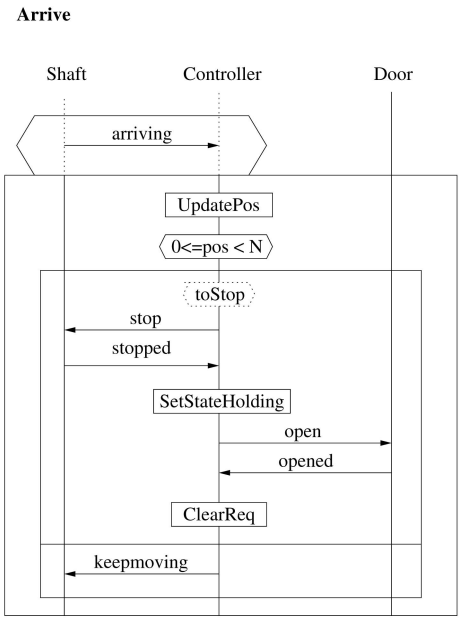

Fig. 4. A scenario of the Lift Control System.

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.

Example. Fig. 4 is a universal chart containing a conditional branch. It is part of the LSC specification for a lift control system. Whenever the lift approaches the next floor, the $shaft$ sends a message $arriving$ to the $controller$. The $controller$ refreshes its knowledge of the current level by updating its local variable pos. A hot condition stating that the value of pos must be within its range is asserted. The $controller$ decides whether to stop at the next floor. If the condition $toStop$ is true, i.e., the next level is requested internally or requested externally with the right direction, the $shaft$ stops and the $door$ is opened and the respective request is cleared. Otherwise, the lift continues traveling in the same direction.

The upper state machine in Fig. 5 is synthesized, for instance, $Shaft$ from the main chart. Events $Arrive.x.main$ and $Arrive.x.sub1$ are barriers used to synchronize the entering or exiting of the main chart and the subchart among all participating instances. Variable x is an identifier which distinguishes different activations of the same chart. Only participating instances in the same activation of the chart are synchronized. Whenever the chart completes (reaching the filled circle), the system behaves freely, i.e., all visible events can be engaged. Fig. 5 also shows the state machine synthesized for instance $Controller$. For readability, we use a transition labeled with * to state that all visible events are allowed to engage. Only transitions labeled with visible events are constructed since transitions concerning invisible events are free to occur by Definition 2. The hot condition is labeled with the state right after the local action $UpdatePos$. After entering the subchart, two states are reached, one labeled with the condition $toStop$ and the other labeled with the negation of it. The conditional

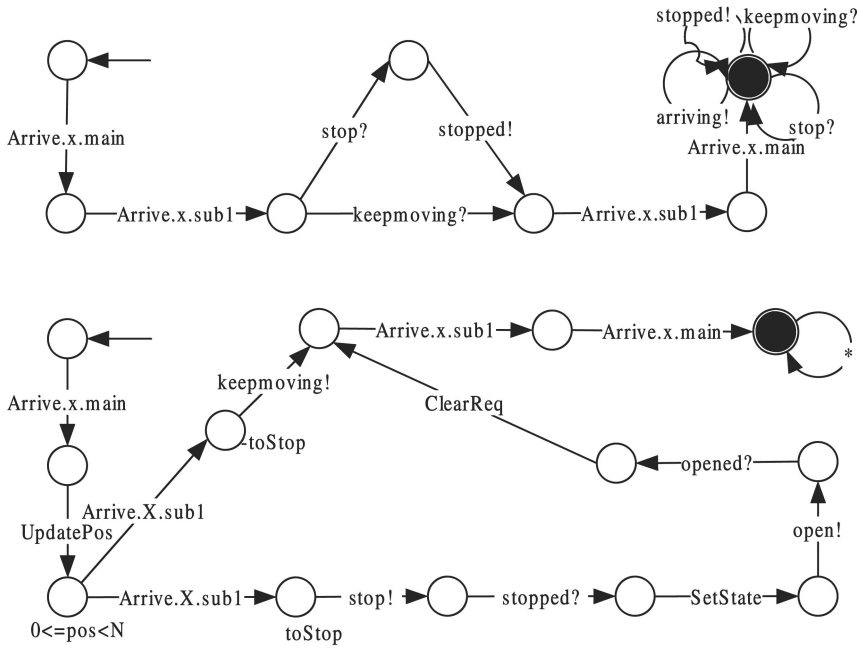

Fig. 5. State machine for $Shaft$ and $Controller$ in scenario arrive.

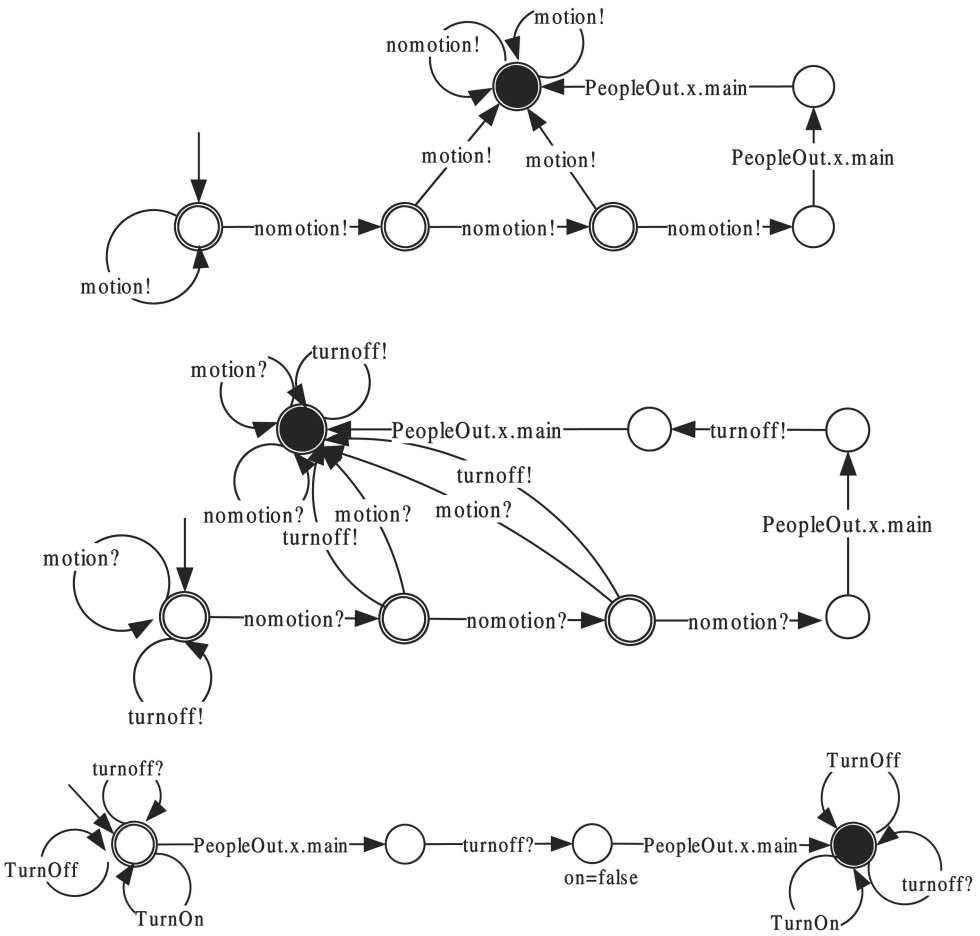

Fig. 6. State machines for instances in scenario **PeopleOut**.

branch is effectively flattened. In general, the state machine for hierarchical charts can be constructed from the state machines synthesized from the subcharts recursively. We skip the detail on how to flatten hierarchal charts because it is quite standard.

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 Σ_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. $\Sigma_u \widehat{=} \cup \Sigma_u^i$ is the visible events of chart u. Let $M_p^i \widehat{=} (S, S_0, F, \Sigma, T, I)$ be the state machine synthesized from instance i in prechart p. There is a transition (s_1, e, s_2) in $M_p^i.T$ 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. In addition, a transition (s_1, e', s_{max}) is constructed for every event e' in Σ_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. Fig. 6 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

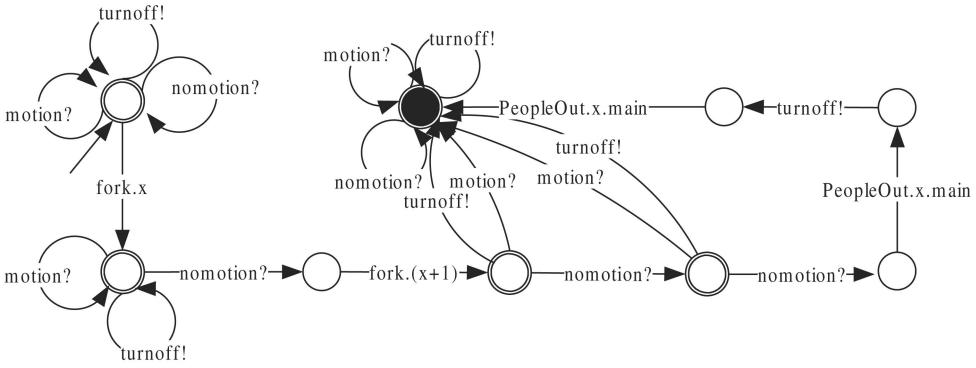

Fig. 7. The state machines synthesized for Instance $RoomController$ in Chart PeopleOut.

because the state machine will not constrain the system execution before entering the main chart.

Parallel composition of the state machines, e.g., the ones in Fig. 6, 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 Fig. 2, yet it is allowed by the state machine for $RoomController$ in Fig. 6. 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. In [4], Bontemps and Schobbens have shown that every LSC has an equivalent deterministic Büchi automaton that contains at most exponentially more states than there are locations in the LSC. 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.

Example. Fig. 7 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 Fig. 8. We identify the very last state (the one composed by three filled state) with the initial state so as to allow nonoverlapping activations. We remark that the final state machine can be further reduced using standard technique like bisimulation reduction [27], etc. For instance, all states labeled with event $fork$ are removed as they are irrelevant to system behaviors.

3.3 Discussion

We remark that the product of the state machines for all instances in the chart, $\|_i M_u^i$, refines the chart, i.e., all accepting runs of the state machine satisfy the chart. An immediate consequence is that the product of the state machines for all the universal charts, $\|_u \|_i M_u^i$, refines the LSC specification, i.e., only behaviors satisfying all the universal charts are allowed. Because the parallel composition operator is symmetric and associative, the following

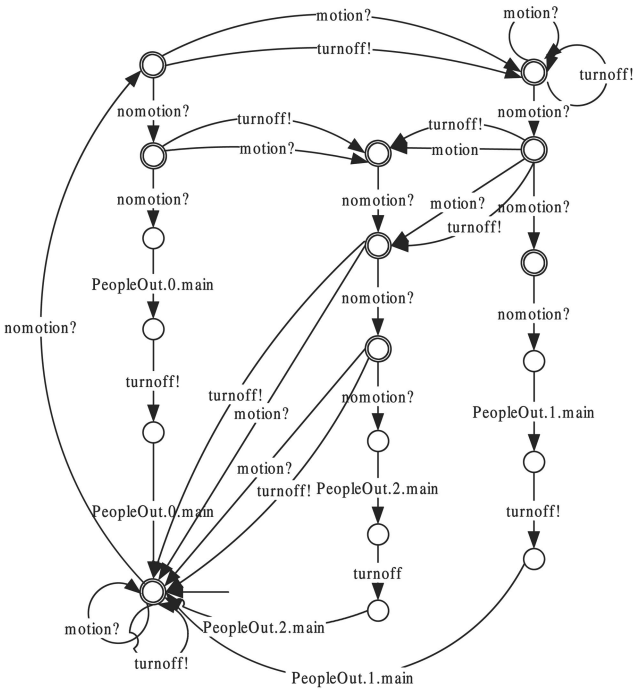

Fig. 8. The final state machine synthesized for instance $RoomController$ in chart PeopleOut.

rule is established. Let M^i_{LSC} denote the local behaviors of an object i.

$$\|_u\|_i M^i_u \widehat{=} \|_i\|_u M^i_u \widehat{=} \|_i M^i_{LSC}.$$

Due to the above transformation, the local behaviors of an object are determined without constructing the global state machine. For example, the behaviors of the $RoomController$ are captured by the product of the state machines synthesized from all the universal charts.

We skip the formal soundness proof of the synthesis. In our technical report [40], which extends our earlier work in [41], we formally defined a trace-based denotational semantics of LSC. We then developed a sound interpretation of LSC in the classic notion of CSP [20]. By transforming CSP interpretation of the LSC model using CSP's algebraic laws, the local behaviors of each object are effectively grouped together as a set of distributed processes. A bisimulate relationship between the synthesized state machine in this work and the transition system interpretation of the distributed processes would prove the soundness of our synthesis. Alternatively, we may define a similar set of algebraic laws in terms of state machines and prove the soundness directly.

So far, our synthesis does not distinguish environmental objects from system objects. In other words, we handle only closed systems but not open systems, i.e., systems that interact with their environment constantly. The synthesis problem for closed systems is often referred to as satisfiability, i.e., whether the language of a specification is nonempty or, equivalently, if considering the environment as part of the system, if there is a benevolent environment in which some implementation can be deployed in order to fulfill the specification. Synthesis for open systems asks whether there is an implementation that can be deployed in any malevolent environment. In the literature, synthesis for open systems has long been recognized as a hard problem. And, it is even harder to synthesize distributed implementations without constructing the global state machine, i.e., undecidable in almost all interesting settings [43], [30], [35]. Thus, we take a lightweight approach to tackling the problem. The intuition is that, when system engineers design systems, implicit assumptions on the environment are often made (enforced later by blocking the user-interface at certain time, using a queue to delay the arrival of the environmental event, etc.). Instead of synthesizing an implementation that works in any environment, we synthesize one that works in the intended environment. In other words, we deal with a restatement of the synthesis for open systems: Given a (partial) modeling of the environment and an LSC specification, build a distributed object system such that, for every refinement of the environment, the object system satisfies the LSC specification.

The objects are marked as either environmental objects or system objects. Consequently, events are also marked as either environmental events or system events. An event is an environmental event if and only if it is a local action of an environmental object or a communication event that requires the participation of an environmental object. The system designer is asked to offer a modeling of the intended environment, preferably using universal charts, which capture all implicit assumptions on the environment. Local state machines for the environmental objects are synthesized in the same way as system objects. We then verify that the synthesized state machine for the environment (parallel composition of all state machines for the environment objects) simulates the user-supplied modeling of the environment. From another point of view, the state machines synthesized for environmental objects are indeed system processes which monitor the interaction between the environment and the system as well as trigger the appropriate special events at the proper time. The refinement relationship therefore ensures that no interaction is missed.

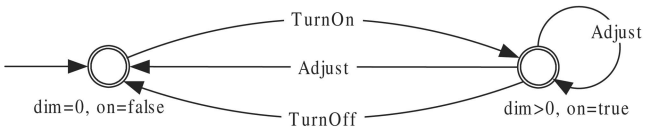

Fig. 9. Abstract interpretation of the Z-package of $Light$.

4 Refinement of the Distributed Object System

In our combined specification, local actions could be complicated computation constrained by pre/postcondition. A sound design will make sure that a local action is only invoked that satisfies its precondition, a hot condition will be satisfied in all circumstances, etc. However, it is difficult to tell if some assertion is true after certain (or a series of) local computations because the state space of a Z specification may often be infinite. The problem is further complicated as Z operation schemas may take inputs from the environment, which cannot be controlled by the system.

Our remedy is predicate abstraction thanks to recent development on software model checking [1]. Predicate abstraction allows us to interpret and then restrict the behaviors of an object based on the abstract view of the data variables. For instance, Fig. 9 presents an abstract interpretation of the $Light$ package in which only whether the light is on or off is of interest. From such an abstract graph, it is easy to tell that, after operation $TurnOn$, the variable dim must be positive, etc. Predicate abstraction is essential for our synthesis since an implementable control structure may only contain a finite number of control states.

We present a systematic way of calculating the predicate abstraction of a Z package. An abstract machine where an operation can be engaged only at states where its precondition is satisfied and can reach all states satisfying its postcondition is constructed. That is, we construct an abstract state machine for the underlying data structure. The abstract machine is then used to refine the distributed object system synthesized from the LSC model on an object basis, i.e., the control flow, by removing invocation of local actions that might violate their preconditions, restricting the operation by requiring it must result in states satisfying the hot conditions, etc.

4.1 Predicate Abstraction of Z Packages

Given a finite set of predicates P (in terms of the state variables) for abstracting a Z package, the set of abstract

states, denoted as S_a, contains the conjunction of a subset of the predicates in P and their negations:

$$S_a \widehat{=} \{x \mid \exists X \subseteq P \bullet x = \wedge(X \cup \{\neg e | e \in P \setminus X\})\}.$$

An abstract state groups all possible valuations of the state variables which satisfy the predicate x. For instance, the state $dim > 0$ *and* $on = true$ in Fig. 9 groups all instances of schema $Light$ where dim is positive (1-100) and on is $true$. In order to guarantee the correctness of the synthesized design, we require that the set of predicates for abstraction include all conditions in the universal charts (as well as the predicate in the initial schema for simplicity).

Given an operation, it is necessary to find out the abstract states where the operation can be invoked without violating its precondition and the abstract states which can be reached via engaging it at a state satisfying its precondition. We define a function $\mathcal{W}$ to calculate the weakest formula over P, which implies a predicate p, i.e., $\mathcal{W}(p) = \{x \in S_a | x \Rightarrow p\}$. The motivation is that if p is the precondition of an operation, then $\mathcal{W}(p)$ is the set of all abstract states where the operation can be safely invoked. Similarly, we define a function $\mathcal{S}$ to compute all abstract states where a given predicate might be true, i.e., $\mathcal{S}(p) = S_a \setminus \{x \in S_a | x \Rightarrow \neg p\}$. If p is the postcondition of an operation at a state, then $\mathcal{S}(p)$ is the set of abstract states that may be reached via invoking the operation.

A finite state abstraction of a Z package is built by abstracting both its initial schema and operation schemas. Because only sound designs are of interest, a local action will be invoked only when we are certain no assertions will be violated. It is achieved by replacing the precondition of each operation by $\mathcal{W}(\mathrm{pre}(Operation))$ and replacing the postcondition of the operation engaged at an abstract state s_a in $\mathcal{W}(\mathrm{pre}(Operation))$ by $\mathcal{S}(\mathrm{post}(Operation, s_a))$. By replacing the precondition with a more restrictive one, we make sure no precondition will be violated. By replacing the postcondition with a less restrictive one, we make sure that no hot conditions will be violated in all circumstance. Our abstraction is consistent with the notion of Z data refinement, i.e., weakening the precondition or strengthening the postcondition. All abstract states are accepting because the system may idle infinitely long before engaging an enabled operation according to the Z semantics [49].

Definition 4. *Given a set of predicate P, $M_Z^i \widehat{=} (S, S_0, F, \Sigma, T, I)$ is an abstraction of the Z package associated with object i only if $S \widehat{=} S_a$ and $S_0 \widehat{=} \mathcal{W}(initialcondition)$ and $F \widehat{=} S$ and Σ is the set of operation schemas in the package and I labels a state with itself and*

$$T \widehat{=} \{(s_1, e, s_2) : S \times \Sigma \times S | s_1 \in \mathcal{W}(\mathrm{pre}(e)) \wedge s_2 \in \mathcal{S}(\mathrm{post}(e, s_1))\}.$$

Example. Assume the set of predicates for abstracting the $Light$ package is $\{dim = 0, on = false, dim > 0\}$, the set of abstract states thus contains two states:

$$S_a \widehat{=} \{dim = 0 \wedge on = false, dim > 0 \wedge on = true\}.$$

The abstract initial state is exactly the state where $dim = 0 \wedge on = false$. Operation $Adjust$ is abstracted by computing the following:

$$\begin{aligned}
&\mathcal{W}(\mathrm{pre}(Adjust)) \\
&\quad \widehat{=} \mathcal{W}(\exists dim' : Illumination; on' : \mathbb{B} | dim' > 0 \\
&\qquad \iff on' = true \bullet on = true \wedge dim' = dim?) \\
&\quad \widehat{=} \{dim > 0 \wedge on = true\} \\
&\mathcal{S}(\mathrm{post}(Adjust, dim > 0 \wedge on = true)) \\
&\quad \widehat{=} \mathcal{S}(\exists dim, dim? : Illumination; on : \mathbb{B} | dim > 0 \iff \\
&\qquad on = true \bullet \\
&\qquad dim > 0 \wedge on = true \wedge on = true \wedge dim' = dim?) \\
&\quad \widehat{=} \{dim' = 0 \wedge on' = false, dim' > 0 \wedge on' = true\}.
\end{aligned}$$

Thus, the abstract operation $Adjust$ is enabled only at the abstract state where on is true, from which both abstract states can be reached by engaging $Adjust$ once. We skip the abstraction of the other operations in $Light$ package. Fig. 8 shows the resultant state machine.

However, both function $\mathcal{W}$ and $\mathcal{S}$ in our context are undecidable, i.e., we may not be able to tell if a predicate is true at a state due to the limited power of proving. The remedy is to compute approximations of the functions. The key idea is that an approximation of the function $\mathcal{W}$ will contain at most the set of abstract states in $\mathcal{W}(p)$, whereas the approximation of the function $\mathcal{S}$ will contain at least states in $\mathcal{S}(p)$. This way, our abstraction remains robust with the Z data refinement. In our prototype, we make use of the theorem prover PVS [33] to compute such approximations in order to construct an abstract state machine by paying a reasonable price.

4.2 Pruning

After constructing the abstract state machine from the Z package, the product of M_{LSC}^i and M_Z^i is computed. By removing states labeled with $false$, we guarantee that no preconditions or hot conditions will be violated. However, the problem is complicated by the uncontrollability of the environment because removing the states may put restrictions on the inputs from the environment. Dealing with the uncontrollability of the environment is essential for open systems. Informally, it requires that the design should be able to function correctly, regardless of the environment input. For instance, if we allow the user to adjust the illumination by setting it to a certain value, captured by the universal chart in Fig. 10. It requires that, after operation $Adjust$, $dim > 0$ must hold. Intuitively, we know that this hot condition may not be satisfied because the user may set the dim to 0 and, hence, accidentally turn off the light (due to the state invariant). In the following, we show how such behaviors can be detected and pruned systematically. Another important property for open systems is nonblocking, i.e., the design should not introduce any fresh deadlock. Both requirements have been discussed in works of control theory [37], [30]. Our solution is a pruning algorithm which determines whether there is a satisfying design and synthesizes one if possible by refining the product state machine.

We partition the actions enabled at a state into two sets, controllable actions and uncontrollable actions. An action e is uncontrollable at a state s_1 if its postcondition $\mathrm{post}(e, s_1)$ depends on some environmental input. An action at a state is controllable if it is not uncontrollable. An action may be

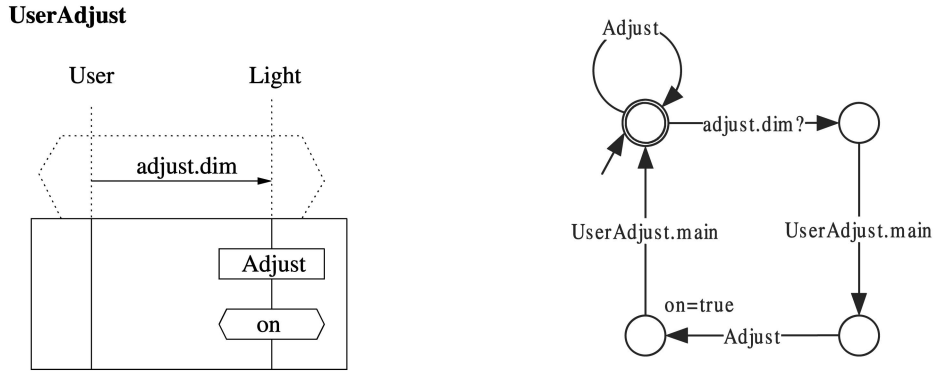

Fig. 10. Scenario $\mathbf{UserAdjust}$ and the state machine for the object $Light$.

controllable at a state but uncontrollable at another one. This is different from previous works on supervisory control [37], [36], where all occurrences of an uncontrollable event are uncontrollable.

Definition 5. *Let $M^i \widehat{=} (S, S_0, F, \Sigma, T, I)$ be the product of M^i_{LSC} and M^i_Z. A state machine $M^i_D \widehat{=} (S, S_0, F, \Sigma, T, I)$ is a design if it satisfies the following conditions:*

$\mathbf{A_1}$: There is a fair simulation relation $\mathcal{R}$ from M^i_D to M^i.
$\mathbf{A_2}$:

$$\forall (s^1_{LSC}, s^1_Z) : M^i.S; s^2_Z : M^i_Z.S; e : M^i_Z.\Sigma \bullet (s^1_Z, e, s^2_Z) \in M^i_Z.T$$
$$\Rightarrow \exists (s^3_{LSC}, s^3_Z) : M^i.S; e' : M^i_Z.\Sigma \bullet (\mathcal{R}((s^1_{LSC}, s^1_Z)), e',$$
$$\mathcal{R}((s^3_{LSC}, s^3_Z))) \in M^i_D.T.$$

$\mathbf{A_3}$:

$$\forall (s^1_{LSC}, s^1_Z), (s^2_{LSC}, s^2_Z) : M^i.S; s^3_Z : M^i_Z.S; e : M^i_Z.\Sigma \bullet$$
$$(\mathcal{R}((s^1_{LSC}, s^1_Z)), e, \mathcal{R}((s^2_{LSC}, s^2_Z))) \in M^i_D.T \wedge (s^1_Z, e, s^3_Z) \in M^i_Z.T$$
$$\wedge\, e \text{ is uncontrollable at } s^1_Z$$
$$\Rightarrow (\exists s^3_{LSC} : M^i_{LSC}.S \bullet (\mathcal{R}((s^1_{LSC}, s^1_Z)), e,$$
$$\mathcal{R}((s^3_{LSC}, s^3_Z))) \in M^i_D.T).$$

Informally, $\mathbf{A_1}$ guarantees that the design satisfies both the LSC and Z specification since only behaviors allowed by the product are permitted. $\mathbf{A_2}, \mathbf{A_3}$ guarantee the additional properties are satisfied. $\mathbf{A_2}$ states if a state is not a deadlock state, it will not be a deadlock state in the design. $\mathbf{A_3}$ states if a target state is reachable from a source state by engaging an uncontrollable operation, the target state will also be reachable in the design. Both requirements $\mathbf{A_2}, \mathbf{A_3}$ concern only local actions and, hence, communication events remain.

In the following, we present the pruning algorithm that prunes states and transitions from the product $M^i_{LSC} \| M^i_Z$ recursively so as to construct a minimally restrictive (if possible) design. For every reachable state s, we will check if it satisfies requirement $\mathbf{A_3}$. If it does not, i.e., there is an uncontrollable action e at s whose poststates have been partially removed, all transitions from s labeled with e are pruned at once. The intuition is that, if allowing this operation at the state may result in trouble (given certain environment inputs), we do not allow the operation at the state at all. We also check if any state is a fresh deadlock state. If it is, the state is pruned along with all its incoming and outgoing transitions. Pruning transitions may create deadlock states. A state may violate $\mathbf{A_3}$ after some of its immediate successor states get pruned because its outgoing transitions are pruned too. Therefore, the pruning must be applied recursively.

Our algorithm is presented in Fig. 11. Lines 1 to 4 declare the variables. Variable $Successor$ is the set of the initial states, which can be viewed as immediate successor states of an imaginary single "initial" state. Variable $Path$ will contain the states in the path from an initial state to the current state (inclusive). It is initially empty. The state machines $Product$, Raw represent M^i and M^i_Z, respectively. Line 5 invokes our recursive procedure of pruning. All four variables are passed as parameters. In the procedure `Pruning`, the first line declares a local variable $Done$ as a local holder of processed states out of $Successor$. Line 3 checks if all states in $Successor$ have been processed and returns true if every state in $Successor$ is also in either $Path$ or $Done$. If a state is in $Path$, it is a common ancestor of all states in $Successor$. A state out of $Successor$ but not $Path$ or $Done$ is chosen at line 4. At line 7, we have another loop. The intuition is that the state will be checked repeatedly until none of its decedent state is pruned. Lines 8-12 verify if the state satisfies $\mathbf{A_3}$. The function $A3(Product, Raw, e, s)$ returns true if all possible environment inputs to operation e at s are handled properly. Lines 13-16 state that if the state is a fresh deadlock state, then the pruning backtracks by returning false, i.e., the parent of the state will be checked again because one of its child states has been pruned. If the state satisfies both $\mathbf{A_2}$ and $\mathbf{A_3}$, its child states are retrieved (line 17). Line 20 is a recursive method call. If the recursive call returns false, it means some child state has been pruned and, thus, the state has to be reexamined. Otherwise, we are done with the state.

Line 6 in procedure `prune` checks if there is a design by removing the states out of reach and the states leading to no accepting state. There is a design (the pruned state machine) if and only if the pruned state machine has at least one initial state and one reachable accepting state. $\mathbf{A_1}$ is obviously satisfied as the identity relation is a fair simulation relation from the pruned state machine to the product. The correctness of the algorithm is an immediate consequence of the fact that a state is not pruned if and only if it satisfies both requirements and all reachable states from

```
void Prune () {
1.    let Successor := the set of initial states;
2.    let Path := an empty set;
3.    let Product := the product state machine;
4.    let Raw := the abstract state machine;
5.    Pruning(Successor, Path, Product, Raw);
6.    ExistDesign(Product);
}

boolean Pruning (Successor, Path, Product, Raw) {
1.    let Done := an empty set;
2.    while (true)
3.       if (Successor = Path union Done) return true;
4.       let s := a state in Successor but not in Path or Done;
5.       Add s into Done;
6.       let childStatePruned := false;
7.       while (!childStatePruned)
8.           for all uncontrollable actions e at s
9.               if (!A3(Product, Raw, e, s))
10.                   prune all transitions labelled with e from Product;
11.               endif
12.           endfor
13.           if (!A2(Product, Raw, s))
14.               prune s from Product;
15.               return false;
16.           endif
17.           let Children := immediate_successors(Product, s);
18.           if (Children is not empty)
19.               Add s to Path;
20.               if (!Prune (Children, Path, Product, Raw))
21.                   childStatePruned := true;
22.               endif
23.           endif
24.       endwhile
25.   endwhile
}
```

Fig. 11. Pruning Algorithm.

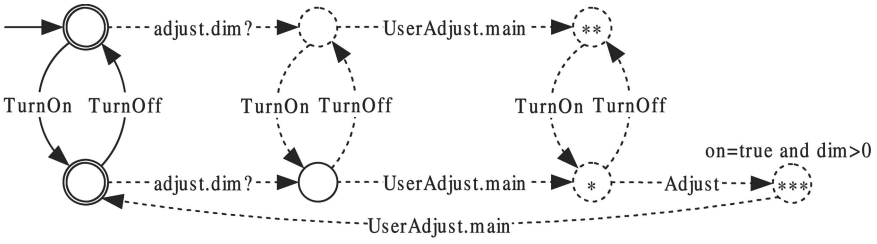

Fig. 12. Product of state machine in Fig. 9 and Fig. 10.

it are not pruned. The algorithm converges because the states and transitions are finite and it backtracks only when a state is pruned.

Example. Fig. 10 presents the state machine for instance $Light$ from scenario UserAdjust assuming there are no overlapping activations of the chart for simplicity. The product of the state machines in Fig. 9 and Fig. 10 is shown in Fig. 12 (where one state labeled with $false$ has been removed). Applying the pruning algorithm to the product removes all but the two accepting states and the transitions between them. The * state is removed because $Adjust$ is an uncontrollable event at the state and the state labeled with $on = false \wedge dim = 0$ is not reachable from the * state by engaging $Adjust$ while it does in the abstract state machine. Thus, line 10 of the algorithm applies so that the transitions labeled with $Adjust$ are removed. The *** state is removed because it is not reachable any more. The ** state is removed because it becomes a fresh deadlock state and, thus, line 14 of the algorithm applies so that the state is pruned. The rest are removed because they cannot reach an accepting state. The resulting state machine is a valid design for closed systems because there are initial and accepting states. Intuitively, the resulting state machine guarantees that the chart **UserAdjust** is satisfied by requiring that it is never activated. However, for open systems, $user$ is

considered as part of the environment and, therefore, there is no way to prevent users from activating the chart through sending message $adjust.dim?$. In our approach, the synthesized modeling of the $User$ failed to simulate of the default modeling (where users can initiate any communication at any time) and, thus, there is no design satisfying the chart.

In the following, we briefly discuss the soundness of the techniques used in this section. In Section 3.3, we have shown that the state machines constructed are consistent with the LSC model treating local actions as abstract events. We now argue that the refined state machines satisfy both the LSC model and the Z model. First of all, by Definition 4, local actions can only be engaged within their (strengthened) domain. Engaging in a local action may appear to reach more states than it could because the postcondition is weakened. This causes no problem because local actions will be replaced by concrete implementations which satisfy their pre/postcondition. Thus, there may be infeasible pathes in the synthesized implementation. The point is that, using the weakened postcondition, we can detect a possible violation of hot conditions early in the synthesis process (instead of at runtime). The product of the state machines synthesized from the LSC model of an object and the abstract state machine of the Z package thus satisfies both the LSC model and the data requirements. During the pruning process, transitions and states are pruned. It is easy to verify that the pruned state machine is fairly simulated by the original one ($\mathbf{A_1}$ in Definition 5). Fair simulation implies fair trace containment. Thus, the pruned state machine is consistent with the specification.

5 Automation

5.1 Prototype

We have implemented a prototype to experiment with our method with standard case studies. The input to our experimental tool is an XML representation of the Z model and an XML representation of the LSC model. The XML representation for Z family languages is proposed in [42] and is actively developing. 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, we define the syntax of LSC using both BNF grammar and XML schema (available at [22]). Together with the XML schema, a parser and a synthesizer module are built based on an existing Java XML parser [13] to parse the LSC and construct the design for each object automatically.

The predicate abstraction is automated with the help of PVS. By default, the predicates for abstraction include exactly those conditions in the LSC model. An embedding of Z in PVS is developed. Lemmas are generated automatically from the Z specification for calculating the abstract initial schema and abstract pre/postcondition of each operation schema. PVS is invoked in batch mode to prove the lemmas automatically without user interaction. The rest of the steps, i.e., computing the product of the two state machines and removing states labeled with $false$ and pruning the product, etc., are fully automated by the synthesizer module.

Finally, executable codes are to be generated from the design. Code synthesis from state machines is reasonably straightforward. However, it is complicated in our case because of the state invariant. During the pruning process, nondeterministic choices may get pruned partially. In other words, the action's postcondition may be strengthened, which is not directly implementable in traditional programming languages like C or Java. Two different remedies have been explored. The first remedy is to guard each invocation of the action with a proper guard condition. For partially pruned nondeterministic choices, the transitions will be guarded with the weakest precondition that guarantees the reachability of the desired state. Let $\mathcal{WP}$ denote the weakest precondition operator introduced in [10]. Given an operation e and a source state s_a and a target state s_b, the weakest precondition for the state machine to reach s_b from s_a via operation e is (defined in [8]):

$$\mathcal{WP}(e, s_a, s_b) \mathrel{\widehat{=}} (\exists State' \bullet e) \wedge (\forall State' \bullet (e \wedge s_a) \Rightarrow s_b).$$

The first part of the condition guarantees the termination of the operation. The second part guarantees to reach the desired state. If the weakest precondition evaluates to $false$, there is no way to guarantee that the transition ends up with the desired state. This is normally due to internal nondeterminism, i.e., some information is missing in the model. After equipping transitions with proper guards, executable implementation can be synthesized straightforwardly with the implementation of each local action supplied by users. As long as the implementation of local actions conforms to its precondition/postcondition specification, our synthesized implementation is correct. However, because a reasonable guard condition must not involve any primed variables, computing the weakest precondition requires elimination of the primed variables. Variable elimination is, in general, undecidable. Therefore, this remedy can hardly be fully automated. The other remedy is to generate a set of proof obligations for nondeterministic transitions which are partially pruned. When the user provides an implementation of the operation, the proof obligations are verified (or tested) in addition to the pre/postcondition so as to make sure the operation satisfies the desired more restrictive postcondition at the system states.

5.2 Efficiency

Our approach is designed to handle complex systems. During the first step, we synthesize a distributed object system from the LSC model without constructing the global state machine, which saves us from state space explosion. We limit the number of overlapping activations of the same chart so as to further reduce the size of the local state machines. For instance, all universal charts except **PeopleOut** forbid overlapping activation in the LCS example. Nevertheless, computing the product of multiple state machines ($\|_i M_u^i$) explicitly is expensive, e.g., the state machine for instance $Light$ contains 760 states without any reduction. Therefore, we reuse existing CSP-based process oriented design patterns [47] to generate structural prototypes.

To handle systems with infinite data space, we adopt predicate abstraction to construct abstract view of system behaviors in terms of finite assertions. It is the most time-consuming operation in our method. In general, the size of the abstract state machine is exponential to the number of predicates for abstraction. However, we remark that our abstraction is based on one Z package at a time and there are unlikely to be a large number of conditions concerning an object. Our abstraction is developed to construct an abstract state graph by paying a reasonable price. In our prototype, sound approximations of the function $\mathcal{W}$ and $\mathcal{S}$ are used. To further speed up the abstraction, as well as to guarantee termination of the proof, a more loop-free proving strategy than *grind* (the highest-level command in PVS) is used to prove every single lemma in a limited amount of time. The data aspect of the LCS example is slightly trivial. As for reference, in a vending machine example with state variables with infinite domain and multiple operation schemas, 190 lemmas are generated altogether and all 105 provable lemmas are proven without user interaction correctly. We also experimented with the lift control system, which has arrays of variables (refer to [22]). In addition, a number of tricks have been used to further reduce the abstract state space, for instance, removing a false state by considering the corelation between the predicates and the state invariant before abstract. The operations that we perform over state machines are all polynomial time in terms of the number of states, for example, removing false states, removing unreachable states, removing states leading to no accepting state, etc. So is our pruning algorithm. Therefore, they are carried out reasonably fast.

6 Conclusion

In this work, we presented a systematic way of synthesizing designs from a combination of state-based modeling and interaction-based modeling, namely, Z and LSC. Our contribution is threefold. First, we proposed an intuitive integration of Z model and LSC model, which is capable of modeling systems with not only complicated data structures but also with complex interactive behaviors. Second, we developed a systematic way of synthesizing distributed finite state designs all the way from the specifications. Third, we developed an experimental tool to automate our method. Our method does suffer from being overrestrictive sometimes, i.e., the design implements a subset of the behaviors allowed by the specification. There are a few reasons why the designs may be overrestrictive. One of them is already mentioned in Section 3. Moreover, because our pruning applies on an object basis, valid designs requiring the cooperation of multiple system objects are not possible. For instance, inputs from other components in the system can be controllable if we consider the global state machine, e.g., the value of dim from $RoomController$ is actually never 0. However, we remark that there is no way to synthesize such designs without constructing the global state machine. The other reason is the limited power of proving. The effectiveness of the predicate abstraction, e.g., fewer spurious behaviors, depends on the proving power. Spurious behaviors may result in pruning valid designs. For instance, if an uncontrollable operation may reach only a safe state from a given state and yet the abstraction insists that it is possible to reach another undesired state, then the uncontrollable operation will be prohibited from happening at the given state. Nevertheless, we believe our method serves as a promising attempt to apply synthesis techniques to complicated systems and is general enough to be applied to other integrations of state-based and interaction-based modeling.

References

[1] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani, "Automatic Predicate Abstraction of C Programs," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* pp. 203-213, 2001.

[2] Y. Bontemps, "Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)," PhD thesis, Institut d'Informatique, University of Namur, Computer Science Dept., 2005.

[3] Y. Bontemps, P. Heymans, and P. Schobbens, "Lightweight Formal Methods for Scenario-Based Software Engineering," *Scenarios,* pp. 174-192, 2005.

[4] Y. Bontemps and P. Schobbens, "The Complexity of Live Sequence Charts," *Proc. Int'l Conf. Foundations of Software Science and Computational Structures,* pp. 364-378, 2005.

[5] Y. Bontemps, P. Schobbens, and C. Löding, "Synthesis of Open Reactive Systems from Scenario-Based Specifications," *Fundamenta Informaticae,* vol. 62, no. 2, pp. 139-169, 2004.

[6] M. Broy, "Message Sequence Charts in the Development Process—Role and Limitations," *Electronic Notes Theories of Computer Science,* vol. 65, no. 7, 2002.

[7] M. Broy, "A Semantic and Methodological Essence of Message Sequence Charts," *Science of Computer Programming,* vol. 54, nos. 2-3, pp. 213-256, 2005.

[8] A. Cavalcanti and J. Woodcock, "A Weakest Precondition Semantics for Z," *Computer,* vol. 31, no. 1, pp. 1-15, Jan. 1998.

[9] W. Damm and D. Harel, "LSCS: Breathing Life into Message Sequence Charts," *Formal Methods in System Design,* vol. 19, no. 1, pp. 45-80, 2001.

[10] E.W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, 1976.

[11] D.L. Dill, A.J. Hu, and H. Wong-Toi, "Checking for Language Inclusion Using Simulation Preorders," *Proc. Int'l Conf. Computer Aided Verification,* pp. 255-265, 1991.

[12] R.L. Feldmann, J. Munch, S. Queins, S. Vorwieger, and G. Zimmermann, "Baselining a Doman-Specific Software Development Process," Technical Report SFB501 TR-02/99, Univ. of Kaiserslautern, 1999.

[13] The Apache Software Foundation, "Xerces Java Parser v1.4.4," http://xml.apache.org/xerces-j/, 2001.

[14] D. Harel, "Statecharts: A Visual Formulation for Complex Systems," *Science of Computer Programming,* vol. 8, no. 3, pp. 231-274, 1987.

[15] D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Foundations of Computer Science,* vol. 13, pp. 5-51, 2002.

[16] D. Harel, H. Kugler, and A. Pnueli, "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements," *Proc. Formal Methods in Software and Systems Modeling,* pp. 309-324, 2005.

[17] D. Harel and R. Marelly, *Come, Let's Play—Scenario-Based Programming Using LSCs and Play-Engine.* Springer, 2003.

[18] O. Haugen and K. Stølen, "Stairs c Steps to Analyze Interactions with Refinement Semantics," *Proc. Int'l Conf. UML,* pp. 388-402, 2003.

[19] T.A. Henzinger, O. Kupferman, and S.K. Rajamani, "Fair Simulation," *Proc. Int'l Conf. Concurrency Theory,* pp. 273-287, 1997.

[20] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[21] ISO/IEC 13568:2002, "Information Technology-Z Formal Specification Notation-Syntax, Type System and Semantics," 2002.

[22] J. Sun and J.S. Dong, "Live Sequence Charts as CSP," http://www.comp.nus.edu.sg/sunj/LSC2CSP.html, 2005.

[23] C.B. Jones, *Systematic Software Development Using VDM.* Prentice-Hall, 1990.
[24] J. Klose and H. Wittke, "An Automata Based Interpretation of Live Sequence Charts," *Proc. Symp. Theoretical Aspects of Computer Science,* pp. 512-527, 2001.
[25] K. Koskimies and E. Mäkinen, "Automatic Synthesis of State Machines from Trace Diagrams," *Software—Practice and Experience,* vol. 24, no. 7, pp. 643-658, 1994.
[26] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCS to Statecharts," *Proc. IFIP Working Conf. Distributed and Parallel Embedded Systems,* pp. 61-72, 1998.
[27] A. Kucera and R. Mayr, "Weak Bisimilarity with Infinite-State Systems Can Be Decided in Polynomial Time," *Proc. Int'l Conf. Concurrency Theory,* pp. 368-382, 1999.
[28] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and Control in Scenario-Based Requirements Analysis," *Proc. Int'l Conf. Software Eng.,* pp. 382-391, 2005.
[29] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem, "Property Preserving Abstractions for the Verification of Concurrent Systems," *Formal Methods in System Design,* vol. 6, pp. 11-44, 1995.
[30] P. Madhusudan and P.S. Thiagarajan, "Branching Time Controllers for Discrete Event Systems," *Theoretical Computer Science,* vol. 274, pp. 117-149, 2002.
[31] B. Mahony and J.S. Dong, "Timed Communicating Object Z," *IEEE Trans. Software Eng.,* vol. 26, no. 2, pp. 150-177, Feb. 2000.
[32] R. Marelly, D. Harel, and H. Kugler, "Multiple Instances and Symbolic Variables in Executable Sequence Charts," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 83-100, 2002.
[33] S. Owre, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," *Proc. Int'l Conf. Automated Deduction,* pp. 748-752, 1992.
[34] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proc. ACM Symp. Principles of Programming Languages,* pp. 179-190, 1989.
[35] A. Pnueli and R. Rosner, "Distributed Reactive Systems Are Hard to Synthesis," *Proc. IEEE Symp. Foundations of Computer Science,* 1990.
[36] P.J. Ramadge and W.M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM J. Control and Optimization,* vol. 25, no. 1, pp. 206-230, 1987.
[37] F. Lin and W.M. Wonham, "Decentralized Supervisory Control of Discrete-Event Systems," *Information Sciences,* vol. 44, no. 3, pp. 199-224, 1988.
[38] A. Roychoudhury and P.S. Thiagarajan, "Communicating Transaction Processes: An MSC-Based Model of Computation for Reactive Embedded Systems," *Proc. Lectures on Concurrency and Petri Nets,* pp. 789-818, 2003.
[39] G. Smith and J. Derrick, "Specification, Refinement and Verification of Current Systems—An Integration of Object-Z and CSP," *Formal Methods in System Design,* vol. 18, pp. 249-284, 2001.
[40] J. Sun and J.S. Dong, "From Live Sequence Charts to Distributed Implementations," Technical Report TRC5/05, Nat'l Univ. of Singapore, 2005.
[41] J. Sun and J.S. Dong, "Synthesis of Distributed Processes from Scenario-Based Specifications," *Proc. Int'l Conf. Formal Methods,* pp. 415-431, 2005.
[42] J. Sun, J.S. Dong, J. Liu, and H. Wang, "A Formal Object Approach to the Design of ZML," *Annals of Software Eng.,* vol. 13, pp. 329-356, 2002.
[43] W. Thomas, "On the Synthesis of Strategies in Infinite Games," *Proc. Symp. Theoretical Aspects of Computer Science,* pp. 1-13, 1995.
[44] S. Uchitel and J. Kramer, "A Workbench for Synthesising Behaviour Models from Scenarios," *Proc. Int'l Conf. Software Eng.,* pp. 188-197, 2001.
[45] S. Uchitel, J. Kramer, and J. Magee, "Detecting Implied Scenarios in Message Sequence Chart Specifications," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 74-82, 2001.
[46] Int'l Telecomm. Union, *Message Sequence Chart (MSC),* 1999, Series Z: Languages and general software aspects for telecomm. systems.
[47] P. Welch, "Communicating Sequential Processes for Java (JCSP)," http://www.cs.kent.ac.uk/projects/ofa/jcsp/, 2003.
[48] J. Whittle, J. Saboo, and R. Kwan, "From Scenarios to Code: An Air Traffic Control Case Study," *Proc. Int'l Conf. Software Eng.,* pp. 490-497, 2003.
[49] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, 1996.

Jun Sun received the BSc degree from the School of Computing, National University of Singapore (NUS) in 2002. Since then he has been pursuing the PhD degree in software engineering from NUS. As of June 2006, he is a research fellow in the Department of Computer Science at NUS.

Jin Song Dong received the bachelor's (first class honors) and PhD degrees in computing from the University of Queensland in 1992 and 1996. From 1995-1998, he was a research scientist at the Commonwealth Scientific and Industrial Research Organization in Australia. Since 1998, he has been with the School of Computing at the Nation University of Singapore, where he is currently an sssociate professor and assistant dean. He is a steering committee member of the International Conference on Formal Engineering Methods (ICFEM) and the Asia Pacific Software Engineering Conference (APSEC) series.