COMPLEMENTARY FORMALISMS - SYNTHESIS, VERIFICATION AND VISUALIZATION

SUN JUN

(B.Sc. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

I am deeply indebted to my supervisor, Dr. DONG Jin Song, for his guidance, insight and encouragement throughout the course of my doctoral program and for his careful reading of and constructive criticisms and suggestions on drafts of this thesis and other works.

I owe thanks to CHEN Chun Qing, FENG Yu Zhang, LI Yuan Fang, Dr. QIN Sheng Chao, Dr. SUN Jing, Dr. WANG Hai, and other office-mates and friends for their help, discussions and friendship. I also owe thanks to Prof. Dines BJORNER, Dr. Yves BONTEMPS, Dr. Abhik ROYCHOUDHURY and Prof. P. S. THIAGARAJAN for suggestions and help on this thesis and other works.

I would like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis. I would also like to thank Hugh Anderson for his helpful comments on the draft of the thesis.

This study received funding from the project "Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems" supported by Ministry of Education (MOE) of Singapore and the project "Defense Innovative Research Project (DIRP) – Formal Design Methods and DAML" supported by Defense & Science Technology Agency (DSTA) of Singapore and the project "Reliable Software Design and Development for Sensor Network Systems" supported by National University of Singapore Academic Research Fund. The School of Computing also provided the finance for me to present papers in several conferences overseas. In addition, I have been encouraged by receiving the Dean's Graduate Award 2005. For all this, I am very grateful.

I sincerely thank my parents Bai Xing and Hui Juan, and my aunt Yue Juan for their love, encouragement and financial support in my years of study. Lastly, I would like to thank my wife HUANG Qiu Qin, for all the love.

Contents

Summary

Over the last few decades, many specification languages have been proposed, targeting different systems, different aspects of complex systems, and systems at different stages of development. Two complementary approaches have proven useful in practice. Logic-based formalisms like Z and CSP are based on mathematical techniques which provide the means for defining notions like consistency, completeness, and refinement. Diagrammatic notations like sequence charts or Statecharts are based on visual transition diagrams and are widely accepted by industry. One challenge of designing complex computer systems is to find benefiting formalisms from those that may vary significantly in presentation and establish sound connections between them. A long-cherished goal of software engineering is the mechanized synthesis of implementations from high-level specifications. An important part of this thesis is dedicated to the problem of synthesis. For system engineering starting with state-based formal specification, we developed a method of synthesizing implementable finite state machines from logic-based Object-Z models with history invariants. For system development starting with scenario-based diagrams, we investigated ways of synthesizing distributed object systems from Live Sequence Charts without constructing the global state machine. By combining the two approaches, we achieve the goal of generating implementations from system specifications with not only complicated control flow but also complex data structures. In addition, this thesis also investigates sound transformations between different formalisms so that existing theory and tool support can be reused for visualization and verification. Logic-based models can be visualized by diagrammatic languages like UML to allow easy grasp of essential facts. Using transformation techniques, mature verification mechanisms can be reused over formalisms other than those intended to discover design errors inexpensively. In a nutshell, we established various connections between complementary formalisms, which provide constructive methods for system development.

List of Figures

Chapter 1

Introduction and Overview

'it would be of very little use without my shoulders.
Oh, how I wish I could shut up like a telescope!
I think I could,
if I only knew how to begin.'
- *Alice's Adventures in Wonderland*, Lewis Carroll

1.1 Motivation and Goals

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 [154, 81, 65, 32, 79, 161, 137, 134, 102, 83, 132, 2]. 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 [63].

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. Logic-based formalisms are further divided into two groups[1], state-oriented formalisms, including VDM [83], Z [161], Object-Z [137], etc., and event-oriented formalisms, including Communicating Sequential Processes (CSP [79]), Timed CSP [134], Π-calculus [132], etc. The other category is visual formalisms, including diagrammatic modeling languages and notations. Two groups of them are of particular interest in this thesis. One is scenario-based diagrams, e.g., Message Sequence Charts (MSC) [81] and its variations like Live Sequence Charts (LSC) [32]. The other group includes those based on the notion of state machines, including finite state machines, Statecharts [65], Petri-net [119], Timed Automata [2], etc.

Both groups of formalisms have their unique strengths. Logic-based formalisms are strictly based on mathematical techniques which provide the means of precisely defining notions like consistency, refinement, completeness and, more relevantly, specification, implementation, and correctness. They often have strong tool support to validate their models, e.g., FDR (Failure Divergence Refinement) for CSP [128], Z/EVEs for Z [131]. Used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. However, logic-based formalisms are relatively unpopular compared to visual formalisms. One of the reasons is that they are used only by system engineers with relevant mathematical background [29]. By contrast, 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

[1]Besides these two groups, there are also properties-oriented formalisms including CafeObj [37], Larch [62] and variants of temporal logic [105]. Because property-oriented formalisms lack the notion of state or event and need not to be complete, they can not be used as a complete system specification from which the implementation is derived.

used to capture test cases. Visual formalisms with formal semantics also have tool support for simulation and verification, e.g., *Play-Engine* for LSC [70], UPPAAL for Timed Automata [9]. 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 [94, 43, 118, 55]. In this thesis, we explored complementary interplays between logic-based and visual formalisms so that more *constructive* methods than specification, for example specification development, analysis and evolution, can be provided. The goal is to maximally reuse mature formal modeling techniques and their tools to benefit the software development process. Ultimately, the following shall be achieved:

- Promote the usage of logic-based formal methods by connecting them to popular industrial modeling languages.
- Extend the usage of existing mature tools to visualize, validate models in different modeling languages.
- Mechanically generate implementable models all the way from early stage requirements.

One of the long-cherished goals of software engineering is the mechanized synthesis of implementations from high-level specifications. A main part of our work is dedicated to the problem of synthesis. For system engineering starting with logic-based formalisms, a state-based modeling language like Object-Z serves as an abstract and complete basis for synthesis of finite state machine designs. We developed a method of synthesizing implementable finite state machines from logic-based Object-Z models with history invariants. Thus, we achieve separation of concerns by modelling the data and functional aspects, and automatically generating dynamic control flow which in

term leads to prototype implementations. For system development starting with visual formalisms, scenario-based sequence diagrams are often used as a high-level specification language to capture system requirements in the early stage of system development. We explored ways of synthesizing distributed object systems from LSC using theoretical results from CSP. The key point is that our synthesis strategy works without constructing the global state machine so as to avoid state space explosion. Lastly, we propose that logic-based and visual formalisms can be used in combination to specify industrial scale systems. By combining the two approaches, we achieve the goal of generating implementations from system models with not only intensive interactive behaviors but also complex data structures. The challenge of automatically constructing an object-system, especially a distributed one, from high-level specifications has been long recognized [122].

This thesis also explores semantic-based transformations between logic-based and visual formalisms pursuing objectives including visualization and verification. The lightweight and intuitive complementary interplay is that logic-based models can be visualized by diagrammatic notions like UML to allow easy grasp of essential facts. Reusing mature verification mechanism over formalisms other than those intended allows discovery of design errors inexpensively. The challenge of such interplay is to find benefiting formalisms from those that may vary dramatically in syntax and establish sound connections between them.

1.2 Thesis Outline and Overview

The main contribution of our work is the investigation of complementary connections between logic-based formalisms and visual formalisms. The three objectives, namely visualization, verification and synthesis, are presented in the order of their importance.

Chapter 2 is devoted to an overview of relevant specification languages which are shared among the subsequent chapters. We review the Z specification language and its object-oriented extension Object-Z as representatives of state-based formalisms. The classic CSP and its timed extension Timed CSP are briefly introduced as examples of event-based process algebra. Introductions to

diagrammatic notations like sequence diagrams, state machines are scattered in the chapters where they are relevant.

In Chapter 3, an intuitive yet effective complementary interplay is presented, i.e., visualize logic-based specifications with UML diagrams. To demonstrate that visualization may be applied to both state-based and event-based formalisms, we investigate an integrated formal specification language named TCOZ and develop semantic-based transformation from TCOZ to both sequence diagrams and state machines. Although visualization may be theoretically lightweight, it is highly practical and we believe that it may improve the popularity of formal methods in industry.

Chapter 4 addresses the verification problem. The aim is to show that existing verification mechanisms can be effectively reused. Without building new tool support from scratch, we show that LSC, as an example of visual formalisms, can be verified by using a mature model checker for logic-based formalisms, namely FDR for CSP. In the other direction, verification of Timed CSP and TCOZ using existing tools for visual formalisms like UPPAAL are briefly discussed.

Chapters 5, 6 and 7 are devoted to the problem of synthesis. We show that low-level design languages like state machines can be systematically synthesized from high-level specification. In Chapter 5, we propose a way of synthesizing distributed designs from scenario-based specifications, namely LSC. Mature theories developed for CSP are used to group local behaviors of each object without constructing the global state machine. For system engineering starting with logic-based formalisms, state-based modeling language like Object-Z serves as an abstract and complete basis for synthesis of finite state machine designs. In Chapter 6, we present a systematic way of extracting implementable system designs from Object-Z models with history invariants. In Chapter 7, the two approaches are combined so that we may achieve the goal of generating implementations from system models with not only intensive interactive behaviors but also complex data structures.

Lastly, Chapter 8 concludes this thesis with possible future research trends. For the sake of readability, related works of this thesis are distributed to the relevant chapters. Figure 1.1 and 1.2 shows the structure of the thesis.

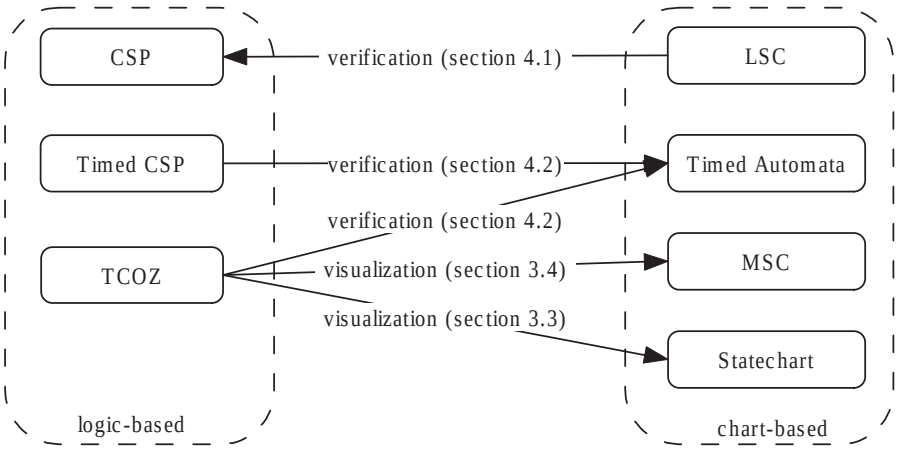


Figure 1.1: Visualization and verification

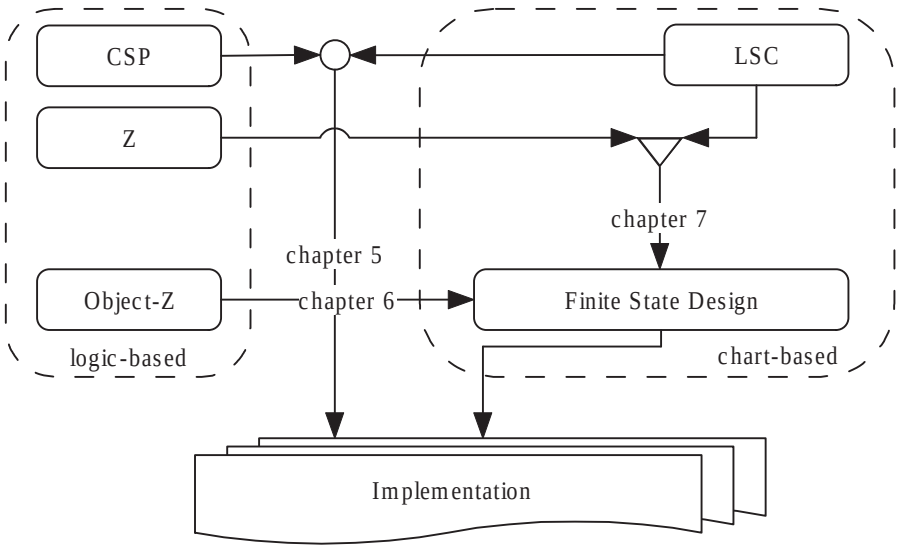


Figure 1.2: Synthesis

1.3 Publications from the Thesis

Most chapters of the thesis have been accepted in international refereed conference proceedings or journals. The work in Chapter 3 Section 3.4 was presented at *The 4th International Conference on Integrated Formal Methods IFM'04 (April 2004, Canterbury, UK)* [48]. The work in Chapter 3 Section 3.3 was used as a basis for the paper presented at *The 4th International Conference on Formal Engineering Methods ICFEM'02 (October 2002, Shanghai)* [46]. The work in Section 4.1 was presented at *The 10th International Conference on Engineering of Complex Computer Systems ICECCS'05 (June 2005, Shanghai)* [144]. The work in Chapter 4 Section 4.2 was used as a basis for the paper presented at *The 6th International Conference on Formal Engineering Methods ICFEM'04 (November 2004, Seattle)* [42]. Part of the work in Chapter 5 was presented at *The International Symposium of Formal Methods Europe FM'05 (July 2005, Newcastle upon Tyne)* [145]. The work in Chapter 6 was presented at *The 10th International Conference on Engineering of Complex Computer Systems ICECCS'05 (June 2005, Shanghai)* [143]. The work in Chapter 7 has been published in *IEEE Transactions on Software Engineering* [146].

Besides, part of Section 4.2 has been accepted for publication [44]. Part of Chapter 5 has been submitted for publication [147]. I also made partial contributions to other publications [49, 151, 45, 96, 93, 155, 64] which are although related to this thesis, they can be considered as side-stories to the impact of this thesis work.

Chapter 2

Notations and Languages

'Have you seen the Mock Turtle yet?'

'No,' said Alice.

'I don't even know what a Mock Turtle is.'

'It's the thing Mock Turtle Soup is made from,' said the Queen.

- *Alice's Adventures in Wonderland*, Lewis Carroll

In this chapter, representatives of logic-based formalisms are reviewed. Brief introductions to diagrammatic notations like sequence diagrams, state machines are scattered in later chapters where they are relevant.

2.1 State-based Formalisms

The Z specification language [161] and its extension [50] 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-like syntax is used throughout the thesis to formalize our work.

2.1.1 The Z 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 [73], including transaction processing systems and communication protocols. It has been standardized by ISO 13568:2002 [80].

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:

$$\mathcal{Q}x : R \mid c \bullet p$$

where $\mathcal{Q}$ 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 2.1.1 (Quantified predicate)

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

where $\mathbb{Z}$ is the set of integers and $\mathbb{N}$ 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. **end**

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 $Predicate$. We may also define things by abbreviation, or by axiom.

Example 2.1.2 (Abbreviation definition)

$$Illumination == 0 \mathinner{..} 100$$

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

Example 2.1.3 (Axiom definition)

$$\begin{array}{|l} pythagorean : \mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N} \\ \hline \forall\, x, y, z : \mathbb{N} \bullet (x, y)\ \underline{pythagorean}\ z \Leftrightarrow x * x + y * y = z * z \end{array}$$

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 $x^2 + y^2 = z^2$. **end**

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 2.1.4 (Free type definition) The set $\mathbb{N}$ 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. **end**

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 2.1.5 (State schema) The following schema encapsulates the state information of a light object.

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

where $\mathbb{B}$ 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. **end**

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 2.1.6 (Operation schema) The following operation schema defines the operation $Adjust$ by stating how the state variables of the $Light$ schema are updated:

$$
\begin{array}{|l}
\hline
\textit{Adjust} \\
\Delta \mathit{Light} \\
dim? : \mathit{Illumination} \\
\hline
on = \mathit{true} \wedge dim' = dim? \\
\hline
\end{array}
$$

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. We remark that if $dim?$ is zero, the state variable on will be set to false because of the state invariant. **end**

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 2.1.7 (Z package)

$$
\begin{array}{|l}
\hline
\textit{Light} \\
dim : Illumination \\
on : \mathbb{B} \\
\hline
dim > 0 \Leftrightarrow on = true \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{Adjust} \\
\Delta Light \\
dim? : Illumination \\
\hline
on = true \wedge dim' = dim? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{TurningOn} \\
\Delta Light \\
\hline
on = false \wedge dim' = 100 \wedge on' = true \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{TurningOff} \\
\Delta Light \\
\hline
on = true \wedge dim' = 0 \wedge on' = false \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{LightInit} \\
Light' \\
\hline
dim' = 0 \wedge on' = false \\
\hline
\end{array}
$$

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 $Adjust$ are defined to turn on or turn off the light or set the light level to a specific level. **end**

The glossary of Z notation is summarized in Appendix A. 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. Example combinations are presented in Section 3.2 and 7.2.

2.1.2 Object-Z

Object-Z [50] is an object-oriented extension of the Z language. It has been developed by a team of researchers at the Software Verification Research Center, University of Queensland. It improves the clarity of large Z specifications through enhanced structuring. The main Object-Z construct is the $class$ definition, which captures the object-oriented notion of a class by encapsulating a state schema with all the operations which may affect its variables. As well as being used to specify objects, Object-Z classes can be directly reused in definitions of other classes. A class may be specified as a specialization or extension of another class using inheritance.

An Object-Z class is represented syntactically as a named box with zero or more generic parameters. There may be local types and constant definitions, at most one state schema and one initial schema written as INIT and zero or more operations. The declarations of the state schema are referred to as state variables and the predicate as class invariants. The class invariant restricts the possible valuations of the state variables. The initial schema identifies the possible initial valuations. An operation is either an operation schema or a schema expression involving existing class operations and schema operators.

Example 2.1.8 (Object-Z class) Figure 2.1 shows an Object-Z specification of a queue class, where $Package$ is a given type representing network communication packages. The internal structure of a package is of no interest in the modeling. The queue is modeled as a sequence of packages as defined in the (anonymous) state schema. The sequence is initially empty as specified in the INIT schema. Operations are provided to allow items to join or leave the queue on a first-in/first-out basis. This queue class models an incoming channel of a network router. The total function $expires$ tells whether a package has expired (by examining certain flag bits in the package). A package is put into the queue only if it is not expired and all packages in the queue may be later forwarded. **end**

Operation schemas in an Object-Z class are given a standard Z semantics, which is used to develop a transition-system semantics [161]. 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

$[Package]$

Queue

$\upharpoonright(\textsc{Init}, Join, Leave)$

$expires : Package \rightarrow \mathbb{B}$

$items : \mathsf{seq}\, Package$

Init

$items = \langle\, \rangle$

Join

$\Delta(items)$
$item? : Package$

$expires(item?) \Rightarrow items' = items$
$\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle$

Leave

$\Delta(items)$
$item! : Package$

$items = \langle item! \rangle \frown items'$

Figure 2.1: Object-Z class

the initial states for which the outcome of the operation is properly defined. In Z semantics, if an operation is applied outside its domain (the precondition), the system diverges. By contrast, Object-Z adopts a blocking semantics. An operation can only occur when its precondition is satisfied. When its precondition is not satisfied, the operation is said to be *blocked*, i.e., it is not available for application. The blocking semantics is safer because there is no need for the specifier to check if operations are sufficiently defined and, in the cases they are not, combine them with appropriate error handling. Another consequence of the blocking semantics is that Z refinement of an operation by weakening pre-conditions is not applicable.

Definition 1 *Let $Operation$ be an operation schema. Let $State$ be the state schema, and $inputs$ ($outputs$) be the list of inputs (outputs) associated with the operation. The precondition of the operation, written as pre($Operation$), is defined as:*

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

where the schema $Operation \setminus outputs$ may be obtained by existentially quantifying each component in $outputs$ within $Operation$.

The precondition hides any components that correspond to the state after the operation, and any outputs that happen to be present.

Definition 2 *If a state ($State_a$)*[1] *satisfies the precondition of $Operation$, the postcondition of the $Operation$ from state $State_a$, written as post($Operation, State_a$) is:*

$$\mathit{post}(Operation, State_a) \mathrel{\widehat{=}} State_a \wedge Operation$$

Example 2.1.9 (Precondition and postcondition) The precondition of operation $Adjust$ in Example 2.1.7 is:

$$\begin{array}{ll}
\mathrm{pre}(Adjust) \mathrel{\widehat{=}} & \\
\quad \exists\, dim' : 0 \mathinner{..} 100;\ on' : \mathbb{B} \mid & \\
\qquad dim' > 0 \Leftrightarrow on' = true \bullet & \text{– Invariant in Post-state} \\
\qquad\quad on = true \wedge dim' = dim? & \text{– Def. of } Adjust
\end{array}$$

[1] In this thesis, state and predicate are used interchangeably.

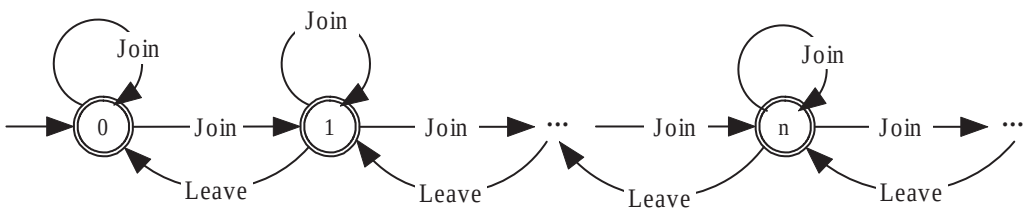
Given the state where $dim > 0 \wedge on = true$, the postcondition of the operation $Adjust$ is:

$$\begin{aligned}
&\text{post}(Adjust, dim > 0 \wedge on = true) \mathrel{\widehat{=}} \\
&\quad dim > 0 \Leftrightarrow on = true \wedge dim' > 0 \Leftrightarrow on' = true \wedge && \text{– invariant} \\
&\qquad dim > 0 \wedge on = true \wedge on = true \wedge dim' = dim?
\end{aligned}$$

The precondition and postcondition can be further simplified using predicate logic. For instance, the above postcondition can be simplified as $dim' = dim?$. **end**

The operations of a class form a named collection of relations, which determines a transition system in which a given operation may fire exactly when its Z precondition is satisfied. The semantic model thus consists of all the sequences of operations/events which can be performed by the transition system.

Example 2.1.10 (Transition system semantics) The class $Queue$ defines the following state transition system:



For simplicity, the state is only distinguished by the number of packages in the queue since the packages in the queue are considered identical in the modeling. **end**

The properties represented by a state transition system are referred to as safety properties. They specify which state changes may occur but do not require that any state changes actually do occur. Properties which state that a state change, or an operation, must occur are referred to as liveness properties. Object-Z allows the specification of liveness properties by associating each class with a history invariant in the form of a temporal logic formula. The history invariant restricts the set of histories derived from the state of the class. The notion of history invariant was introduced

in early versions of the Object-Z language [136, 51]. However, it is not included in the Graeme Smith's work [137] for practical reasons. We believe that the history invariant is an effective method to strengthen the weak process control logic of Object-Z. For simplicity, the history invariant is restricted to Linear-time Temporal Logic (LTL [120]) in this thesis. History invariants other than standard LTL formulæ appearing in [136, 51] can be reframed in LTL by introducing auxiliary variables.

Example 2.1.11 (Object-Z class with history invariant) The following is an Object-Z class with history invariant:

$FairBoundedQueue$

$Queue$

$max : \mathbb{N}$

$\Box\Diamond\#items = 0$
$\Box\#items \leq max$

This class is a subclass of $Queue$, indicated by the first line in the class box. A state variable max is defined in the state schema, in addition to those defined in the state schema of class $Queue$. The variable max models the capacity of the queue. The state invariant (the last two lines) states that the queue is eventually empty and the number of items in the queue is always bounded by max. The temporal operators $\Box$ and $\Diamond$ are borrowed from modal logic [53]. Intuitively, $\Box$ can be read as 'always' and $\Diamond$ as 'eventually'. **end**

2.2 Event-based Formalisms

Hoare's classic Communicating Sequential Process (CSP) and its timed extensions Timed CSP are our choice of representatives for event-oriented formalisms.

2.2.1 Communicating Sequential Processes

The notion of CSP was introduced in Tony Hoare's classic paper [79]. The original language derives its full name from the built-in syntactic constraint that processes belong to the sequential subset of the language. A characteristic of CSP is that processes have disjoint local variables, which was influenced by Dijkstra's principle of *loose coupling* [39]. CSP has passed the test of time. It has been widely accepted and influenced the design of many recent programming and specification languages including Ada [54], occam [110], Concurrent ML [126], BPEL4WS [84], and Orc [112].

CSP is a formal specification language where processes proceed from one state to another by engaging in events. Processes may be composed by using operators which require synchronization on events, i.e., each component must be willing to participate in a given event before the whole system makes the transition. Synchronous communication, rather than assignments to shared state variables, is the fundamental means of interaction between agents. A CSP process is defined by process expressions.

Definition 3 *Let P denote all possible CSP processes. The syntax of a CSP process is defined as:*

$$
\begin{array}{rcll}
P & ::= & \textsc{Run}_{\Sigma} & \text{– replicated choice} \\
 & | & \textsc{Stop} & \text{– deadlock} \\
 & | & \textsc{Skip} & \text{– termination} \\
 & | & \perp & \text{– divergence} \\
 & | & e \rightarrow P & \text{– event prefixing} \\
 & | & P_1 \langle\!\langle\ b\ \rangle\!\rangle P_2 & \text{– conditional choice} \\
 & | & P_1 \sqcap P_2 & \text{– internal choice} \\
 & | & P_1 \Box P_2 & \text{– external choice} \\
 & | & P_1 \,|||\, P_2 & \text{– interleaving} \\
 & | & P_1 \,|[\,\Sigma\,]|\, P_2 & \text{– generalized parallel} \\
 & | & P_1 \ {}_X||_Y\ P_2 & \text{– alphabetized parallel} \\
 & | & \|_{k=1}^{n} (P_k, \Sigma_k) & \text{– replicated parallel} \\
 & | & P_1;\ P_2 & \text{– sequential composition} \\
 & | & P_1 \ \nabla_e\ P_2 & \text{– interrupt} \\
 & | & \mu\, X \bullet P(X) & \text{– recursion}
\end{array}
$$

$\textsc{Run}_{\Sigma}$ is a process always willing to engage in any event in Σ. $\textsc{Stop}$ denotes a process that deadlocks and does nothing. A process that terminates is written as $\textsc{Skip} \mathrel{\widehat{=}} \checkmark \rightarrow \textsc{Stop}$, where $\checkmark$ is the

termination event. The process $\perp$ is the most unpredictable and most uncontrollable of processes. It behaves chaotically. A process which may participate in event e then act according to process description P is written as $e \rightarrow P$. The event e is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially.

Diversity of behavior is introduced through choice operators. The conditional choice $P_1 \langle\!\langle\ b\ \rangle\!\rangle P_2$ behaves as P_1 if the Boolean formula b is true and else P_2. The external choice operator ($\Box$) allows a process of choice of behavior according to what events are requested by its environment[2]. For instance, the process $(a \rightarrow P) \Box (b \rightarrow Q)$ begins with both a and b enabled. The environment chooses which event actually occurs by requesting one or the other first. Subsequent behavior is determined by the event which actually occurred. Internal choice represents variation in behavior determined by the internal state of the process. The process $a \rightarrow P \sqcap b \rightarrow Q$ may initially enable either a or b or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.

Example 2.2.1 (Simple vending machine) The following is a specification of a trivial vending machine.

$$VM \mathrel{\widehat{=}} coin \rightarrow (coffee \rightarrow \textsc{Stop} \sqcap candy \rightarrow \textsc{Stop})$$

Event $coin$ is the action of inserting a coin to the vending machine. After a coin is inserted, the vending machine dispatchs a cup of coffee or a candy randomly and then stops reacting. **end**

The parallel composition of processes P_1 and P_2, synchronized on common events of their alphabets X, Y (or a common set of events A) is written as $P_1 \ {}_X||_Y \ P_2$ (or $P_1 \,|[\, A \,]|\, P_2$). No sharing event may occur unless enabled jointly by both P_1 and P_2. When a sharing event does occur, it occurs in both P_1 and P_2 simultaneously and is referred to as *synchronization*. Events not sharing may occur in either P_1 or P_2 separately but not jointly.

[2]External choice and temporal operator 'always' share the same symbol for historical reasons. In this thesis, $\Box$ is used to denote external choice if not explicitly stated otherwise.

The sequential composition of P_1 and P_2, written as $P_1;\ P_2$, acts as P_1 until P_1 terminates by communicating a distinguished event $\checkmark$ and then proceeds to act as P_2. The termination signal is hidden from the process environment and therefore occurs as soon as enabled by P_1. The interrupt process $P_1 \ \triangledown_e \ P_2$ behaves as P_1 until the first occurrence of event e, then the control passes to P_2. Recursion is used to give a finite representation of non-terminating processes. The process expression $\mu\, X \bullet P(X)$ describes a process which contains a recursion point X.

In general, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal state. The approach adopted by CSP is to allow a process definition to be parameterized by state variables. A definition of the form $P(x)$ represents a family of definitions, one for each possible value of x.

Example 2.2.2 (Vending machine) The following is a CSP specification of a more realistic vending machine:

$$
\begin{array}{l}
VendingMachine(quote) \mathrel{\widehat{=}} \\
\quad drop?coin \rightarrow VendingMachine(quote + coin) \\
\quad \Box\ [quota > 0] \bullet release \rightarrow releasecoin \rightarrow VendingMachine(0) \\
\quad \Box\ [quota \geq 80] \bullet button?coffee \rightarrow VendingMachine(quota - 80) \\
\quad \Box\ [quota \geq 50] \bullet button?candy \rightarrow VendingMachine(quota - 50)
\end{array}
$$

The vending machine dispatches either coffee or candy. A coffee costs 80 cents and a candy costs 50 cents. The process is parameterized by the amount inserted by the user. A user may insert coins repeatedly before requesting an item. He (she) may as well ask the machine to release all coins inserted so far. An item is dispatched only when sufficient coins have been inserted. A channel is a method to group events. Two channels are used in this example, $drop$, $button$. A synchronization on channel $drop$ represents an insertion of a coin. A synchronization on channel $button$ represents a request of an item. Event $release$ is the user request to release all the coins. **end**

Three mathematical models for CSP have been defined. In the traces model, a process is represented by the set of finite sequences of communications it can perform, denoted as $traces(P)$. In the

stable failures model, a process is represented by its traces and also by its failures. A failure is a pair (t, Σ), where t is a finite trace of the process and Σ is a set of events it can refuse after t (refusal). The set of P's failures is denoted as $failures(P)$. In the failures/divergences model [22], a process is represented by its failures as well as its divergences, denoted as $divergences(P)$. A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions. Interested readers should refer to [128] for detailed definitions of the three semantics models.

Three forms of refinement have been defined, corresponding to the three semantics models. Traces refinement means traces containment. It is used for proving safety properties. Failures refinement is normally used to prove failures-divergence refinement for divergence-free processes. Failures-divergence refinement is used for proving safety, liveness and combinational properties, and also for establishing refinement and equality relations between systems. Two processes P_1, P_2 are equivalent, denoted as $P_1 = P_2$, if and only if $failures(P_1) = failures(P_2)$ and $divergences(P_1) = divergences(P_2)$. Equivalence of processes can be proved or disproved by appealing to algebraic laws. The laws that are relevant to the works in this thesis include the following: the formal proof of the laws can be found in [79] or [128],

$$
\begin{array}{lll}
P \,|[\,\Sigma\,]|\, \textsc{Run}_{\Sigma} & = P & \text{– L1} \\
P \parallel \textsc{Stop} & = \textsc{Stop} & \text{– L2} \\
P \parallel P & = P & \text{– L3} \\
P_1 \;{}_X||_Y\; P_2 & = P_2 \;{}_Y||_X\; P_1 & \text{– L4} \\
(P_1 \;{}_X||_Y\; P_2) \;{}_{X \cup Y}||_Z\; P_3 & = P_1 \;{}_X||_{Y \cup Z}\; (P_2 \;{}_Y||_Z\; P_3) & \text{– L5}
\end{array}
$$

2.2.2 Timed CSP

The language of CSP and the semantics models introduced so far are appropriate for describing and analyzing systems in terms of their possible sequences of events. All the semantics models deliberately abstracted away concerns about timing such as the precise time at which events occur. Real-time systems, which can only be modeled and analyzed using a quantitative notion of time, are commonplace, for example traffic control, robotics, virtual reality, etc. Timed CSP is an extension of the CSP language to specify and model real-time systems [125]. It extends ordinary CSP by

introducing a capability to quantify temporal aspects of sequencing and synchronization. To the standard CSP process operators, Timed CSP adds a number of time specific ones, e.g., timed event prefix, the delay, the timed interrupt and the timeout.

Definition 4 *Let T denote the set of all possible timed CSP processes. The syntax of a Timed CSP process is defined as:*

$$
\begin{array}{rcll}
T & ::= & P & \text{– CSP process} \\
 & | & e \bullet t \rightarrow T(t) & \text{– timed prefix} \\
 & | & e \xrightarrow{t} T & \text{– delay} \\
 & | & \textsc{Wait}[t] & \text{– wait} \\
 & | & T_1 \triangledown\{t\}\ T_2 & \text{– timed interrupt} \\
 & | & T_1 \triangleright\{t\}\ T_2 & \text{– timeout}
\end{array}
$$

The optional timing parameter t of the timed prefix records the time, relative to the start of the process, at which the event e occurs and allows the subsequent behavior $T(t)$ to depend on its value. The process $e \xrightarrow{t} P$ delays process P by t time units after engaging in event e. A process which allows no communications for period d time units then terminates is written as $\textsc{Wait}[t]$. It is used to delay a subsequent process for a specific number of time units.

The timed interrupt $T_1 \triangledown\{t\}\ T_2$ initially behaves as process T_1, and passes control to a subsequent process T_2 as soon as the time period t has elapsed. The timeout process, written as $T_1 \triangleright\{t\}\ T_2$, passes control to a subsequent process T_2 if no event has occurred in the primary process T_1 by some deadline t.

In the operational semantics of Timed CSP [135], the semantics of a Timed CSP process is defined by identifying how the process may evolve through time or by engaging in instantaneous events. In the denotational semantics models, it is defined by stating the set of possible observations, e.g., traces, failures and timed failures [33]. Semantically, the only addition to the CSP language is the process construct $\textsc{Wait}[d]$. Other timed related operators can be interpreted in terms of ordinary CSP operators and the $\textsc{Wait}[d]$ process [33]. For instance,

$$
\begin{array}{ll}
e \xrightarrow{t} T & = e \rightarrow (\textsc{Stop} \triangleright\{t\}\ T) \\
T_1 \triangleright\{t\}\ T_2 & = T_1 \Box (\textsc{Wait}[t];\ T_2) \\
T_1 \swarrow\{t\}\ T_2 & = T_1 \triangledown (\textsc{Wait}[t];\ T_2)
\end{array}
$$

Example 2.2.3 (Timed vending machine) The following is a specification of a timed vending machine.

$$
\begin{array}{l}
TVendingMachine(quote) \mathrel{\widehat{=}} \\
\quad [quota = 0] \bullet drop?coin \xrightarrow{1} TVendingMachine(coin) \\
\quad \Box\ [quota > 0] \bullet (drop?coin \xrightarrow{1} TVendingMachine(quote + coin) \\
\qquad \Box\ [quota > 0] \bullet release \rightarrow releasecoin \rightarrow TVendingMachine(0) \\
\qquad \Box\ [quota \geq 80] \bullet button?coffee \xrightarrow{3} TVendingMachine(quota - 80) \\
\qquad \Box\ [quota \geq 50] \bullet button?candy \xrightarrow{2} TVendingMachine(quota - 50)) \\
\qquad \rhd\{60\}\ (releasecoin \rightarrow TVendingMachine(0))
\end{array}
$$

When variable $quota$ is of value 0, the only action enabled is by communication through channel $drop$, i.e., insert a coin. There is a delay of one time unit before the machine accepts the coin and updates the variable $quota$. After that, the user may continue inserting coins or request the vending machine to release all coins inserted thus far. Once sufficient coins have been inserted, the user may request for either coffee or candy. Whenever the choice is made, the vending machine dispatches the corresponding drink, taking a reasonable amount of time. If the user idles more than 60 seconds after inserting a coin, the machine releases the coins. **end**

Chapter 3

Visualization

'And what is the use of a book', thought Alice.

'without pictures or conversations?'

- *Alice's adventures in wonderland*, Lewis Carroll

Visualization is more than a method of computing. It is a process of transforming information into a visual form enabling the viewer to observe, browse, make sense, and understand the information. Visualization typically employs computers to process the information and computer screens to view it using methods of interactive graphics, imaging, and visual design. It relies on the visual system to perceive and process the information. The beauty of effective visualization is more than skin deep.

3.1 Introduction

Logic-based formalisms, either state-based ones like Z or event-based ones like CSP, are elegant and precise. However, logic-based modeling often relies on heavy mathematical notations. It presents a difficulty for the software engineers without relevant mathematical background, which we believe is one of the reasons why logic-based formal methods are relatively unpopular in industry.

By contrast, diagrammatic notation like MSC, Statechart and UML are intuitive and easy to understand. They are widely accepted by the industry for system modeling and analyzing. Massive amounts of human power and resources have been dedicated to system engineering based on those graphical notations. In this chapter, we investigate ways of visualizing logic-based modeling using visual diagrams. As well as showing essential facts, visualizing logic-based modeling using popular graphic notations like UML diagrams makes it possible to reuse existing tool support for test case generation, verification, code synthesis, etc.

This work has been based on the notion of Timed Communicating Object-Z (TCOZ [100]). TCOZ is an effective integration of Timed CSP and Object-Z. It is capable of specifying systems with not only complicated control flow but also complex data structures. Sound projections have been established to visualize different viewpoints of the integrated modeling. The intra-object control flows of TCOZ models are visualized using the notion of Statecharts [65]. The inter-object interaction is visualized using Message Sequence Charts. Being based on an expressive language like TCOZ, we are confident that the approach can be applied to other logic-based formalisms.

3.2 An Integrated Specification Language

TCOZ was introduced by Mahony and Dong in [100] to allow complete and coherent specification of complex systems. It is a blending of Object-Z and Timed CSP. The basic structure of a TCOZ document is the same as for Object-Z, which consists of a sequence of definitions, including type and constant definitions in the usual Z style. TCOZ varies from Object-Z in the structure of class definitions, which may include CSP channel and process definitions. Channels in TCOZ are defined as communication interfaces between objects. All dynamic interactions between objects must take place through the channel communication mechanism. The true power of TCOZ comes from the ability to make use of Timed CSP primitives in describing the process aspects of an operation's behavior. All operation definitions in TCOZ are Timed CSP process definitions. Operation schemas are identified with terminating Timed CSP processes. The data-related aspects of TCOZ are modeled using state bindings and the process-related aspects are modeled using event traces and

refusals [101]. In the following, a simplified version of the Light Control System (LCS) [52] is used as a running example to illustrate the features of TCOZ as well as to demonstrate the visualization.

3.2.1 Light Control System

LCS is an intelligent control system. It can detect the occupation of a building, and then turn on or turn off the lights automatically. It is able to tune illumination (in percentage) in the building according to the outside light level. It consists of three components: a light, a motion detector and a room controller. A typical system behavior is that when a user enters a room: the motion detector senses the presence of the person, and then the room controller reacts by receiving the current daylight level and turning on the light group with appropriate illumination setting. When a user leaves a room (leaving it empty): the detector senses no movement, the room controller waits for certain time units and then turns off the light group. In addition, the occupant can directly turn on/off the light by pushing the button.

Light

$dim : Illumination$
$on : \mathbb{B}$

INIT

$dim = 0 \wedge on = false$

TurningOn

$\Delta(dim, on)$

$dim' = 100 \wedge on' = true$

TurningOff

$\Delta(dim, on)$

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

Class *Light* is an ordinary Object-Z class. An ordinary Object-Z class in TCOZ simply defines a data type. It does not have its own thread of control. It is thus called as a passive class.

$\mathit{ControlledLight}$

Light

$\mathit{button}, \mathit{dimmer} : \textbf{chan}$

$$\mathit{ButtonPushing} \mathrel{\widehat{=}} \mathit{button}?1 \rightarrow ([\mathit{dim} > 0] \bullet \mathit{TurningOff} \\ \Box\ [\mathit{dim} = 0] \bullet \mathit{TurningOn})$$

$$\mathit{DimChange} \mathrel{\widehat{=}} [n : 0 \mathinner{..} 100] \bullet \mathit{dimmer}?n \rightarrow ([\mathit{on}] \bullet \mathit{dim} := n \Box [\neg\ \mathit{on}] \bullet \textsc{Skip})$$

$$\textsc{Main} \mathrel{\widehat{=}} \mu\, N \bullet (\mathit{ButtonPushing} \Box \mathit{DimChange});\ N$$

Class $\mathit{ControlledLight}$ extends Light class with channel and process definitions. The state schema is extended with channel definitions. The channel definition specifies the communication interface between the object and its environment. Channels defined in different classes with the same name are connected implicitly. In this example, button and dimmer are channels connecting the light to the environment and the room controller. The process definitions precisely state how the object interacts with its environment through the interface and reacts to environment inputs. Object-Z operation schemas are treated as terminating processes in the process definitions. For instance, in process $\mathit{ButtonPushing}$, once there is a synchronization on channel button, if the light is on, the operation $\mathit{TurningOff}$ is invoked. State guards, written as $[b] \bullet P$, is the short form for

$$P \langle\!|\ b\ |\!\rangle \textsc{Stop}$$

where b is a Boolean formula over the state variables and environmental inputs. A $\textsc{Main}$ process indicates an active object, which has its own thread of control. It determines the behavior of objects of the active class after initialization. The class $\mathit{ControlledLight}$ is a typical example of TCOZ-style specification of active agents.

$\mathit{MotionDetector}$

$\mathit{motion} : \textbf{chan}$

$\mathit{md} : (\mathit{Move} \mid \mathit{NoMove})\ \textbf{sensor}$

$$\mathit{NoUser} \mathrel{\widehat{=}} \mathit{md}?\mathit{Move} \rightarrow \mathit{motion}!1 \rightarrow \mathit{User} \\ \Box\ \mathit{md}?\mathit{NoMove} \rightarrow \textsc{Wait}\ 1;\ \mathit{NoUser}$$

$$\mathit{User} \mathrel{\widehat{=}} \mathit{md}?\mathit{NoMove} \rightarrow \mathit{motion}!0 \rightarrow \mathit{NoUser} \\ \Box\ \mathit{md}?\mathit{Move} \rightarrow \textsc{Wait}\ 1;\ \mathit{User}$$

$$\textsc{Main} \mathrel{\widehat{=}} \mathit{NoUser}$$

The motion detector detects movement in the room so is to tell whether some one is in or not. It sends proper signals to the room controller periodically. This class has a trivial data structure. There are no state variables and operation schemas. The keyword **sensor** identifies a continuous-function interface mechanism [103]. Internally, md takes the role of a CSP channel. The relationship between the public continuous-function variable and the internal channel is that whenever a value is communicated on the internal channel at a time t, that value must be equal to the value of the continuous function at that time. Intuitively, synchronization on channel md represents the output from the movement sensor. Initially, the object behaves as specified by the process $NoUser$. If no movement is detected, the object waits for 1 time unit and then continues to monitor signals from channel md. If there is some movement, a signal is sent on channel $motion$ to inform the room controller and then the object behaves as specified by the process $User$. The process $User$ is similarly defined.

$RoomController$

$dimmer, motion : \mathbf{chan}$
$odsensor : Illumination\ \mathbf{sensor}$
$absent : \mathbb{T}$
$olight : Illumination$

$Adjust$

$dim! : Percent\ \mathbf{on}\ dimmer$

$dim!\ \underline{satisfy}\ olight$

$Ready \mathrel{\widehat{=}} motion?1 \rightarrow On$
$Regular \mathrel{\widehat{=}} \mu R \bullet [n : 0 \mathinner{..} 100] \bullet odsensor?n \rightarrow Adjust;\ dimmer!dim \rightarrow R$
$On \mathrel{\widehat{=}} Regular \triangledown motion?0 \rightarrow OnAgain$
$OnAgain \mathrel{\widehat{=}} (motion?1 \rightarrow On) \rhd \{absent\}\ Off$
$Off \mathrel{\widehat{=}} dimmer!0 \rightarrow Ready$
$\textsc{Main} \mathrel{\widehat{=}} Off$

The room controller communicates with the motion detector and the light through the shared channels. It takes in signals from the motion detector and sends proper signal to the light. The relation $satisfy$ captures the relationship between daylight level and required illumination. The operation $Adjust$ outputs the desired light level, which is sent over channel $dimmer$ to tune the light level. The process expressions are complicated with mutual recursion, in addition to complex operators like interrupt and time-out. Lastly, a light control system consists of the room controller, the motion detector and the light. The MAIN process is the parallel composition of the three instances specified using a network topology, i.e., a graph-like way of specify communication structure in TCOZ [100].

Two objects connected by a double-arrowed horizontal line may communicate through the channels written over the line. In this example, the motion detector shares the channel $motion$ with the room controller, and the room controller shares the channel $dimmer$ with the light.

$$
\begin{array}{|l}
\hline
LCS \\
\begin{array}{|l}
\hline
m : MotionDetector \\
l : ControlledLight \\
r : RoomController \\
\hline
\end{array} \\
\textsc{Main} \mathrel{\widehat{=}} \left\| (m \xleftrightarrow{motion} r \xleftrightarrow{dimmer} l) \right. \\
\hline
\end{array}
$$

This modeling is elegant and precise, but not intuitive. The explicit behavior patterns of the Light Control System are distributed among the class definitions.

3.2.2 Trace Model for TCOZ

The syntax of TCOZ process expression, written as TZE, is defined as the following (refer to Definition 3 and 4 for comparison):

Definition 5 *Let ZE represent Z expressions, ZS represent Z schemas, $NAME$ represent all valid character strings.*

$$
\begin{array}{rcll}
TZE & ::= & \textsc{Stop} \mid \textsc{Run} \mid \textsc{Skip} & \text{– primitives} \\
& \mid & op\langle\!\langle ZS \rangle\!\rangle \mid ref\langle\!\langle NAME \rangle\!\rangle & \text{– referencing} \\
& \mid & (\_.\_ \rightarrow \_)\langle\!\langle \Sigma \times ZE \times TZE \rangle\!\rangle & \text{– event prefixing} \\
& \mid & \textsc{Wait}\langle\!\langle ZE \rangle\!\rangle & \text{– delay} \\
& \mid & (\_ \bullet \_)\langle\!\langle ZS \times TZE \rangle\!\rangle & \text{– state guard} \\
& \mid & (\_ \mid \_)\langle\!\langle TZE \times TZE \rangle\!\rangle & \text{– choice} \\
& \mid & (\_ \nabla_e \_)\langle\!\langle TZE_1 \times TZE_2 \rangle\!\rangle & \text{– interrupt} \\
& \mid & (\_\_ || \_\_)\langle\!\langle TZE \times \Sigma \times \Sigma \times TZE \rangle\!\rangle & \text{– parallel} \\
& \mid & (\_ ||| \_)\langle\!\langle TZE \times TZE \rangle\!\rangle & \text{– interleaving} \\
& \mid & (\_; \_)\langle\!\langle TZE \times TZE \rangle\!\rangle & \text{– sequential composition} \\
& \mid & (\mu \_ \bullet \_)\langle\!\langle NAME \times TZE \rangle\!\rangle & \text{– recursion} \\
& \mid & (\_[\_/\_])\langle\!\langle TZE \times \Sigma \times \Sigma \rangle\!\rangle & \text{– renaming} \\
& \mid & (\_\backslash\_)\langle\!\langle TZE \times \mathbb{P}\Sigma \rangle\!\rangle & \text{– hiding} \\
& \mid & (\_ \triangleright \{\_\} \_)\langle\!\langle TZE \times ZE \times TZE \rangle\!\rangle & \text{– timeout}
\end{array}
$$

The semantic model for TCOZ is the infinite timed-states model which extends the Timed CSP's infinite timed-failure model [101]. A system model specified using Statechart or sequence diagram is characterized by the set of traces it may perform. Thus, a trace semantics is sufficient for the discussion on visualizing TCOZ models. In this section, we present a trace-based semantics simplifying the infinite timed-failure model of TCOZ. The trace model is used as a guideline for developing the mechanized projection. The main connection between the Object-Z model and the Timed CSP model is that an Object-Z operation schema $op\langle\langle ZS\rangle\rangle$ may appear in the process expression as a non-atomic terminating process. Let Σ be the set of all possible events.

$$[\Sigma]$$

A TCOZ event may be an update event (an invocation of an operation schema), a simple synchronization, a channel communication, or a termination event $\checkmark$. A trace is a (finite or infinite) sequence of events. Let Σ^* denote all possible traces that can be composed by events in Σ.

$$\Sigma^* == \mathrm{seq}\,\Sigma$$

Most of the process constructs in TCOZ are borrowed from Timed CSP. Thus, the trace model assembles the trace semantics of CSP [79, 128].

$$\mid \; traces : TZE \rightarrow \mathbb{P}\,\Sigma^*$$

The only trace of STOP is the empty one, and any sequence of events is a trace of RUN.

T1 $traces(\text{STOP}) = \{\langle\rangle\}$
T2 $traces(\text{RUN}) = \Sigma^*$

A trace of ($c.a \rightarrow TZE$) may be empty, because $\langle\rangle$ is a trace of the behavior of every process up to the moment that it engages in its very first action. Every nonempty trace begins with $c.a$, and its tail must be a possible trace of TZE.

T3 $traces(c.a \rightarrow TZE) = \{\langle c.a\rangle \frown u \mid u \in traces(TZE)\} \cup \{\langle\rangle\}$

A trace of WAIT ZE is either an empty one or a delay of ZE time units. The event $wait(ze)$ is an artificial event. It marks a time delay in the trace, which allows us to use timing constructs in MSC and Statechart to visualize simple timing aspects of the system.

T4 $traces(\textsc{Wait}\ ZE) = \{\langle\rangle, \langle wait(ze)\rangle\}$

If the state guard ZS evaluates to false, the only trace of $ZS \bullet TZE$ is the empty one. Every nonempty trace must be a trace of the process expression TZE. The state-guard, is typically used to $block$ or $enable$ execution of an operation on the basis of an object's local state (the instance's state).

T5 $traces(ZS \bullet TZE) = traces(TZE) \cup \{\langle\rangle\}$

A trace of a process which offers a(n internal or external) choice of two process expressions must be a trace of one of the alternatives. For internal choice, the choice is made upon the internal state of the system. For external choice, the choice is made by the environment.

T6 $traces(TZE_1 \mid TZE_2) = traces(TZE_1) \cup traces(TZE_2)$

A trace of $(TZE_1 \triangledown_e TZE_2)$ is a sequence event of TZE up to the moment the event e occurs. Its tail shall be a trace of TZE_2.

T7 $traces(TZE_1 \triangledown_e TZE_2) = \{s \frown \langle e\rangle \frown t \mid s \in traces(TZE_1) \wedge t \in traces(TZE_2)\}$

A trace of the parallel composition $(TZE_1\ {}_X||_Y\ TZE_2)$ is composed by two traces, one from each component, synchronizing on common events of their alphabets.

T8 $traces(TZE_1\ {}_X||_Y\ TZE_2) = \bigcup\{s\ {}_X||_Y\ t \mid s \in traces(TZE_1) \wedge t \in traces(TZE_2)\}$

where given two traces tr_1 and tr_2, $tr_1\ {}_X||_Y\ tr_2$ is a set of traces defined by the following; below x denotes a typical member of X but not Y and y is a typical member of Y but not X and z, z' are typical members of both X and Y and $z \neq z'$.

$$
\begin{array}{ll}
tr_1\ {}_X||_Y\ tr_2 & = tr_2\ {}_Y||_X\ tr_1 \\
\langle\rangle\ {}_X||_Y\ \langle\rangle & = \{\langle\rangle\} \\
\langle\rangle\ {}_X||_Y\ \langle y\rangle & = \{\langle y\rangle\} \\
\langle x\rangle\ {}_X||_Y\ \langle\rangle & = \{\langle x\rangle\} \\
\langle\rangle\ {}_X||_Y\ \langle z\rangle & = \varnothing \\
\langle z\rangle\ {}_X||_Y\ \langle\rangle & = \varnothing \\
\langle x\rangle \frown tr_1'\ {}_X||_Y\ \langle z\rangle \frown tr_2' & = \{\langle x\rangle \frown tr \mid tr \in (tr_1'\ {}_X||_Y\ \langle z\rangle \frown tr_2')\} \\
\langle z\rangle \frown tr_1'\ {}_X||_Y\ \langle z\rangle \frown tr_2' & = \{\langle z\rangle \frown tr \mid tr \in tr_1'\ {}_X||_Y\ tr_2'\} \\
\langle z\rangle \frown tr_1'\ {}_X||_Y\ \langle z'\rangle \frown tr_2' & = \varnothing \\
\langle x\rangle \frown tr_1'\ {}_X||_Y\ \langle y\rangle \frown tr_2' & = \{\langle x\rangle \frown tr \mid tr \in (tr_1'\ {}_X||_Y\ \langle y\rangle \frown tr_2')\} \\
 & \quad \cup\{\langle y\rangle \frown tr \mid tr \in (\langle x\rangle \frown tr_1'\ {}_X||_Y\ tr_2')\}
\end{array}
$$

A trace of the sequential composition $(TZE_1;\ TZE_2)$ is a sequence of events of TZE_1 up to the moment TZE_1 terminates by engaging in event $\checkmark$. After that the trace continues with sequence of events from TZE_2. This definition of sequential composition is known as *strong sequential composition* [6].

T9 $traces(TZE_1;\ TZE_2) = \{traces(TZE_1) \cap (\Sigma \backslash \{\checkmark\})^*\}$
$\cup \{s \frown t \mid s \frown \langle \checkmark \rangle \in traces(TZE_1) \wedge t \in traces(TZE_2)\}$

The trace model for recursion is a fixed point definition. Refer to the detailed discussion in [79].

T10 $traces(\mu X \bullet TZE) = \bigcup_{n \geq 0} traces(F^n(\text{STOP}))$

Process constructs like SKIP or interleaving or timed interrupt can be defined in terms of other primitive ones. The traces of process expressions involving those constructs, thus, can be deduced.

Example 3.2.1 (Traces of $ButtonPushing$**)** The following shows how we may compute the set of traces for a process expression:

$$
\begin{aligned}
& traces(ButtonPushing) \\
& \quad \widehat{=}\ traces(button?1 \rightarrow ([dim > 0] \bullet TurningOff \\
& \qquad \Box\ [dim = 0] \bullet TurningOn)) \\
& \quad \widehat{=}\ \{\langle button?1 \rangle \frown u \mid u \in traces([dim > 0] \bullet TurningOff \\
& \qquad \Box\ [dim = 0] \bullet TurningOn) \cup \{\langle\rangle\}\} && \text{– by } \mathbf{T3} \\
& \quad \widehat{=}\ \{\langle button?1 \rangle \frown u \mid u \in traces([dim > 0] \bullet TurningOff) \\
& \qquad \cup traces([dim = 0] \bullet TurningOn) \cup \{\langle\rangle\}\} && \text{– by } \mathbf{T6} \\
& \quad \widehat{=}\ \{\langle button?1, TurningOff \rangle, \langle button?1, TurningOn \rangle, \langle button?1 \rangle, \langle\rangle\} && \text{– by } \mathbf{T5}
\end{aligned}
$$

end

3.3 From TCOZ to Statecharts

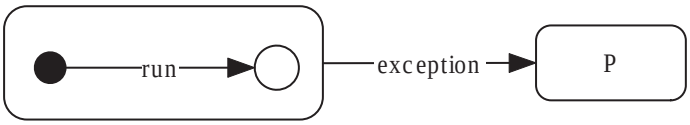
TCOZ is well suited for presenting complete and coherent requirement specifications that comprehensively model various viewpoints for complex systems. Given an integrated model, one can project it into consistent multiple views for specialized analysis. In this section, we are interested in one particular viewpoint projection - the intra-object control flow perspective.

3.3.1 Projection

The notion of Statechart originated from Harel [65]. A Statechart diagram is an important modeling notation in UML. It represents the behavior of entities capable of dynamic behavior by specifying its response to the receipt of events. Typically, it is used for describing the behavior of class instances. The key idea for using UML Statecharts to visualize TCOZ is that TCOZ processes (operations) are identified with states of UML Statecharts and TCOZ events/guards are identified with the state transitions. In the following, we present a set of projection rules, which defines the Statechart patterns for TCOZ process constructs.

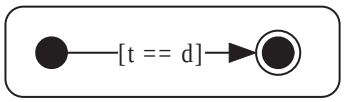
STOP is identified with a state without outgoing transitions. Thus, a system run reaching the state makes no further move unless the control is withdrawn from the composite state containing the state. SKIP is identified with a final state so that once the state is reached, the control is taken away from the composite state containing the state. Thus, the termination event ✓ is hidden. RUN is identified with a state where there is a self-looping transition for each and every event in the alphabet. Thus, the system may execute any sequence of events.

Example 3.3.1 (Visualization by Statechart) Given a process $(run \rightarrow \text{STOP}) \ \nabla_{exception} \ P$, the Statechart is generated as the following:

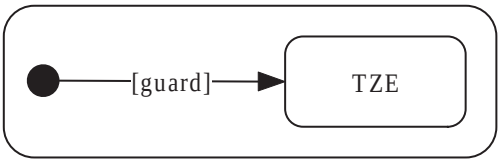


end

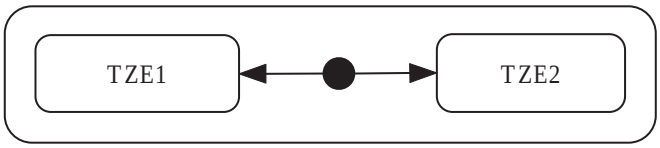
WAIT$[d]$ is identified with a composite state containing one initial state, i.e., pseudostate in UML terms, and one final state. The two states are connected via a transition guarded with the condition $t == d$, where t is local clock.



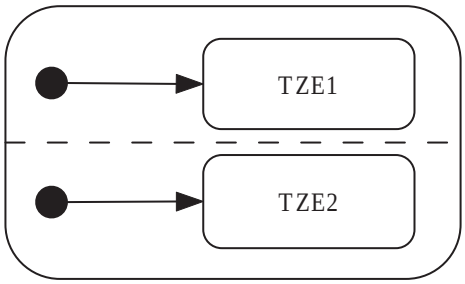
An operation is projected to a simple state. A process composed by sub-processes is projected to a composite state. This way, we preserve the hierarchal structure of the process expression. A state guard $([guard] \bullet TZE)$ is identified with a composite state containing an initial state and a state corresponding to TZE. The initial state is connected to the other state via a transition guarded with $guard$. Similarly, event prefixing $c.e \rightarrow TZE$ is identified with a composite state containing an initial state and a state corresponding to TZE. The two states are connected by a transition labeled with $c.e$ from the initial state to the other state.



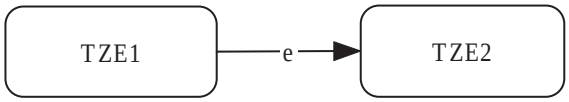
A choice $(TZE_1 \mid TZE_2)$ is identified with a composite state where there are one initial state and two states corresponding to TZE_1 and TZE_2. For external choice, TZE_1 and TZE_2 are often event prefixing or state guard, and hence the transitions from the initial state are often guarded.



Interleaving $(TZE_1 \mid\mid\mid TZE_2)$ is identified with a concurrent state where there are two independent sub-routines. Parallel composition in general $(TZE_1 \ {}_X\!\mid\mid_Y \ TZE_2)$ is identified with a concurrent state with additional synchronization barriers (which makes it a synch state).



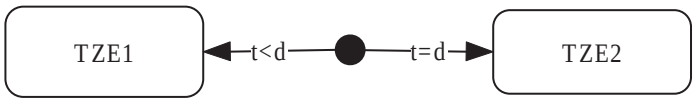
Interrupt $(TZE_1 \triangledown_e TZE_2)$ is identified with two states corresponding to TZE_1 and TZE_2 respectively, and a transition from the (edge of the) state corresponding to TZE_1 to the (edge of the) state corresponding to TZE_2 labeled with e. Thus, once event e is engaged, the control is taken from the state corresponding to TZE_1 (which is a composite state) and transferred to the initial state of the composite state corresponding to TZE_2.



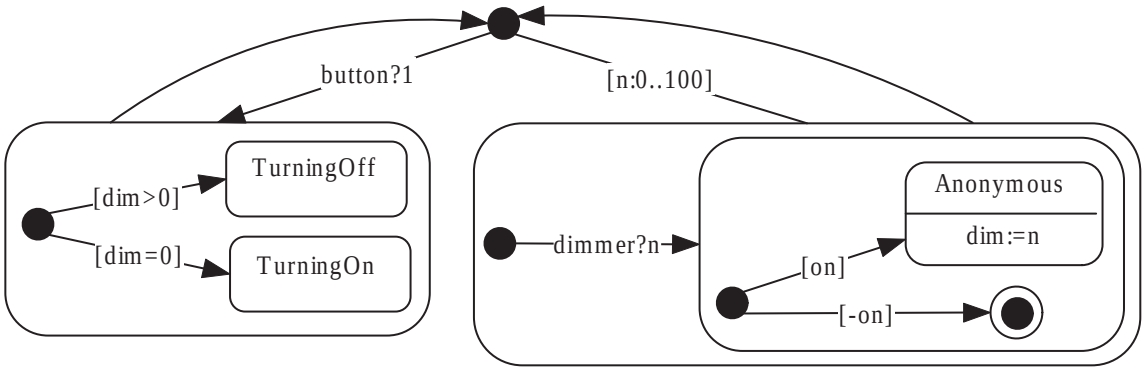
Sequential composition $(TZE_1;\ TZE_2)$ is identified with two states corresponding to TZE_1 and TZE_2, and a transition from the state corresponding to TZE_1 to the state corresponding to TZE_2 labeled with nothing. Thus, once the system reaches the final state in the composite state corresponding to TZE_1, the control is transferred to the state corresponding to TZE_2.

Recursion $(\mu X \bullet P(X))$ is handled by connecting all transitions leading to the state corresponding to X to the initial state of $P(X)$. Our projection is restricted to regular processes, and thus recursions which result in irregular processes are ignored.

Timeout $(TZE_1 \triangleright\{d\}\ TZE_2)$ is identified with a composite state where there are one initial state and two states corresponding to TZE_1 and TZE_2. The transition from the initial state to the state corresponding to TZE_1 (TZE_2) is labeled with $t < d$ ($t == d$). Thus, if d time units elapsed before the control moves out from the initial state, the control moves to the state corresponding to TZE_2.



Example 3.3.2 (Statechart for $ControlledLight$) The MAIN process in class $ControlledLight$ is visualized by the following Statechart:



It is recommended to define an operation schema for every state update. However, assignments permitted in the Timed CSP syntax may change the valuation of a state variable. Assignment, for example $dim := n$, is treated as an anonymous operation schema and projected to a simple state where the assignment is identified with the entry action. **end**

3.3.2 Automation

In this section, we discuss how our projection is automated. The work in [150] has used XML and XML schema to define a standard exchange format, named ZML, for Z-family languages (Z, Object-Z and TCOZ). An XML Schema file was created for describing the structure of the Z-family languages. It defines the contents of all elements, the order and cardinality of sub-elements, and data types of the elements, etc. It serves as a good starting point for building lightweight tools based on Z family language.

Example 3.3.3 (ZML) Figure 3.1 is a part of the $ControlledLight$ class model in ZML. The tag *name* identifies the name of the class, i.e., *ControlledLight*. The tag *inheritedClass* indicates the immediate super-class. The tag *state* encodes the state schema, which has been skipped for space

```
<classDef><name>ControlledLight</name>
    <inheritedClass><name>Light</name></inheritedClass>
    <state>...</state>
    <operation>
        <name>Main</name>
        <processExpr>
            <mu>N</mu>
                <processExpr>
                    <processExpr>
                        <simpleProExp>ButtonPushing</simpleProExp>
                    </processExpr>
                    <proConnSym>externalChoice</proConnSym>
                    <processExpr>
                        <simpleProExp>DimChange</simpleProExp>
                    </processExpr>
                </processExpr>
            </processExpr>
        </processExpr>
    </operation>
</classDef>
```

Figure 3.1: XML markup of class $ControlledLight$

saving. The tag *operation* defines a process with its name encoded in tag *name*. The tag *processexpr* encodes the computational logic of the operation. In this example, it is a recursion (a μ function) of a choice (indicated in a tag named *proConnSym*) between two process expressions encoded in the tags named *simpleProExp*. **end**

XMI (XML Metadata Interchange [130]) is an industry standard for storing and sharing object programming and design information, allowing developers of distributed systems to share object models and other metadata over the Internet. Three key industry standards, XML (eXtensible Markup Language), UML (Unified Modeling Language) and MOF (Meta Object Facility), are integrated in XMI. XMI marries the OMG and W3C metadata and modeling technologies. Rational Rose 2001 from OMG [124] which supports XMI can generate UML diagrams once it imports XMI documents, and it can also export XMI documents for any existing UML diagrams. This is very useful for our work. All we need is to generate the proper XMI documents from the TCOZ specification and make use of facilities offered by tools like Rational Rose for visualization and possibly code generating. The syntax definition of XMI for UML is specified in XMI 1.1 RTF UML DTD [130]. This DTD

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version='1.1' xmlns:UML='//org.omg/UML/1.3'>
    <XMI.header> ... </XMI.header>
    <XMI.content>
        <UML.Model>
             <UML.StateMachine> ... </UML.StateMachine>
        </UML.Model>
        <UML:Diagram> ... </UML:Diagram>
    </XMI.content\>
</XMI>
```

Figure 3.2: XMI structure

file defines all entities and XMI syntax signatures for UML. An XMI file validated by UML.DTD version 1.3 has the structure as in Figure 3.2. `XMI.header` contains general information like the UML.DTD version. `UML.StateMachine` is the most important part of `UML.content`, which contains information about the Statechart. `UML:Diagram` is used to display the UML diagrams. It contains the exact position of every displayable unit in the UML diagram.

The projection rules for translating TCOZ models (in XML) to UML Statecharts (in XMI) are implemented by a JAVA application. To systematically build the Statecharts, the projection takes place in stages. The first stage is *preparation*, during which the XML representation of the TCOZ model is fetched in and parsed class-by-class, operation-by-operation. The activities preformed are:

- Build up the operation table for each class and the variable table for each class.
- Associate each class with its corresponding super class. One class may have more than one super class and it may invoke operations defined in different super classes.
- For each operation, identify its $processexpr$ which is the tag identifying the computational logic for the operation. We check whether the operation is an operation schema. If it is, mark this operation as a simple operation. Otherwise, we identify the type of the $processexpr$. For each type of $processexpr$, gather relevant information for the type.

The second stage is *Generation*. For each active object, a new XMI file is created with the necessary header information. A top level composite state named 'Main' is added to the `Statemachine`,

which corresponds to the MAIN process. An initial state is added to the top-level composite state. Starting from the MAIN operation, we syntactically analyze the $processexpr$ and apply the proper projection rule to generate states and transitions. The challenge is that we may not know which projection rules could be used at some point. For example, if some other operation is invoked by MAIN, shall we model the called operation as a simple state or a composite state? (At this point, we may not be able to find out whether the called operation will consequently invoke other operations.) Our remedy is to model all called operations as composite states and later replace those trivial composite states by simple states.

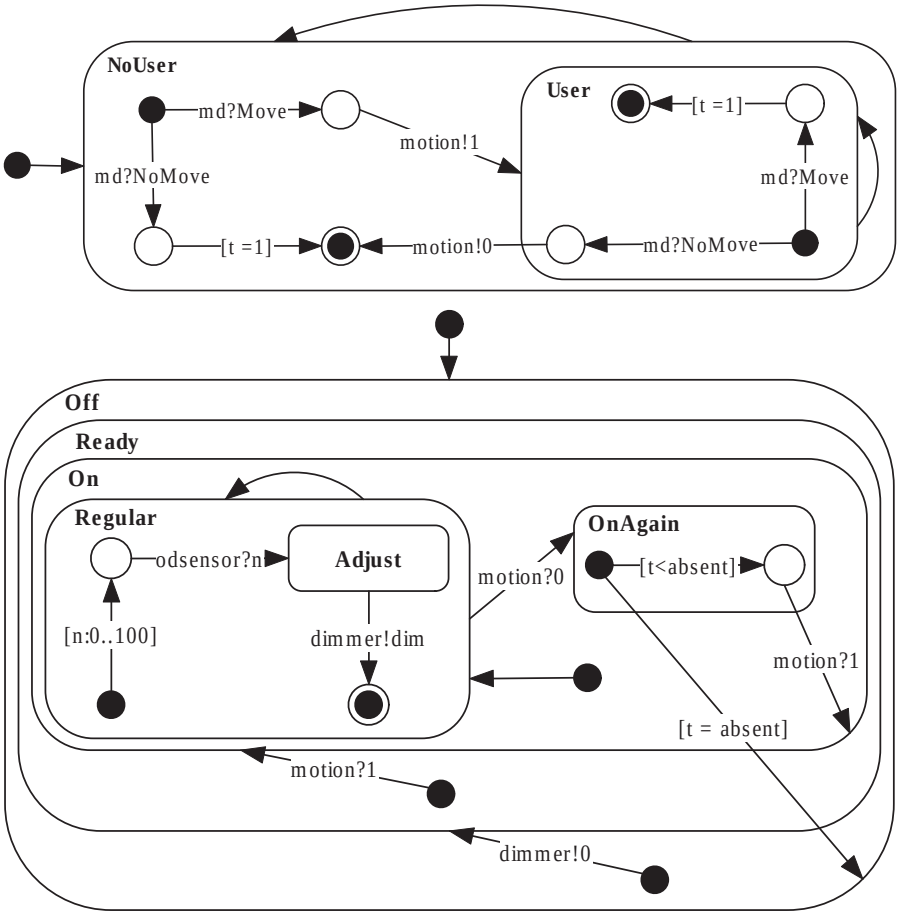
The third stage is *simplification*. After the Statechart model is generated, a number of activities take place to simplify the Statechart without changing its traces. For example, trivial composite states, i.e., composite states that have at most one sub-state, are removed. We also check the Statechart for violation of well-formedness rules. The last step is to *layout* the diagrams nicely. We calculate the exact positions of all the states, transitions and events/guards in a diagram. This is theoretically irrelevant but practically very important. The following formulæ are used to calculate the width and height of a composite state. Let W be the width, H be the height, M be the number of simple states in the composite state, N be the number of composite states in the composite state. $WSimple$ ($HSimple$) is the default width (height) of a simple state. $W_1, \cdots W_N$ are width for each composite state in this composite state. $H_1, \cdots, H_N$ is the height for each composite state in this composite state. S is the default horizontal space between states. K is the default vertical space between states. P is the width (or height) of an initial state and Q is the width (or height) of a final state.

$$W = max\{\left(\sqrt{M}+1\right) * (WSimple + S), W_1, W_2, \cdots, W_N\} + 4S + P + Q$$
$$H = (\sqrt{M}+1) * (HSimple + K) + (H_1 + H_2 + \cdots + H_N) + N * K$$

The calculation is done in a bottom-up manner because the size of the outer composite state depends on the size of the inner one. Once we know the width and height, we place simple states at the top ($\sqrt{M}$ simple states per row) and composite states at the bottom (one per row). Lastly, the XMI file is generated.

Example 3.3.4 (Statecharts for $MotionDetector$ and $RoomController$) The following are the Statecharts generated from the LCS specification (the first one for class $MotionDetector$ and the second

for class $RoomController$). Simplification has been involved to make the Statecharts compact.



end

3.4 From TCOZ to Scenarios

In this section, we are interested in another viewpoint projection - the communication and interaction perspective. MSC [81] is a popular graphical notation for presenting interactive viewpoints of a system. It is termed as sequence diagrams in the UML framework. We investigate the projection from TCOZ (trace models) to MSC (process models). By identifying a set of traces with MSC, the

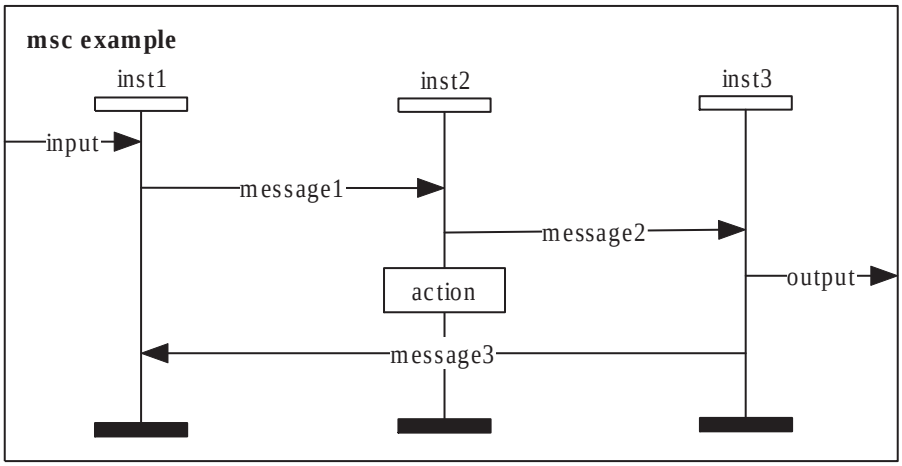


Figure 3.3: Basic Message Sequence Chart

cause and effect relations between distributed events in concurrent systems are captured graphically. A prototype projection tool is developed for generating MSCs automatically. By inserting class invariants and operation constraints (as assertions) into the generated MSCs (execution scenarios), system testing requirements can be obtained.

3.4.1 Message Sequence Chart

The language MSC is standardized by the International Telecommunication Union (ITU). It provides a means for visualization of the interaction of system components. The core of MSC is called the Basic Message Sequence Chart (BMSC), which concerns communications and actions only. Then, additional basic concepts like process creation, termination, time handling, incomplete message events and conditions are added. Later, more complicated constructs are introduced. They are inline expressions, MSC reference expressions and High-level Message Sequence Chart (HMSC), which enrich MSC with intricate possibilities of describing complex systems.

Example 3.4.1 (Basic MSC) Figure 3.3 is an example of a BMSC. Each vertical line represents an active component (Z.120 terminology, an instance) in the system. The frame (Z.120 terminology,

parallel frame) represents the environment. Instances can interact with other instances by sending messages, e.g., $message1$, $message2$, $message3$. A message originated from the frame represents an input from the unspecified environment, e.g., $input$. Similarly, a message targeting the frame is an output to the environment, e.g., $output$. The square labeled with $action$ is a local action performed by instance $inst2$. **end**

The timing information is captured by the following two rules and their transitive closure: for each message passing, the message output event precedes the corresponding message input event and for each vertical line representing an instance, the time progresses from top to bottom. The two rules and the transitive closure define a partial event ordering relation, which captures the semantics of BMSC [109].

High-level MSC can be constructed incrementally by referencing an MSC using its name (or equivalently using inline expressions). MSC can be combined vertically, horizontally or alternatively.

Example 3.4.2 (High-level MSC) The chart in Figure 3.4 is a simple example of an HMSC. The triangle at the top represents the starting point. The one at the bottom represents the ending point. Each rounded rectangle abstracts an MSC. The semantics of the HMSC is captured by the process expression $(A \circ C)^{\circledast} \circ (A \circ B)$, where $\circ$ denotes sequential composition and $\circledast$ denotes infinite iteration. **end**

Various constructors for composing MSCs are: *alt* for choices, *seq* for sequential composition, *par* for parallel composition, *opt* for optional, *exc* for exception and *loop* for iteration. Precise semantics are developed for these key words.

Definition 6 *An MSC reference expression is defined as the following:*

$$
\begin{array}{rcll}
MRE & ::= & ref\langle\langle NAME\rangle\rangle \mid \epsilon \mid \delta & \text{– primitives} \\
 & \mid & (\_ \mp \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– delayed choice} \\
 & \mid & (\_ \,||\, \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– delayed parallel} \\
 & \mid & (\_ \circ \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– sequential composition} \\
 & \mid & (\_)^{\circledast}\langle\langle MRE\rangle\rangle \mid (\_)^{\infty}\langle\langle MRE\rangle\rangle & \text{– iteration}
\end{array}
$$

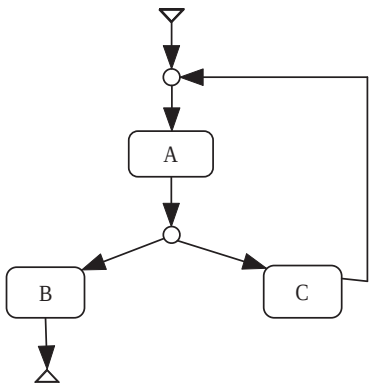


Figure 3.4: High-level Message Sequence Chart

An HMSC can reference other HMSCs or BMSCs by their names. The two most primitive constructs are δ and ϵ. The former does nothing at all and the latter terminates immediately. The structural operator $delayed\ choice$ is written as $\mp$. Graphically, it is a sub-chart marked with the keyword *alt*. The $delayed\ parallel$ is written as $\|$. The notion of sequential composition in MSC is referred as $weak\ sequential\ composition$, denoted as $\circ$. Given sequential composition of two MSCs (say m_1 and m_2), interactions over shared instances in m_2 is delayed until interactions in m_1 completes. However, the execution of actions over instances not in m_1 from m_2 is allowed before m_1 has the option to terminate. The iteration operator $\circledast$ is defined as any number of sequential composition of a chart, whereas ∞ is the unbounded repetition of a chart.

A number of semantic models have been developed for MSC. Examples are the operational semantics based on process algebra [6, 81], Petri nets [74], automata, etc. The informal MSC semantics and formal process algebra semantics presented in [81] are adopted in this thesis. In [81], semantics of various constructs of MSC are defined by sets of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and conclusion are of the form $s \xrightarrow{a} s'$ or $s \downarrow$ for arbitrary $s, s' \in MRE$ and $a \in A$, where A denotes all events represented by atomic actions in MSC, i.e., message input, message output, local action and timer events. For instance, no deduction rule is associated with δ because it does nothing. The only rule associated with ε is $\varepsilon \downarrow$, i.e., termination. The semantics of $\mp$ is captured by the following rules:

$$\frac{x \downarrow}{x \mp y \downarrow}\,[\,DC1\,] \qquad \frac{y \downarrow}{x \mp y \downarrow}\,[\,DC2\,] \qquad \frac{x \xrightarrow{a} x', y \overset{a}{\not\rightarrow}}{x \mp y \xrightarrow{a} x'}\,[\,DC3\,]$$

$$\frac{x \overset{a}{\not\rightarrow}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}\,[\,DC4\,] \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}\,[\,DC5\,]$$

The rules $DC1$ and $DC2$ express that the delayed choice of the two processes has the option to terminate if and only if at least one of the alternatives has this option. $DC3$ and $DC4$ express that the delayed choice will behave as one of the options given that some initial event of this option takes place. $DC5$ captures the idea that in case both of the alternatives are enabled, the choice is delayed. The rest of the constructs are similarly defined [81].

3.4.2 Visualizing Traces

Given an active object, we can identify the set of possible traces by applying the $traces$ function to the MAIN process. A trace can be transformed to a Basic MSC by identifying operation schemas in TCOZ with MSC local actions and identifying channel communications in TCOZ with message passing in MSC.

A TCOZ event is either an update event, a simple synchronization, a channel communication, or a termination event, or a $wait(d)$ event. Update events are distinguished from the others as they do not require cooperation of the environment. They perform on a single instance. An MSC local action is defined as an orderable single instance event requiring no cooperation from the environment. Update events are identified with local actions in MSC. Synchronization and channel communication do require cooperation either from the environment or other processes. Channel communications in TCOZ are identified with message passing in MSC (message passing with a 0-capacity buffer). The special $wait$ event is identified with the timer event in MSC. In particular, it is identified with a timer set event in MSC and consequently associated with a timeout or reset event.

Example 3.4.3 (MSC from *ControlledLight*) Figure 3.5 is a Basic MSC visualizing a scenario of LCS. Initially the light is off. Starting with MAIN, the process $DimChange$ is executed. A message

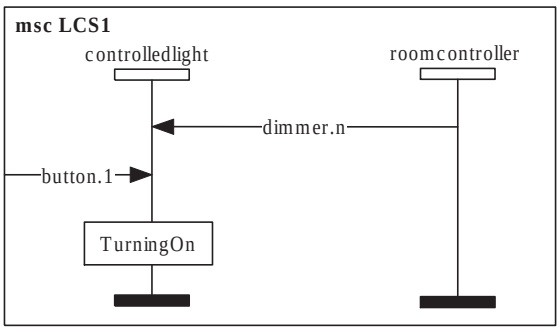


Figure 3.5: Visualizing traces

$dimmer?n$ from *RoomController* to *ControlledLight* takes place. Because on is false, no action is taken. Process $ButtonPushing$ is then activated by a message input event from channel $button$. Action $TurningOn$ is invoked. After that, no event occurs. **end**

Identifying the traces of a parallel composition of multiple processes, for example the MAIN process of the LCS class, is computationally expensive. In our prototype, a set of traces for each object (*ControlledLight*, *MotionDector*, *RoomController*) is generated independently. Traces from different objects sharing the same sequence of communication over the shared channels are then identified. Lastly, the corresponding communication is connected and visualized using MSCs. This way, we make use of the full power of MSC's partial ordering property, i.e., to leave the order of single instance events from different instances unspecified. Thus, one MSC is capable of representing a set of scenarios.

Example 3.4.4 (MSC from LCS) In the LCS class, active object m (the motion detector) shares the channel $motion$ with the active object r (the room controller). Two matching traces, one generated from MAIN in class $MotionDetector$ and one from MAIN in class $RoomController$, must contain the same sequence of events on channel $motion$.

$$\langle md?NoMove, wait\ 1, md?Move, motion!1, md?NoMove, motion!0\rangle$$
$$\langle dimmer!0, motion?1, odsensor?n, Adjust, dimmer!dim, motion?0\rangle$$

The above are a pair of matching traces. This interaction is visualized as in Figure 3.6. **end**

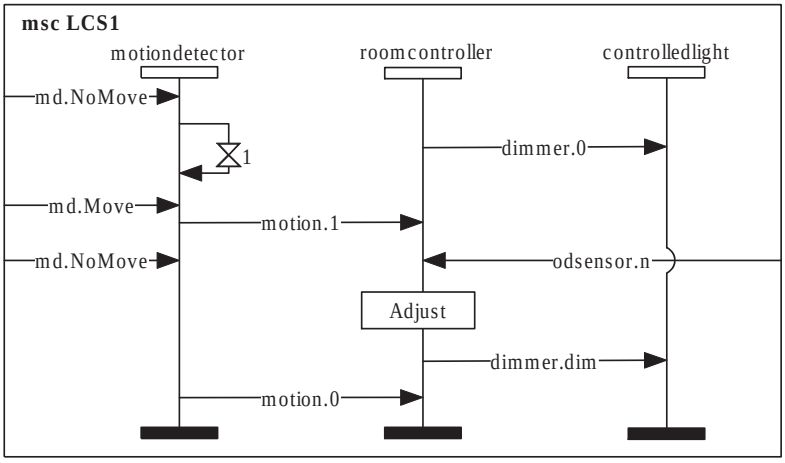


Figure 3.6: Scenario of Light Control System

3.4.3 Visualizing Process Expression

Due to unbounded recursion (iteration) and non-determinism, the set of traces (and therefore the generated BMSCs) for complex systems could be numerous or even infinite[1]. HMSC offers various constructive operators to compose MSCs in a hierarchical, iterating and nondeterministic way. So does CSP. Thus, it is natural to link process constructs in TCOZ with constructs in HMSC so that we may visualize multiple or even infinite scenarios using a single chart.

The body of a TCOZ class is essentially a system of simultaneous equations defining a collection of operations (processes). Each equation consists of a name and a TCOZ process expression. A TCOZ class is identified with an MSC document, which consists of a set of MSCs. A TCOZ process expression is identified with an MSC. A TCOZ process reference is identified with an MSC reference.

The trace model of an MSC process expression is constructed according to the operational semantics of MSC defined in [81]. Let function $traces : MRE \rightarrow \mathbb{P}\,\Sigma^*$ return the traces of the process

[1] In the LCS case study, 600+ traces are generated if we unfold each recursion 5 times.

expression. Let function $\mathcal{P} : TZE \rightarrow MRE$ be the projection function from TCOZ to MSC. A TCOZ process expression is projected to an MSC process expression if they are trace-equivalent.

$$\forall\, s : TZE;\ t : MRE \bullet \mathcal{P}(s) = t \Rightarrow traces(s) = traces(t)$$

STOP means deadlock and does nothing. SKIP performs no action except termination. Two basic constants, written as δ and ε, play the same role in the process semantics of MSC. Thus, STOP is identified with δ and SKIP is identified with ε. Graphically, SKIP is drawn as an empty MSC.

$$traces(\delta) = \{\langle\rangle\}$$
$$traces(\varepsilon) = \{\langle\checkmark\rangle, \langle\rangle\}$$

According to the deduction rules associated with the delayed choice, a trace of $x \mp y$ is either a trace of x or y. Therefore, the choice operators in TCOZ are projected to delayed choice in MSC. Graphically, a choice in TCOZ is drawn as an MSC sub-chart marked as *alt*.

$$traces(x \mp y) = traces(x) \cup traces(y)$$

Example 3.4.5 (HMSC from $ControlledLight$**)** Figure 3.7 shows the MSCs generated from the MAIN process in class $ControlledLight$. The choice between $DimChange$ and $ButtonPushing$ is captured by the delayed choice in the bottom chart, indicated by marking the sub-chart with the keyword *alt*. Recursion is visualized as an infinite iteration in HMSC. **end**

Sequential composition in TCOZ is best described as $strong\ sequential\ composition$, i.e., no action from the later process can be executed before the earlier one has the option to terminate. The sequential composition $\circ$ that composes two MSCs vertically is described as $weak\ sequential\ composition$. It allows execution of actions from the later chart before the earlier one has the option to terminate. However, if two MSCs involve only events on the same instances, the two notions are identical. Sequential composition in TCOZ is identified with sequential composition in MSC. Graphically, sequential composition of MSCs on the same instances is captured by putting the MSCs one below the other.

MSC has a key word *exc* for representing exceptions, however there is no formal rules defined in [81] for it. Following the same style, we define the deduction rules for *exc* (written as $\triangledown_m$) as follows.

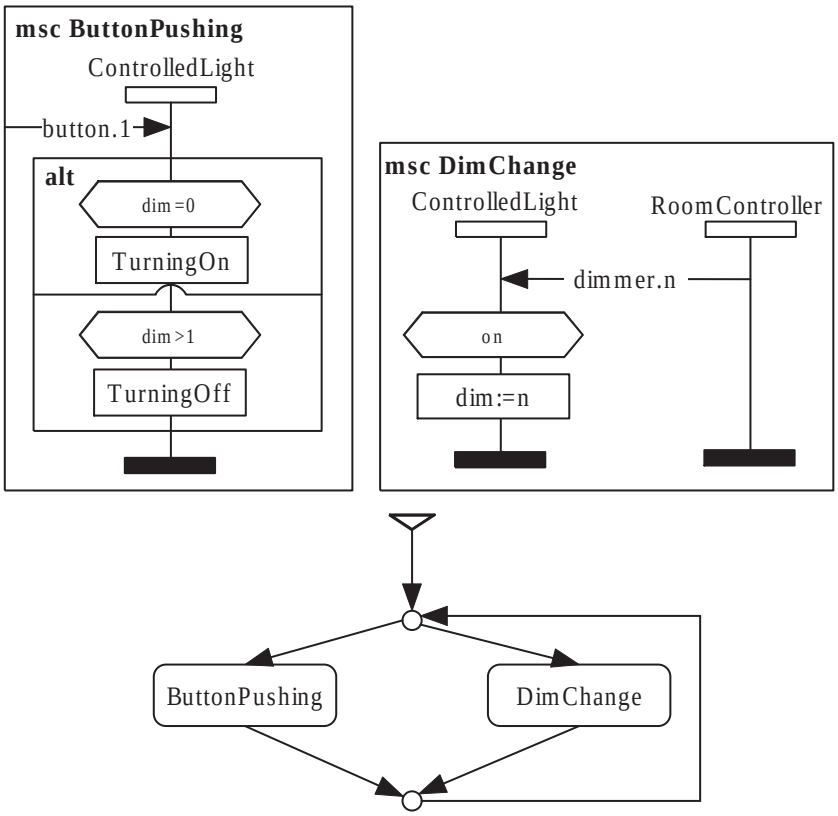


Figure 3.7: Scenarios of class $ControlledLight$

$$\frac{x \downarrow}{x \triangledown_m y \downarrow} \qquad \frac{x \xrightarrow{a} x', y \overset{a}{\not\rightarrow} y'}{x \triangledown_m y \xrightarrow{a} x' \triangledown_m y} \qquad \frac{y \xrightarrow{a} y'}{x \triangledown_m y \xrightarrow{a} y'}$$

In the process $X \triangledown_e Y$, any time e takes place, the control is withdrawn from X and transferred to Y. Interrupt in TCOZ is identified with $\triangledown_m$ in MSC with e as the initial event of the interrupting process.

Example 3.4.6 (Interrupt in MSC) Figure 3.8 presents the MSC generated from process expression On in $RoomController$. The event $motion.0$ is projected to the first communication in the down portion of the sub-chart. **end**

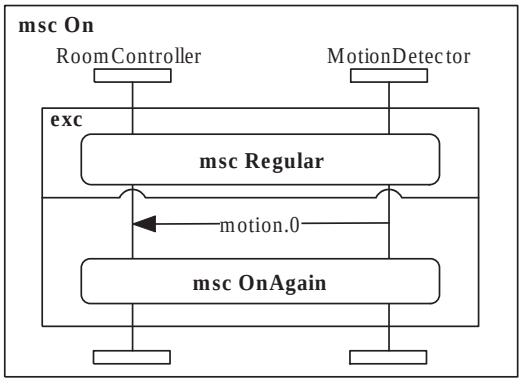


Figure 3.8: Interrupt in MSC

Delayed parallel composition defines the interleaving operator, i.e., no synchronization is required and processes can interleave freely. Thus, interleaving in TCOZ ($|||$) is identified with delayed parallel composition in MSC. In TCOZ, all dynamic interactions between active objects must take place through the CSP channel communication mechanism. Graphically, given two MSCs (MSC_1 and MSC_2), the parallel composition is constructed by putting the MSCs in the same parallel frame and connecting corresponding message output and message input events.

Besides the projection rules above, other constructs in TCOZ can be projected to MSC indirectly. For instance,

$$P \rhd \{t\}\ Q = P \Box (\textsc{Wait}\ \ t;\ Q)$$

By identifying external choice with MSC delayed choice and $\textsc{Wait}\ \ t$ with timer events, timeout can be identified with a delayed choice between the MSC for P and the MSC for Q with a timeout event as the initial event of Q. Moreover, TCOZ recursion can be resolved as iteration and interpreted by a sequence of sequential compositions. TCOZ state-guard is identified with local condition in MSC.

Example 3.4.7 (Timeout in MSC) Figure 3.9 shows the MSC visualizing the process $OnAgain$ in the LCS example. **end**

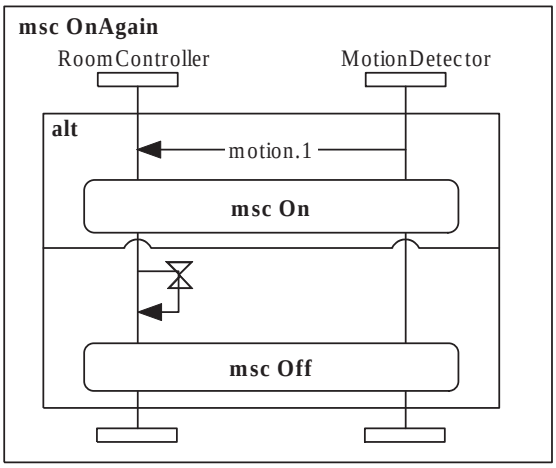


Figure 3.9: Timeout in MSC

3.4.4 Automation

The projection is automated by employing XML/XSL technology adopting a similar strategy as in Section 3.3.2. MSC offers a standard text representation for the graphical notations. Thus, we developed an automatic transformation tool to project TCOZ models (in ZML) into MSC (in standard text format). Building on the strength of ZML, our tool makes use of XML parser Xerces [59] to extract information from TCOZ specifications. The mechanized projection is achieved by first implementing a ZML parser, which takes in a specification model in ZML and builds a virtual model in the memory. This ZML parser can be reused for other projection tools, e.g., the transformation from TCOZ to Timed Automata for timing analysis (refer to Section 4.2).

A trace generation module is built to automatically generate all possible traces from the specification, and each trace is transformed to a BMSC by syntax rewriting. In the case of unbounded recursion, users are asked for the number of times to resolve the iteration. An MSC interface is built according to the MSC document structure, e.g., each MSC document contains multiple MSCs and each MSC contains one or more instances. A transformation module is built to get information from the ZML parser, apply the proper projection rules and feed the outcome to the MSC interface.

The projection rules are used as a design document and guide the construction of the algorithm in the implementation. The outcome of our transformation tool is Z.120 standard text representation of MSC, which is ready to be taken as inputs for various tool supports for MSC. The same strategy can be applied for implementing various transformation tools. For example, for the timing analysis purpose, a TCOZ specification can be transformed into Timed Automata, the same ZML parser can be reused and we only need to build a Timed Automata interface and a new transformation module (refer to Section 4.2).

Example 3.4.8 (Text representation of MSC) The following is the HMSC in standard text format, generated from operation MAIN in class *ControlledLight*:

```
msc ControlledLight;
    instance i1;
        loop begin;
        alt begin;
            reference ButtonPushing;
        alt;
            reference DimChange;
        alt end;
        loop end;
    endinstance;
endmsc;
```

end

In our prototype, we allow generation of test cases by adding assertions to the generated MSC. Test requirements can be used to develop test cases, test oracles and test drivers in a system development. Specification based testing can play an important role in software engineering [127, 142]. Our goal is to support automatic generation of test requirements from TCOZ specifications. Starting with an HMSC, one can expand it into a set of BMSCs. For recursions, at least one iteration should be covered by the expanded BMSCs. Upon creation of an instance, the TCOZ class initial state condition is instrumented as an assertion at the start of the BMSC. For each instance in the system, TCOZ class invariants are instrumented as assertions before and after every local action on the BMSC instance. The pre/post-conditions of TCOZ operations are projected to assertions at the entry/exit of the corresponding MSC actions.

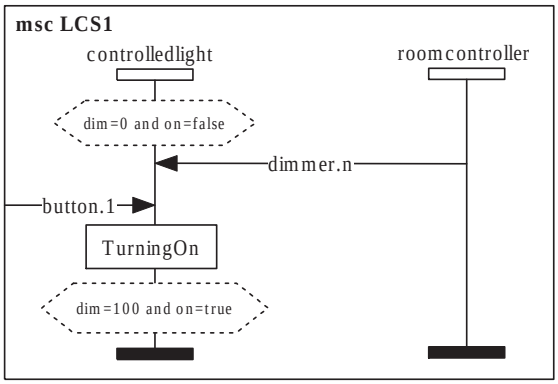


Figure 3.10: Test case

Example 3.4.9 (Test case) A test case generated from class $ControlledLight$ is captured in Figure 3.10 (the BMSC in Example 3.4.3 with assertions). Assertions are placed in the dash-lined box. The first assertion ensures the INIT schema is satisfied after creation of the $ControlledLight$ instance. The second one asserts the post-condition of the operation $TurningOn$. In this example, the pre-condition of the operation is simply true. **end**

Systematic test case generation allows specification-based testing of system designs. For instance, we may design a system using languages and notations which allow mechanized generation of executable codes, whilst document functional and dynamic system requirements using TCOZ specifications. Once executable codes have been generated from the design model, test cases generated from TCOZ model may be used to systematically validate the code. One design language of special interest is Communicating Transaction Processes (CTP [129]). CTP has been recently introduced by Roychoudhury and Thiagarajan. It is an MSC-based formalism targeting reactive embedded systems. Given a network of communicating sequential processes that synchronize on common actions, the key idea of the CTP model is to refine each common action into a set of sequence charts, each with a precondition and a postcondition. The key feature of the CTP model is that it yields an executable specification, e.g., SystemC program [61]. Thus, we may formally specify system requirements using TCOZ specifications, whilst design the system as CTP models. A finite set of test

cases is generated from the TCOZ specification. We may then guide and validate the execution of the generated SystemC module using one test case at a time. By integrating the test case generation and code synthesis algorithms, we may achieve on-the-fly testing. A full investigation is left to the future work.

3.5 Summary

Visualization is an intuitive complementary interplay between logic-based formalisms and visual formalisms. It is often theoretically lightweight, e.g., a simple trace model is sufficient for the soundness discussion, but its practical implication is promising. Because it allows easy visualized of essential facts, it may extend users of logic-based formalisms to system engineers without relevant mathematical background. Visualization may promote the usage of logic-based formalisms in industry because it links logic-based formal notations with well-accepted diagrammatic languages. In order to link formalisms which vary vastly in syntax, we have to look at the semantics behind the intuition of the language constructs. The work on visualization thus help us to deeply understand the similarities and differences between different modeling languages.

There have been attempts to connect formal specifications with graphical notations, some of which are evidenced in [12, 31, 34, 114]. A number of these works have been focused on formalizing graphical notations using logic-based formalisms. For instance, Bolton and Davies [12] have given a process semantics in CSP for UML activity diagrams. They use the process semantics to demonstrate the consistency of the object model. Instead of solving the consistency problem of diagrams, our work in this chapter aims at benefiting logic-based formalisms by connecting them to popular graphical notations. Brooke and Paige developed a tool-supported graphical notation for Timed CSP [21]. The difference between Brooke and Paige's approach and ours is that we use existing popular graphical notations instead of creating new ones. In [149], Dong *et al* visualized TCOZ models with UML class diagrams. Our work focuses on dynamic behaviors of objects. Ng and Butler [114] have developed a tool for visualizing CSP in UML for both the static architecture and the dynamic behaviors. In our approach, we are particularly interested in capturing intra-object

and inter-object dynamic behaviors. The characteristics of our work are that first our visualization has been based on a rather complicated specification language, which makes it challenging as well as valuable, and secondly our work has been connected to other practical problems like test case generation.

Chapter 4

Verification

The executioner's argument was,
that you couldn't cut off a head
unless there was a body to cut it off from.
- *Alice's Adventures in Wonderland*, Lewis Carroll

Formal verification aims at establishing properties of system designs using logic, rather than just testing or informal arguments. It involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove that the implementation satisfies the specification. Formal verification reveals inconsistency of the specification and thus improves the reliability of the product. The notion of model checking [28] has been widely accepted as a successful means of formal verification. Model checking is a method for formally verifying finite-state concurrent systems. The technique has been applied to a wide range of complex industrial systems. Formal checking has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains errors, model checking will produce a counterexample that can be used to pinpoint the source of the error.

Mature verification mechanism based on model checking has been developed for quite a number

of specification languages, e.g., FDR for CSP [58], UPPAAL for Timed Automata [9]. Building verification mechanisms for newly designed specification languages from scratch is time consuming and repetitious. All the code for timely efficient verification needs to be implemented, for instance, partial order reduction, data abstraction, etc. An inexpensive yet effective way of verification is to reuse existing verification mechanisms. A semantics preserving transformation from the language to the language supported by the verification mechanism is essential for the task. In this chapter, we demonstrate formal verification based on transformation techniques.

4.1 Model Checking Live Sequence Charts

In this section, we show that tool support for logic-based formalisms can be reused to verify visual formalisms. MSC [81] is widely used to describe scenarios that capture communication between processes or objects. It is used in the early stages of system development. It has found its way into many methodologies [81, 154]. However, MSC (both BMSC and HMSC) suffers from the rather weak partial-order semantics that makes it incapable of capturing many kinds of behavioral requirements. Moreover, MSC only captures example runs of the system and thus it is not suitable to specify complete system behaviors. The notion of Live Sequence Chars (LSC) was introduced by Damm and Harel [32] to overcome the shortcomings of MSC by adding liveness. LSC extends MSC with constructs to distinguish scenarios that must happen from scenarios that may happen, conditions that must be fulfilled from conditions that may be fulfilled, etc. Together with the notion of symbolic objects and various high-level operators like bounded loop, if-then-else, LSC may well be used to specify complicated inter-object system requirements. A software package named *Play-Engine* has been developed by Damm and Harel to interactively “play-in” and “play-out” scenarios [70]. However, *Play-Engine* does not support automatic verification of LSC. We believe that it is important to expose inconsistencies of system requirements in the early stage of system development. One effective approach to verify LSC models is via reusing existing mature model checkers instead of building new ones from scratch.

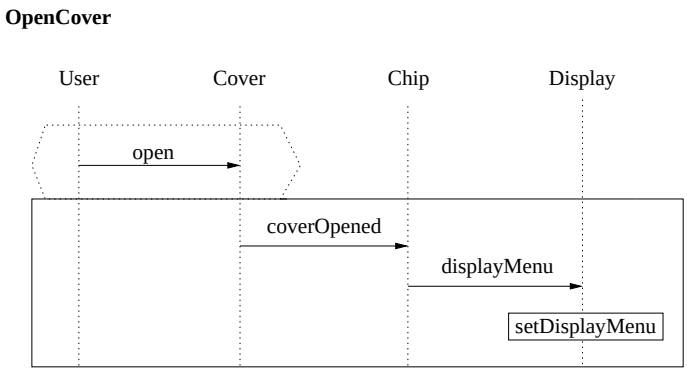
Semantically, system behaviors specified by LSC correspond to CSP’s traces and failures. This

close semantic correspondence makes FDR a potential model checker for LSC. The challenge is to construct semantics preserving CSP models from LSC models systematically. In this section, we investigate theoretical relations between LSC and CSP and develop an interpretation of LSC in CSP. The investigation is more than of theoretical interest. Its practical implication is that tool support for CSP can be reused to validate LSC models. In particular, FDR, a well-known CSP model-checker, is used for the verification.

4.1.1 Live Sequence Chart

There are two kinds of charts in LSC. Existential charts are mainly used to describe possible scenarios of a system in the early stage of system development, i.e., the same role played by MSC except that existential charts are scoped. 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 with a pre-chart, which serves as the activation condition of the main chart. Whenever a communication sequence matches the pre-chart, the system must proceed as specified by the main chart. A system run may activate a universal chart more than once and some of the activations might overlap [108].

Example 4.1.1 (Universal chart) The following is a universal chart as part of the mobile phone specification:



This scenario *OpenCover* illustrates the interaction between the objects when the $user$ opens the $cover$. Once the cover is opened by the user, the main chart is activated. The $chip$ is notified that the $cover$ is opened. It then requests the $display$ to display the menu. Lastly, the $display$ carries out a local action $setDisplayMenu$ to initialize the menu screen. **end**

Let $\mathcal{C}$ be the set of all possible charts. Let $\mathcal{E}$ be the set of existential charts. Let $\mathcal{U}$ be the set of universal charts. In this work, we assume that an LSC specification, denoted as $\mathcal{S}$, consists of a set of universal charts and existential charts. Throughout the section, c, e, u are used to denote a chart, an existential chart, a universal chart respectively. Let $\mathcal{B} \subset \mathcal{C}$ be the set of basic charts, i.e., Basic-MSC [81]. Let Σ be the set of all possible events. Σ is partitioned into two groups, communication messages M, e.g., $coverOpened$ in Example 4.1.1 and local actions A, e.g., $setDisplayMenu$. A communication event $m : M$ is followed by '?' if it is an input event or '!' if it is an output event. A local action $a : A$ may be an assignment or a (local or external) function call. Each chart c is associated with a set of visible events, $\Sigma_c \subset \Sigma$. Only events visible to a chart are constrained by the chart. A chart typically consists of multiple instances (for instance, $User$, $Cover$, $Chip$ and $Display$), which are represented as vertical lines graphically. Let $instances : \mathcal{C} \rightarrow \mathbb{P}\, Instances$ be the function returning the set of instances appearing in the chart. Along with each line, there are a finite number of locations. A location carries the temperature annotation for progress within an instance. Intuitively, locations can be thought as the joint points of instance lines and message lines. In the following, we use i to denote an instance, l_i to denote a location on instance i, l_i^0 to denote the first location on instance i and l_i^{max} to denote the very last location on instance i. We write the next location of l_i^k along instance i in the same chart as l_i^{k+1}.

A location may be labeled as either cold or hot. A hot location means that a system run reaching this location has to move beyond. A system run may stay put at a cold location forever. Similarly, 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 violation of a cold condition terminates the chart. A location is labeled with a finite number of events (more than one if it is a co-region) and at most one condition. Let $Location$, $Condition$ be the set of all possible locations and conditions respectively. Function $label : Location \rightarrow \mathbb{P}\Sigma$ labels a location with a finite number of messages and local

actions. Function $cond : Location \rightarrow Condition$ labels a location with a condition. If there is no condition associated with the location, it returns $true$. Function $eval : Condition \rightarrow \mathbb{B}$ evaluates a condition against the current valuation of the variables. Function $temp : Condition \rightarrow Cold \mid Hot$ tells the temperature of a condition. Function $temp : Location \rightarrow Cold \mid Hot$ tells the temperature of a location.

Example 4.1.2 (Mobile phone specification) The universal charts in Figure 4.1 and the one in Example 4.1.1 constitute a self-containing set of scenarios, which specify a mobile phone specification. This example is partially inspired by the phone system specification presented in [71]. The system consists of six participating objects, a $user$, the $cover$, the $display$, the $speaker$, the $chip$ and the environment where the incoming calls are from. Figure 4.1 illustrates scenarios of the system besides *OpenCover*, i.e., the user closes the cover, an incoming call arrives and the user picks up the phone and talks. All vertical lines in the charts are dotted, which means that all locations along the lines are cold and, therefore, the system may pause at any point of execution forever. This is possible because unexpected events like the battery runs out or the system breaks down may occur at any time. The set of visible events for each chart are exactly those appearing in the diagram except the scenario *Talk*. The message $close$ from the user to the cover is forbidden in the scenario *Talk*, i.e., in order to carry out the scenario successfully, the user should not close the cover before the scenario completes. **end**

LSC also supports advanced MSC features like co-region, hierarchy, etc. Moreover, symbolic instances and messages are used to group scenarios effectively. For a detailed introduction on a complete list of features of LSC, refer to [70]. LSC is far more expressive than MSC, which makes it capable of expressing complicated scenario-based requirements. 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

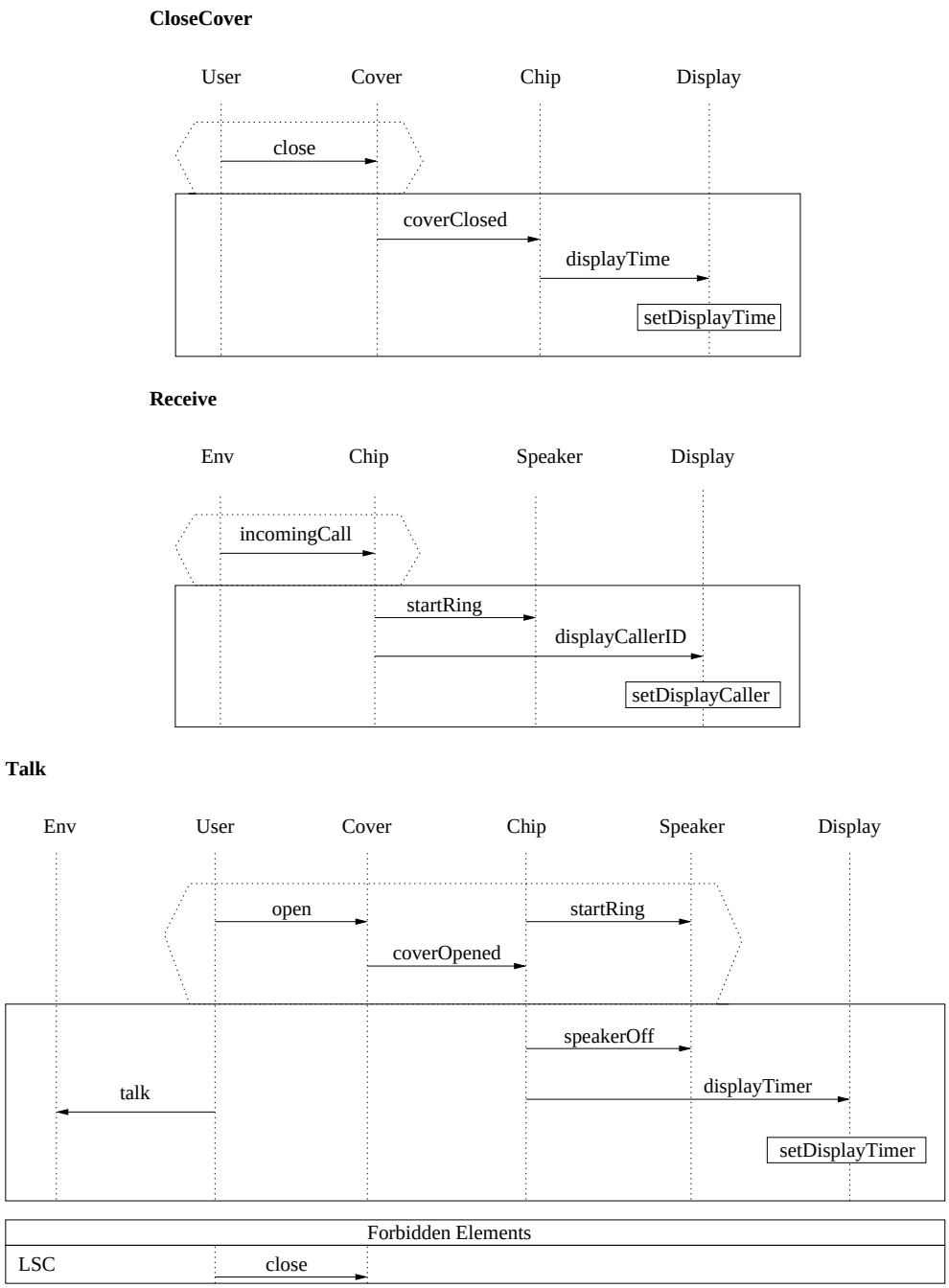


Figure 4.1: Mobile phone system scenarios

that the temperature of messages has no semantic meaning [70]. Thus, in the following discussion, the temperature of messages is discarded.

4.1.2 Semantics of Live Sequence Charts

The semantics of LSC is briefly discussed in [32] using skeleton automata and program-like pseudo-codes. Only basic charts and pre-charts have been covered. To develop a sound interpretation of LSC in CSP, a complete semantics is essential. This section is devoted to a trace-based denotational semantics of LSC, which conforms to the original semantics in [32]. Our semantics completes theirs by defining precisely the traces of a set of universal charts and existential charts.

We assume that all conditions are distributed because we believe that shared-condition is a problematic feature of LSC. In 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. Nor does it conform to the Dijkstra's principle of *loose coupling* [39]. Objects in distributed systems have their own state space (local variables) and all communication between objects would be via messages. We remark that shared-condition can be (partially) supported by rewriting it to a set of distributed conditions with extra synchronization. For simplicity, in this section we also assume that no co-region is allowed and all messages are synchronized. There is nothing interesting about co-region except that it complicates the discussion. Asynchronous message passing is supported by explicitly modeling the behavior of the buffers, e.g., First In First Out (FIFO). A consequence of this assumption is that a message loss is captured as an infinitely long delay of the forwarding by the buffer instead of a $lost\ message$ symbol. A hidden assumption is that the size of the communication buffers is finite.

The semantics of a basic chart b is defined to consist of all runs compatible with the partial ordering relation induced by b and its annotations. We define an automaton interpretation of b completing the skeleton automata in [32] and then define the languages of b based on the automaton. A chart induces a partial order over the events.

Definition 7 *The partial order is defined as the smallest binary relation* $\ll$: $Location \leftrightarrow Location$

satisfying the following axioms and closed under transitivity and reflexivity.

$$\forall\, l_i^k : Location \bullet l_i^k \ll l_i^{k+1} \qquad \text{– vertical}$$
$$\forall\, l_1, l_2 : Location \mid l_1 \neq l_2 \bullet$$
$$\exists\, m : M \bullet m! \in label(l_1) \wedge m? \in label(l_2) \Rightarrow l_1 \ll l_2 \qquad \text{– horizontal}$$
$$\exists\, m : M \bullet m! \in label(l_1) \wedge m? \in label(l_2) \Rightarrow l_2 \ll l_1 \qquad \text{– synchronous}$$

The first axiom states that along each vertical line time progresses from top to bottom. The second axiom states that message output event must precede the corresponding message input event. The third handles synchronous message passing. An LSC chart is well-formed if the relation $\ll$ is acyclic (except trivial cyclic relation between locations connected by synchronous message passing). In the rest of the thesis, we assume that all charts are well-formed. We define function $preset$ to return the set of locations that precede a given location in the relation $\ll$.

$$preset : Location \rightarrow \mathbb{P}\, Location$$
$$\forall\, l : Location \bullet preset(l) = \{x : Location \mid x \ll l \wedge \neg(l \ll x)\}$$

One of the basic concepts used for defining the semantics of LSC is the notion of a cut. A cut through the chart represents the progress each instance has made in the scenario. Let cut be the function which returns the set of all possible cuts of a chart. A cut is a set of locations, one for each instance, satisfying the following condition:

$$cut : \mathcal{C} \rightarrow \mathbb{P}\, Location$$
$$\forall\, c : \mathcal{C} \bullet \forall\, x : cut(c) \bullet \#(x) = \#instances(c) \wedge \forall\, l : x \bullet \nexists\, l' : x \bullet l' \in preset(l)$$

Intuitively, it means no location in a cut is preceded with another. We are now ready to define the automaton which accepts exactly the language of a basic chart.

Definition 8 *The automaton associated with basic chart b is defined as $A_b \mathrel{\widehat{=}} (S_b, S_b^0, F_b, \Sigma_b \cup \{\tau\}, T_b)$. S_b is the state space. $S_b \mathrel{\widehat{=}} \{Aborted, Terminated, Completed\} \cup Active$ where $Active \mathrel{\widehat{=}} cut(b)$. $S_b^0 \mathrel{\widehat{=}} \bigcup_i \{l_i^0\}$ is the initial state. F_b is the set of final (accepting) states.*

$$F_b \mathrel{\widehat{=}} \{Aborted, Terminated, Completed\} \cup \{s : cut(b) \mid \forall\, l : s \bullet temp(l) = Cold\}$$

Σ_b is the set of events appearing in the chart. The special event τ denotes temporal progress along a vertical line. $T_b : S_b \times \Sigma_b \cup \{\tau\} \rightarrow S_b$ is the least transition relation satisfying the following:

$$
\begin{array}{ll}
D1 & \forall\, x : \Sigma_b \bullet (Terminated, x, Terminated) \in T_b \\
D2 & \forall\, x : \Sigma_b \bullet (Completed, x, Completed) \in T_b \\
D3 & (\bigcup_i \{l_i^{max}\}, \tau, Completed) \in T_b \\
D4 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \exists\, a : A \bullet \\
 & \quad (a \in label(l_i^k) \wedge eval(cond(l_i^k)) = true \wedge l_i^k \neq l_i^{max}) \\
 & \qquad \Rightarrow (s, a, (s \setminus \{l_i^k\}) \cup \{l_i^{k+1}\}) \in T_b \\
D5 & s \in cut(b) \wedge \exists\, l_i^k, l_j^p : s \bullet \exists\, m : M \bullet \\
 & \quad (m! \in lable(l_i^k) \wedge m? \in label(l_j^p) \wedge l_i^k \neq l_i^{max} \wedge l_j^p \neq l_j^{max} \\
 & \quad \wedge\, eval(cond(l_i^k)) = true \wedge eval(cond(l_j^p)) = true) \\
 & \qquad \Rightarrow (s, m, (s \setminus \{l_i^k, l_j^p\}) \cup \{l_i^{k+1}, l_j^{p+1}\}) \in T_b \\
D6 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \\
 & \quad eval(cond(l_i^k)) = false \wedge temp(cond(l_i^k)) = Cold \Rightarrow (s, \tau, Terminated) \in T_b \\
D7 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \\
 & \quad eval(cond(l_i^k)) = false \wedge temp(cond(l_i^k)) = Hot \Rightarrow (s, \tau, Aborted) \in T_b
\end{array}
$$

The chart is *completed* if all instances have reached the very last location. It is *terminated* if a cold condition is violated, and *aborted* if a hot condition is violated. Otherwise, we say that the chart is *active*, i.e., there exists a cut through every instance in the chart. Initially, the chart is active and all instances are at their first location. A state is accepting if and only if either it is completed or terminated or aborted, or it is an active state where all instances are at a cold location. *D1* and *D2* state that all behaviors are allowed when a chart is terminated or completed. *D3* states that a chart is terminated only after all instances have reached their last locations. *D4* and *D5* state that a local action or a message passing may occur only if the system can reach a new cut after engaging in the communication event or local action. Whenever a cold condition is evaluated to false, the chart terminates (*D6*). If the condition is labeled hot, the chart aborts so that no further behavior is allowed (*D7*). No compositional operator offered by LSC is discussed in this definition. For a chart with hierarchy, we can flatten the sub-charts by adding transitions connecting the initial and $Terminated$ state of the sub-chart to states in the automaton of the upper-level chart. For instance, a conditional branch can be flattened by connecting the last state of the upper-level chart to the initial states of both branches. As the flattening is a standard process, we omit the detail in this definition. Moreover, we adopt an interleaving semantics, e.g., no priority is associated with conditions, etc.

A trace of the automaton A_b is a sequence of events $\langle e_1, \cdots, e_k, \cdots e_n \rangle$, where there exists a run s_1, e_1, s_2, $\cdots$, s_k, e_k, s_{k+1}, $\cdots$, s_n, e_n, s_{n+1} such that $s_1 = S_b^0$ and $s_{n+1} \in F_b$ and for all $1 \leq k \leq n$ such that $(s_k, e_k, s_{k+1}) \in T_b$. The language of the automaton A_b, denoted as $\mathcal{L}(A_b)$, contains all traces of the automaton A_b.

Definition 9 *The language of a basic chart b, denoted as $\mathcal{L}_\beta(b)$, is the language of the automaton $\mathcal{L}(A_b)$. The executions of b which complete the whole chart, denoted as $\mathcal{F}_\beta(b)$, contain the traces of A_b which reach the state $Completed$ once and once only.*

The semantics of existential charts is different from that of basic charts because existential charts, as universal charts, are scoped. Events invisible to the chart may occur freely between any two successive events in an execution of the chart. Given a set of events $\Sigma_i \subseteq \Sigma$, a trace filter, denoted as $tr \upharpoonright \Sigma_i$, satisfies the following conditions:

$\langle\rangle \upharpoonright \Sigma_i \mathrel{\widehat{=}} \langle\rangle$
$(i \frown tr') \upharpoonright \Sigma_i \mathrel{\widehat{=}} i \frown (tr' \upharpoonright \Sigma_i)$, where $i \in \Sigma_i$
$(j \frown tr') \upharpoonright \Sigma_i \mathrel{\widehat{=}} tr' \upharpoonright \Sigma_i$, where $j \notin \Sigma_i$

In the following definition, forbidden events are properly handled, i.e., they are prevented from occurring until the chart completes.

Definition 10 *Let Σ_e be the set of events visible to an existential chart e. The language of e, denoted as $\mathcal{L}_\epsilon(e)$, is defined as: $\mathcal{L}_\epsilon(e) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid tr \upharpoonright \Sigma_e \in \mathcal{L}_\beta(e)\}$. The executions of e which travel through the whole chart, denoted as $\mathcal{F}_\epsilon(e)$, is defined as:*

$$\mathcal{F}_\epsilon(e) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid tr \in \mathcal{L}_\epsilon(e) \wedge tr \upharpoonright \Sigma_e \in \mathcal{F}_\beta(e)\}$$

A trace tr is a fragment of trace tr', denoted as $tr\ in\ tr'$, if and only if tr is a sub-sequence of tr'.

$$\begin{array}{|l} \_\ in\ \_ : \Sigma^* \leftrightarrow \Sigma^* \\ \hline \forall\, tr, tr' : \Sigma^* \bullet tr\ \underline{in}\ tr' \Leftrightarrow \exists\, tr_1, tr_2 : \Sigma^* \bullet tr_1 \frown tr \frown tr_2 = tr' \end{array}$$

A universal chart is typically preceded with a pre-chart. Whenever an execution completes the pre-chart, the execution must proceed as specified by the main chart.

Definition 11 *Let $p, m : \mathcal{B}$ be the pre-chart and main-chart of a universal chart u. The language of u is $\mathcal{L}_\mu(u)$ satisfying the following:*

$$\mathcal{L}_\mu(u) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid \nexists\, tr_1, tr_2 : \Sigma^* \bullet tr_1 \frown tr_2 \; in \; tr \wedge tr_1 \in \mathcal{F}_\epsilon(p) \wedge tr_2 \notin \mathcal{L}_\epsilon(m)\}$$

Intuitively, a trace violates a universal chart if and only if it completes the pre-chart but fails to conform to the main chart. By associating different sets of visible events to the pre-chart and main chart, various kinds of forbidden events [70] can be handled properly.

An LSC specification consists of a set of universal charts and existential charts, i.e., $\mathcal{S} \subset \{x : \mathcal{C} \mid x \in \mathcal{U} \vee x \in \mathcal{E}\}$. An implementation satisfies an LSC specification if and only if it always exhibits behaviors allowed by the universal charts and it is capable of exhibiting at least one of the behaviors captured by an existential chart.

Definition 12 *An implementation $\mathcal{I}$, whose executions are denoted as $traces(\mathcal{I})$, satisfies an LSC specification $\mathcal{S}$, denoted as $\mathcal{I} \models \mathcal{S}$, if and only if:*

$$(traces(\mathcal{I}) \subseteq \bigcap_{u \in \mathcal{S}} \mathcal{L}_\mu(u)) \wedge (\forall\, e \in \mathcal{S} \bullet \mathcal{F}_\epsilon(e) \cap traces(\mathcal{I}) \neq \varnothing)$$

4.1.3 Operational Semantics of LSC in CSP

This section is devoted to a CSP modeling of LSC. With the operational semantics of CSP defined in [135], the CSP modeling in a way defines an operational semantics for LSC. An intuitive way of constructing CSP models from LSC models is by mimicking the states in the automaton associated with a chart. However, mimicking the states is impractical because it requires constructing the (unstructured) automata. Moreover, it results in an unreadable CSP model and thus creates barriers to linking the verification results to the charts. We present our structure-preserving modeling using a set of operational semantics rules in a bottom-up fashion. The key idea is of using a (bounded) set of special synchronization events to monitor the completion of universal charts.

During a system run, a universal chart may be activated more than once and some of the activations may overlap. In general, there could be infinite overlapping activations of the same chart. Violation

of a cold condition terminates one activation only. Therefore, it is necessary to distinguish different activations (by associating each one with a unique identifier). In [17], 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. Thus, only a finite number of processes are necessary for monitoring overlapping activations, and they can be reused for non-overlapping activations. In practice, large number of overlapping activations of the same chart is unlikely because system behaviors are increasingly restricted as the number of overlapping activations increases. In the following, we present a set of rules for constructing CSP processes allowing no overlapping activations of the same chart. It can be readily extended to allow finite overlapping activations. We remark that the assumption of finite overlapping activations is reasonable comparing to strong assumptions like no overlapping activations made in [68].

The most primitive building blocks of LSC are locations. Along an instance in a chart, there are a finite number of locations. Due to our assumption of no co-region, a location contains at most one event and an optional condition. Let Σ_u^i be the set of events associated with instance i in chart u, including forbidden events. In addition, each chart is associated with three groups of special events, $\Sigma'(u, x) \mathrel{\widehat{=}} \{ter_u.x, hcv_u, syn_u.x\}$ where the optional x identifiers a sub-chart of u. Event $ter_u.x$ is engaged if and only if a cold condition is violated, either in the pre-chart or the main chart, or an unexpected event of the pre-chart is engaged. It is used to terminate all instances in a chart at once. Event hcv_u is engaged only when a hot condition is violated so that the system is forced to fail. This reflects the semantics of hot conditions. However, this is slightly problematic as the intention of hot conditions is to make sure they are never violated. A hot condition is violated either because there is inconsistency in the specification, e.g., wrong implementation of the local action, or the system is insufficiently specified. In our approach, the CSP model checker, e.g., FDR, helps to refine LSC specifications step by step so that all hot conditions hold all the time. Event $syn_u.x$ is used to synchronize the entering or exiting of a chart or the sub-chart x among all participating instances. Let $MainLoca(u, i, l)$ be the process for location l on instance i in the main chart of chart u. Let $MainLoca(u, i, l + 1)$ be the process of the next location. For the sake of readability, the following processes are defined accordingly to the respective states in Definition 8.

$Terminated_u \mathrel{\widehat{=}} \textsc{Run}_{\Sigma_u}$
$Completed_u \mathrel{\widehat{=}} \textsc{Run}_{\Sigma_u}$
$Aborted \mathrel{\widehat{=}} \textsc{Stop}$

Given that the condition labeled with location l is cold and the location is not the last, if the condition evaluates to true, the system engages in the event and proceeds to the next location, otherwise, it engages in a special event ter_u to signal all other instances in the chart. Processes for all other instances in the chart are interrupted by ter_u and terminate. After engaging in the special event, the process restores to the first location in the pre-chart so as to allow later activation of the chart.

R1 $MainLoca(u, i, l) \mathrel{\widehat{=}}$
$(label(l) \rightarrow MainLoca(u, i, l+1)) \langle\!\langle\ cond(l)\ \rangle\!\rangle (ter_u \rightarrow PreLoca(u, i, l_i^0))$

If the condition is cold and the location is the last, after engaging in the event, a special event syn_u is synchronized by all instances in the chart before any of them terminates. If the location is in a sub-chart of the main chart, then the event syn_u is replaced with $syn_u.x$ so that only participating instances synchronize the termination of the sub-chart.

R2 $MainLoca(u, i, l) \mathrel{\widehat{=}}$
$(label(l) \rightarrow syn_u \rightarrow PreLoca(u, i, l_i^0)) \langle\!\langle\ cond(l)\ \rangle\!\rangle (ter_u \rightarrow PreLoca(u, i, l_i^0))$

If the condition is hot and the location is not the last, a special event hcv_u is engaged if the hot condition is violated so that all other instances in the chart are signaled to deadlock.

R3 $MainLoca(u, i, l) \mathrel{\widehat{=}}$
$(label(l) \rightarrow MainLoca(u, i, l+1)) \langle\!\langle\ cond(l)\ \rangle\!\rangle (hcv_u \rightarrow Aborted_u)$

Lastly, if the condition is hot and the location is the last,

R4 $MainLoca(u, i, l) \mathrel{\widehat{=}}$
$(label(l) \rightarrow (syn_u \rightarrow PreLoca(u, i, l_i^0))) \langle\!\langle\ cond(l)\ \rangle\!\rangle (hcv_u \rightarrow Aborted_u)$

Similarly, we may construct the process for a location l in the pre-chart. Let $PreLoca(u, i, l)$ be the process constructed for location l on instance i in the pre-chart of chart u. If the instance is not in the pre-chart, then all visible events are allowed to occur before it synchronizes with the rest of the instance on entering of the main chart.

R5 $PreLoca(u, i, l_i^0) \mathrel{\widehat{=}}$
$(syn_u \rightarrow MainLoca(u, i, l_i^0))$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{ter_u, hcv_u, syn_u\} \rightarrow PreLoca(u, i, l_i^0))$

Given the location l is in the pre-chart and is not the last, if the condition evaluates to false, then the process signals all other instances in the chart and terminates. Otherwise, if the expected event is engaged, the process proceeds to the next location, else, the process engages in the unexpected event and terminates. Conditions are not distinguished as either hot or cold because hot conditions have no semantic meaning in pre-charts.

R6 $PreLoca(u, i, l) \mathrel{\widehat{=}}$
$((label(l) \rightarrow PreLoca(u, i, l+1)) \Box$
$(\Box\, e : \Sigma_u^i \setminus \{label(l), event(l_i^0), ter_u, hcv_u, syn_u\} \rightarrow ter_u \rightarrow \textsc{Skip}))$
$\langle\!|\, cond(l) \,|\!\rangle (ter_u \rightarrow \textsc{Skip})$

If the location is the last, after engaging in the event, the instance waits for the synchronization for termination and proceeds to the first location of the main chart.

R7 $PreLoca(u, i, l) \mathrel{\widehat{=}}$
$((label(l) \rightarrow syn_u \rightarrow MainLoca(u, i, l_i^0)) \Box$
$(\Box\, e : \Sigma_u^i \setminus \{label(l), event(0), ter_u, hcv_u, syn_u\} \rightarrow ter_u \rightarrow \textsc{Skip}))$
$\langle\!|\, cond(l) \,|\!\rangle (ter_u \rightarrow \textsc{Skip})$

A location could be extended to a structuring construct, e.g., a sub-chart, a branching, etc. All LSC structuring constructs have their exact images in CSP, e.g., process reference for sub-charts, choice in CSP for branching, etc. However, in case of sub-charts, violation of a cold condition terminates the sub-chart only and thus we need to attach some identifier of the sub-chart to the event $ter_u.x$ so that only the process for the sub-chart is terminated.

Let $Instance(u, i)$ be the process for instance i in chart u. The process terminates whenever a cold condition is violated in the chart and deadlocks whenever a hot condition is violated. Both are captured using interrupt operators.

R8 $Instance(u, i) \mathrel{\widehat{=}} (PreLoca(u, i, l_i^0) \triangledown_{ter_u} Instance(u, i)) \triangledown_{hcv_u} \rightarrow \textsc{Stop}$

Let $Chart(u)$ be the process for chart u. The process is an alphabetized parallel composition of the processes of all instances in the chart. Whenever a hot condition is violated, the process deadlocks and, therefore, the system deadlocks (refer to L2 in Chapter 2).

R9 $Chart(u) \mathrel{\widehat{=}} \|_i (Instance(u, i), \Sigma_u^i \cup \{ter_u, hcv_u, syn_u\})$

An LSC specification consists of a finite number of universal charts, each constraining its visible events. Let $\mathcal{I}$ be the process synthesized from the LSC specification. The process is the alphabetized parallel composition of the processes of all universal charts in the specification.

R10 $\mathcal{I} \mathrel{\widehat{=}} \|_{u \in \mathcal{S}} (Chart(u), \Sigma_u \cup \{ter_u, hcv_u, syn_u\})$

We claim that $\mathcal{I}$ is an implementation of $\mathcal{S}$. From the construction of $Chart(u)$, it is clear that only behaviors satisfying the chart are allowed, i.e.,whenever the pre-chart is matched, the system run must proceed as specified by the main chart (**R1,2,3,4**), whenever a cold condition is violated, the activation of the chart terminates (**R8**), whenever a hot condition is violated, the system deadlocks (**R8,9**), etc. Because all activation shares the same set of visible events, system execution is constrained by all activations. Therefore, $\mathcal{I}$ only allows behaviors that satisfies all the charts (because of the parallel composition). Lastly, $Chart(u)$ only constraints its visible events (as it is alphabetized) and other events are free to occur.

4.1.4 FDR Verification

In this section, we show how we solve the verification problem of LSC using an existing model-checker instead of building one from scratch. Machine readable CSP processes, i.e., an ASCII based variant of CSP [128], are constructed from LSC models and fed into FDR for checking.

Using FDR, safety, liveness and combination properties can be verified by showing a refinement relation from the constructed CSP model to the CSP process capturing the properties. Since this is the standard usage of FDR, we focus on checking that is closely coupled with our interpretation. Our interpretation ensures that inconsistency between universal charts results in deadlock. FDR is capable of telling whether a CSP program is deadlock-free. A counter example is presented whenever the validation fails, which gives an important clue to the origin of the error. There are two sources of deadlock, one due to inconsistencies between universal charts and the other due to violation of a hot condition. The former requires re-investigation of the system requirements. The latter may suggest either there is some inconsistency or the system is under-specified and thus more system requirements are necessary to constrain the state variables sufficiently. An existential

chart is validated by constructing the corresponding CSP process and checking whether it contains a trace allowed by the CSP model constructed from the universal charts. In addition, manual proof may establish properties of the LSC model expressible with logical expressions over traces or trace-refusal pairs.

Example 4.1.3 (Inconsistent universal charts) Figure 4.2 is presented in [68] as a typical example of inconsistency between universal charts[1]. It is a part of the LSC specification of an automatic railway system [32]. The objects participating in the scenarios are $cruiser$, car and $carHandler$. In the left chart, the message $setDest$ sent from the environment to the car activates the chart, which requires that following the $departReq$ message, $departAck$ is sent from the car handler to the car. This message in turn activates the right chart, which requires the sending of $engage$ from the car to the cruiser before the $start$ and $started$ messages are sent, while the left chart requires the opposite ordering. **end**

The program in Figure 4.3 is constructed automatically by our supporting tool (manually simplified so as to improve readability). The first part of the program consists of channel definitions for all communication events in the charts. The set of events visible to a chart are exactly those that appear in the chart together with the forbidden events. The construction follows exactly the operation semantics rules. Because there is only one message in the pre-chart and we assume that the internal computation is infinitely faster than the arrival of external stimulus, there is no overlapping activations for this example.

FDR instantly reports that process $System$ is not deadlock-free. A trace leading to deadlock is illustrated as a counter example: $\langle departAck, setDest, departReq\rangle$. The right chart is activated by event $departAck$. Right after that the left chart is activated by event $setDest$. This is possible because $setDest$ is not constrained by the right chart. After event $departReq$, the system deadlocks. This deadlock situation is not what we expected. However, it does reveal an implicit assumption that is not captured by the charts, i.e., $departAck$ occurs only after $setDest$ and $departReq$

[1]More complicated examples are available at [148]

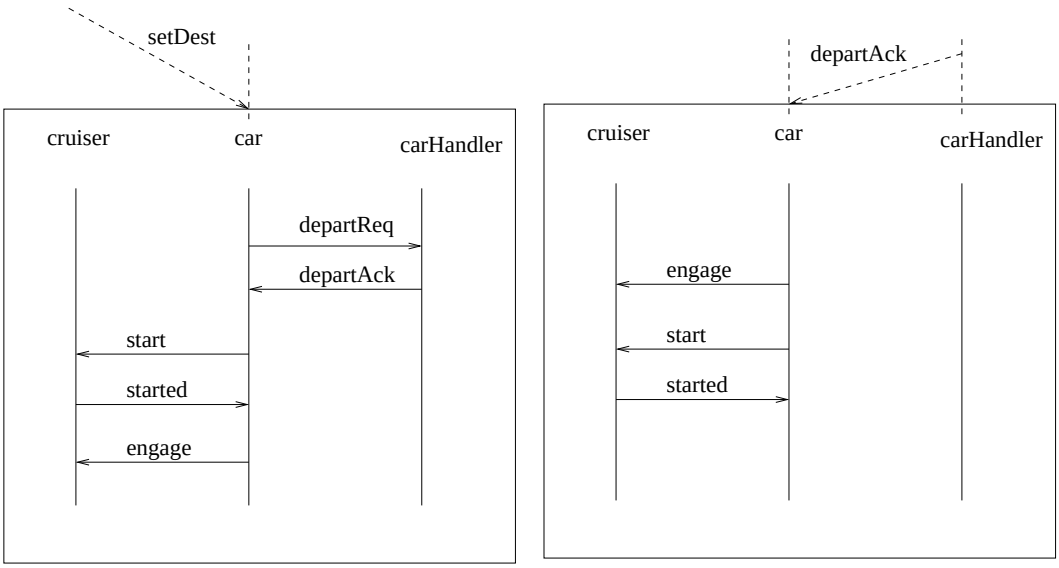


Figure 4.2: Inconsistent universal charts

are engaged. This assumption can be embedded using a universal chart or the following process: $Env = setDest \rightarrow departAck \rightarrow Env$. We refine the process $System$ as the parallel composition of the original $System$ and Env. FDR reports instantly the expected deadlock trace: $\langle setDest, departReq, departAck \rangle$. This example reveals a complication due to implicit assumptions we often make on the system, which is an important issue in solving the synthesis problem.

In order to cope with large systems, CSP algebraic laws are used to simplify the constructed processes before feeding them into FDR. Compression methods available in FDR can be applied as well, as in Figure 4.3 (the first line). For instance, the option *diamond* requires FDR to compress the system using *diamond elimination*, i.e., a node-compression used to reduce the search space based on partial order reduction. Our construction is extended to handle symbolic instances and messages, i.e., symbolic instances are modeled as processes with parameters and local definitions, symbolic messages are modeled as typed channel events.

A CSP process is constructed from an existential chart similarly except that the process for the existential chart deadlocks after the chart completes. This allows validation of existential charts

```
transparent diamond, normalise

channel sync_setDest, sync_departAck, setDest, departReq
       departAck, start, started, engage

SetDestSigma(0) = {|sync_setDest,setDest,departReq,departAck,start,
                    started,engage|}
SetDestSigma(1) = {|sync_setDest,start,started,engage|}
SetDestSigma(2) = {|sync_setDest,departReq,departAck|}

SetDestInst(0) = setDest -> sync_setDest -> departReq -> departAck ->
           start -> started -> engage -> SetDestInst(0)
       [] departReq -> SetDestInst(0) [] departAck -> SetDestInst(0)
       [] start -> SetDestInst(0) [] started -> SetDestInst(0)
       [] engage -> SetDestInst(0)
SetDestInst(1) = sync_setDest -> start -> started -> engage -> SetDestInst(1)
       [] start -> SetDestInst(1) [] started -> SetDestInst(1)
       [] engage -> SetDestInst(1)
SetDestInst(2) = sync_setDest -> departReq -> departAck -> SetDestInst(2)
       [] departReq -> SetDestInst(2) [] departAck -> SetDestInst(2)
SetDest = || x: {0..2}@ [SetDestSigma(x)] SetDestInst(x)

DepartSigma(0) = {|sync_departAck,departAck,engage,start,started|}
DepartSigma(1) = {|sync_departAck,engage,start,started|}
DepartSigma(2) = {|sync_departAck,departAck|}

DepartInst(0) = departAck -> sync_departAck -> engage -> start
           -> started -> DepartInst(0)
       [] engage -> DepartInst(0) [] start -> DepartInst(0)
       [] started -> DepartInst(0)
DepartInst(1) = sync_departAck -> engage -> start -> started -> DepartInst(1)
       [] engage -> DepartInst(1) [] start -> DepartInst(1)
       [] started -> DepartInst(1)
DepartInst(2) = departAck -> sync_departAck -> DepartInst(2)
Depart = || x: {0..2}@ [DepartSigma(x)] DepartInst(x)

Sigma(0) = {|sync_setDest,setDest,departReq,departAck,start,started,engage|}
Sigma(1) = {|sync_departAck,departAck,engage,start,started|}
Sigma(2) = {|setDest,departAck|}

Figure(0) = SetDest Figure(1) = Depart
System = || x: {0..1} @ [Sigma(x)] Figure(x)

{- ************** Assertions ************** -}
assert System :[deadlock free [FD] ]
```

Figure 4.3: Machine readable CSP example

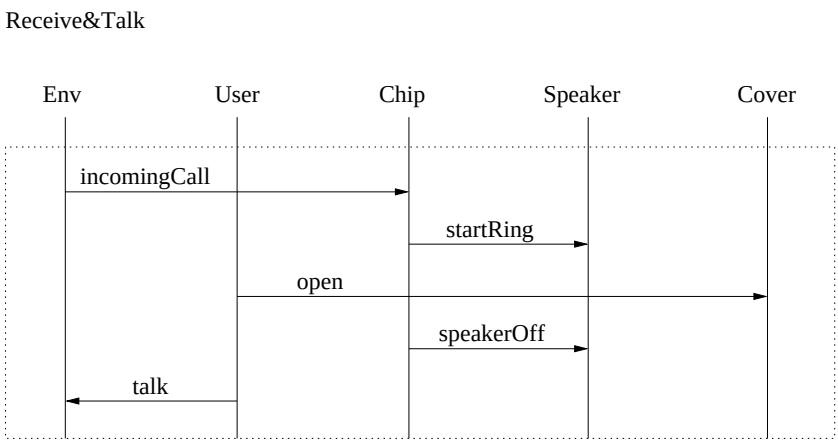


Figure 4.4: Existential chart

using the notion of refinement. An existential chart is consistent with an LSC model if and only if it specifies at least one trace which is consistent with the universal charts.

Example 4.1.4 (Existential chart) The dotted frame indicates that the chart in Figure 4.4 is an existential chart. This chart captures the most common scenario of the mobile phone system: once there is an incoming call, the speaker shall start to ring and the user shall open the mobile to talk. The CSP process constructed for the existential chart in Example 4.1.4 is as the following:

$$
\begin{aligned}
&Instance(0) = incomingCall \rightarrow talk \rightarrow Stop \\
&Instance(1) = open \rightarrow talk \rightarrow Stop \\
&Instance(2) = incomingCall \rightarrow startRing \rightarrow speakOff \rightarrow Stop \\
&Instance(3) = startRing \rightarrow speakerOff \rightarrow Stop \\
&Instance(4) = open \rightarrow Stop \\
&Existential = \|_{x=0}^{4}(Instance(x), \Sigma_x)
\end{aligned}
$$

Σ_x contains exactly the events of the instance appeared in the chart. Given $System$ as the process constructed from the universal charts of the phone system, FDR verifies that $System$ is trace-refined by process $Existential$. Thus, we are certain that the existential chart is consistent with the universal charts. In general, the notion of trace-refinement is too strong for validation of existential charts. If the chart contains branches, we need to show the CSP constructed from the universal charts is capable of exhibiting at least one system run allowed by the existential chart. **end**

FDR can also be used to verify whether a property holds by showing a refinement relationship from the candidate process to a CSP process capturing the property. We can verify safety requirements by trace refinement and liveness requirements by failure/divergence refinement. Safety and liveness properties may be expressed as universal charts or CSP processes intuitively. For example, we may express a safety property as a universal chart (without pre-chart) containing only a hot condition capturing the property. Formal derivation of CSP processes or LSC from temporal specifications and vice versa are non-trivial research topics. The former was discussed in [10, 106] and the latter in [89].

The construction is automated using XML and JAVA technology. There is not yet a standard interchange format for LSC. The XML format used in *Play-Engine* is not intended to exchange LSC models. No schema or DTD definition is developed. A part of our work includes the development of the first XML standard interchange format for LSC. We start with defining the syntax of LSC using both BNF grammar and XML schema. The BNF grammar is presented in Appendix B. The XML schema and XML representation of the charts appeared in this chapter can be found online [148]. Together with the XML schema, a parser and a transformation module is built using JAVA and an existing XML parser [59] to parse XML representation of LSC models and construct CSP programs automatically. The output of the program is a machine readable CSP program with a set of assertions, which is ready to be employed and verified in FDR.

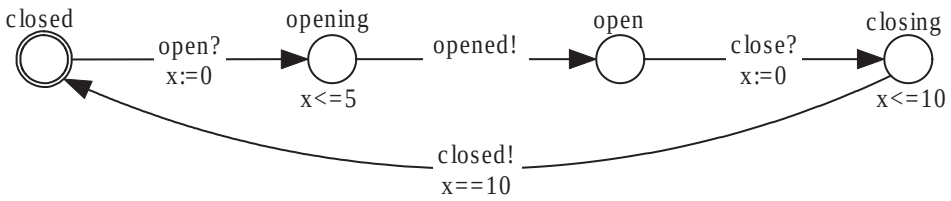
4.2 Verification of Timed CSP and TCOZ

This section is devoted to a brief discussion on applying verification mechanisms for visual formalisms to validate logic-based formalisms using concrete examples. Timed CSP (and of course TCOZ) aims at specifying complex real-time systems. However, there is no mechanical reasoning support for Timed CSP models. Indeed, there has been little automation on reasoning logic-based specification languages for real-time systems. One of the reasons is that logic-based formalisms like Timed CSP and TCOZ are so expressive that they are beyond any efficient verification techniques. For instance, Timed CSP allows specification of languages that are not regular or not even context-

free, which makes model checking infeasible. In this section, we present our attempt at verifying Timed CSP and TCOZ specifications by reusing existing verification mechanisms.

In the field of visual formalism, Timed Automata [2] are often used to model real-time systems. Timed Automata are finite state machines equipped with clocks. Its definition provides a general way to annotate state transition graphs with timing constraints using finitely many real-valued clock variables. In a timed automaton, each node is associated with an invariant, while a transition is labeled with a guard (a constraint on clocks), a synchronization action and a clock reset set (a set of clocks to be reset). Intuitively, a timed automaton starts execution with all clocks initialized to zero. The automaton can stay at a node, as long as the invariant of the node is satisfied, with all clocks increasing at the same rate. A transition can be taken if the values of the clocks fulfill the guard. By taking the transition, all clocks in the clock reset set are set to zero, while the clocks not in the clock reset set keep their values.

Example 4.2.1 (Timed Automaton) The following is a Timed Automaton modeling the behaviors of a $door$:

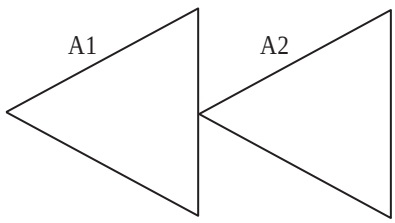


A double-line circle indicates an initial state in the notion of Timed Automata. Initially, the system is at state $closed$. After getting a request through channel $open$, the system moves to the $opening$ state whilst resetting the local clock x. The $opening$ state is labeled with a state invariant $x <= 5$ so that the door takes at most 5 time units to open. After the door is opened, a message is sent over the channel $opened$. At $open$ state, a request on channel $close$ moves the system to state $closing$, which is labeled with a state invariant too. The transition out of state $closing$ is guarded with the condition $x == 10$, and therefore the door takes exactly 10 time units to close. **end**

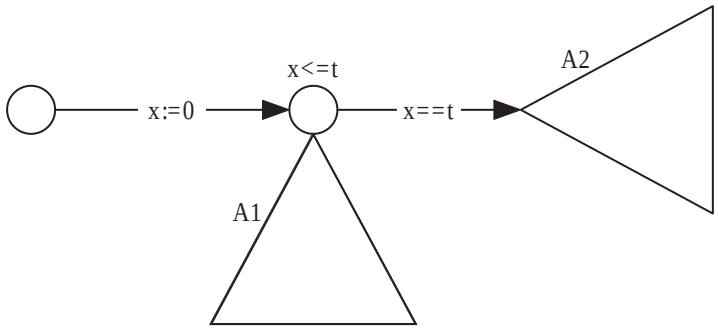
Model checking of Timed Automata has been proven to be decidable [2]. There have been quite a number of model checkers for Timed Automata [35, 30, 9]. UPPAAL [9] is one of the most successful tools. It is a tool for modeling, simulation and verification of real-time systems modeled as a network of timed automata. The properties are expressed as a restricted subset of Timed CTL [76]. UPPAAL is our choice of model-checker for verifying a network of timed automata because of its efficiency (both for model-checking and simulation) as well as its wide recognition.

Real-time system requirements are often stated using high-level timing constraints like *timeout*, *timed interrupt*. Those are regarded as common timing constraint patterns. For example, "a task must complete within t time period" is a typical one ($deadline$). One problem of designing real-time system using Timed Automata is the lack of high level composable graphical patterns. System engineers thus often need to manually cast those timing patterns into a set of clock variables with carefully calculated clock constraints. This process is time consuming and error prone. On the other hand, Timed CSP (and TCOZ) is a good candidate for specifying complex real-time systems because it offers a rich set of constructs that can directly capture those common timing patterns. One interesting question is thus: can we build a set of Timed Automata patterns that correspond to Timed CSP timing constructs? If such Timed Automata patterns can be formulated formally, not only we can systematically translate Timed CSP to Timed Automata for validation, but also Timed Automata can be used for compositional design. We thus investigated possible relationships between Timed CSP and Timed Automata. A set of composable graphical patterns is defined based on the Timed CSP hierarchical constructs.

Example 4.2.2 (Timed Automata patterns) The following figure demonstrates sequential composition of two Timed Automata A_1, A_2:



An automaton is abstracted as a triangle, the left vertex of this triangle or a circle attached to the left vertex represents the initial state, and the vertical edge represents the terminal states. By linking the terminal states of A_1 with the initial state of A_2, the control is passed from A_1 to A_2 when A_1 goes to its terminal state. The following is the Timed Automata pattern for time out:



The initial state of automaton A_1 is labeled with state invariant $x \leq t$, which guarantees the system must go beyond the state after t time units. If a transition is taken before t time units, the control remains in automaton A_1. Otherwise, after exactly t time units, automaton A_1 times out and the control is passed to automaton A_2. **end**

A full list of Timed Automata patterns together with their formal definitions in Z is presented in [42]. These timed composable patterns provide a reusable high level library to facilitate a systematic engineering process using Timed Automata as a design language. Furthermore, these patterns offer an interchange media for transforming Timed CSP (and TCOZ) specifications into a network of Timed Automata, which allows reusing UPPAAL to verify Timed CSP (and TCOZ) specifications. The projection from TCOZ to Timed Automata is automated using the same method presented in Section 3.4.4, i.e., the ZML parser is reused and a Timed Automata interface and a new transformation module are built for the task.

However, because UPPAAL aims at efficient verification based on the notion of model checking, it puts strong restrictions on both the system models and properties to be verified. For instance, guard conditions in Timed Automata can not compare the valuation of clocks. Moreover, the properties supported by Timed Automata are limited to a small subset of Timed CTL [9]. In order to overcome

those limitations, we investigate a complemental approach. The key idea is of using Constraint Logic Programming (CLP [82]) as an underlying reasoner for real-timed systems modeled with Timed CSP or TCOZ. We omit the details of the work (refer to [44]) because it is only loosely connected to the scheme of the thesis.

4.3 Summary

In this chapter, we investigated the validation of visual and logic-based formalisms by reusing existing verification mechanisms. In order to demonstrate that model checkers for logic-based formalisms can be reused for verification of diagrammatic notations, we developed a CSP modeling of LSC and then applied FDR to reveal inconsistency in LSC models. In the other direction, we verified logic-based formalisms Timed CSP and TCOZ using tool support for diagrammatic notations Timed Automata.

As for related works, there have been attempts on formalizing LSC [85, 14]. In [14], Bontemps and Heymans used Büchi automata to define the language expressed by a set of LSCs. They claim that the standard algorithm for automata can be used to check consistency and refinement. However, because automata are typically low-level and not structured, flattening high-level LSC into automata suffers from the state explosion problem. CSP provides a rich set of compositional constructs. Our work preserves the structure of the LSC model and avoids constructing the global state machine both at the chart level or globally. Klose and Wittke [85] derive a similar timed Büchi automaton to capture the semantics of an LSC chart in isolation. Our approach handles multiple charts and is extensible. In [89], Kugler *el at* provided a semantics for a kernel subset of LSC using CTL*, which may be used in the development of tools for analyzing and executing LSC. However, no explicit verification support has been discussed. Our CSP modeling of LSCs allows not only mechanized verification of LSCs but also using CSP algebraic laws to solve the synthesis problem of LSCs (refer to Chapter 5).

Our work is also loosely related to works on formalizing, simulating, and validating MSCs/LSCs, e.g., the simulation tool developed by Wang *el at* based on Constraint Logic Programming [156]

and theoretical works on MSCs by Thiagarajan [91] and Mauw *el at* [109]. For CSP, the *de facto* mechanized verification support is FDR. There is not yet a mechanized proving method for Timed CSP. The main reason is the complexity of time, e.g., the timed trace and failure semantics of Timed CSP is far more complex than those of CSP. As far as the authors know, the only attempt is Brooke's work on partial encoding Timed CSP in PVS [20], which relies on heavy user interaction.

Chapter 5

Synthesis from Scenario-based Specification

'how am I to get in?' asked Alice again, in a louder tone.
'Are you to get in at all?'
said the Footman.
'That's the first question, you know.'
- *Alice's Adventures in Wonderland*, Lewis Carroll

Synthesis (from the Greek words syn = plus and thesis = position) is commonly understood to be an integration of two or more pre-existing elements which results in a new creation. In the software engineering literature, the term *synthesis* has been broadly used, to denote different kinds of problems. In the field of formal languages and temporal logics, the *synthesis* problem is well defined, since the late 80s. For open systems, it is interpreted as building an implementation that will preserve a specification against any malevolent environment [50], [51], [30], [52], [53], [22], [54], [55]. The problem of synthesis for closed systems is synonymous with satisfiability [56], [57]. In the realm of scenarios, the meaning of *synthesis* is somewhat vaguer, as the problem is always left undefined and only algorithms are discussed [15]. The problem we address in this chapter

is of generating implementations that will preserve an LSC specification (against any malevolent environment if it interacts constantly with the unspecified environment).

5.1 Introduction

A major challenge of software engineering is to automatically generate low-level executable implementations from high-level specifications. One high-level specification of special interest is scenario-based diagrams, which serve as an abstract and natural way of capturing inter-object system requirements. 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 UML [154], MSC in Specification and Description Language (SDL) [81], etc. In this work, we propose an approach to generate executable programs automatically from sequence diagrams, in particular, Live Sequence Charts.

Before generating implementations from sequence diagrams, there are two problems to be solved. The problem of verification is of exposing inconsistency between the diagrams. The problem of synthesis is of deciding whether there exists a satisfying object system and if so, synthesize one automatically. The former has been addressed in Chapter 4. The latter is crucial in the development of complex systems, as sequence diagrams serve as the manifestation of use cases and if synthesizable they could lead directly to implementation. In the setting of classic MSC, the problem of synthesis has been tackled by many researchers [5, 3, 87, 86]. The conclusion is that for reactive distributed systems, synthesizing a distributed object system with precisely the set of behaviors could be impossible because of its computational complexity as well as the notion of implied scenarios. Intuitively, implied scenarios are additional behaviors that may be present in every distributed object system which is consistent with the specified scenarios, i.e., the set of MSCs.

Example 5.1.1 (Implied scenarios) The charts presented in Figure 5.1 show a typical example of an implied scenario (inspired by the example in [3]). The first two MSCs are the specification, where the two events Req, Ack may occur in either order. The third is an implied scenario, where

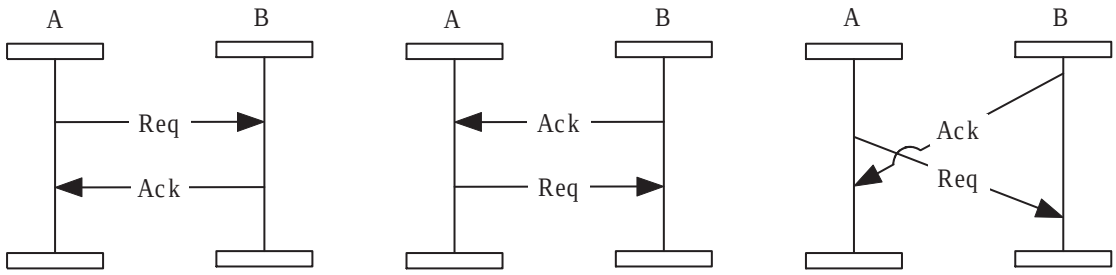


Figure 5.1: Implied scenarios

the reception of both events is delayed. In the third scenario, as far as the object A or B can tell, both of them are executing a specified scenario, i.e., A executes accordingly as the first and B executes accordingly as the second. **end**

In order to avoid the problem of implied scenarios, our synthesis is based on the notion of LSC. LSC is rapidly recognized as a rather rich and useful extension of MSC. It offers a far more powerful means for stating requirements for complex systems than MSC. It thus serves as an excellent basis of mechanized analysis of scenarios, for example, the study of the synthesis problem. In LSC, mandatory behaviors are specified using universal charts, which are distinguished from possible ones (as contrasted with MSC). In this work, we assume that an LSC specification contains a set of universal charts, whereas existential charts are only used for specifying test cases. We thus avoid the problem of implied scenarios (refer to the formal explanation in Section 5.6).

Despite the absence of implied scenarios, synthesis of a distributed object system from a set of scenarios remains a hard problem. In general, the distributed synthesis problem is undecidable in almost all interesting settings [122]. In order to deal with the great complexity, we developed a synthesis method relying on using a finite set of special events to monitor global execution locally. Nevertheless, our method automatically synthesizes distributed implementations efficiently and soundly. The key idea is to develop a CSP model of LSC and then use CSP's algebraic laws to transform the CSP model so that the local behaviors of each object are identified. The consequence is that we may construct one process for every object in the system capturing exactly its roles in

the system without constructing the global state machine. Lastly, distributed implementations are synthesized based on the distributed processes straightforwardly.

5.2 CSP with Liveness

In the last chapter, modality on locations is ignored because there is not much to verify about it. It is however important that we capture the liveness constraint in the work on synthesis. CSP lacks the expressiveness to capture liveness, i.e., a process may wait infinitely long before engaging in one of the enabled events. On the other hand, modality on locations in LSC constrains the execution of the system by requiring that no instance is stuck at a hot location forever, i.e., events labeled with a hot location must eventually be engaged. In order to capture the semantics of LSC using CSP, it is necessary to amend the traditional trace semantics of CSP to capture liveness. We solve the problem by distinguishing $signals$ from ordinary CSP events. $Signals$ are events that must be observed in the future state. The name, $signal$, is suggested by Davies. In his work [33], signals are used to express broadcast communication effectively in Timed CSP.

Let $\widehat{\Sigma} \subset \Sigma$ be the set of all signals. For each ordinary event e, a signal $\widehat{e}$ is registered. We remark that signals play the same role as ordinary events, e.g., synchronizing with signals or events obeying the CSP rules, except that they must be engaged eventually. In order to reflect the additional constraint caused by signals, we define a filter function to eliminate behaviors from the CSP trace model. The filter function $\mathcal{F}$ is defined as the following:

$$
\begin{array}{|l}
\mathcal{F} : \mathcal{P} \rightarrow \mathbb{P}\,\Sigma^* \\
\hline
\forall\, P_1 : P \bullet \mathcal{F}(P_1) = \{tr : \Sigma^* \mid tr \in traces(P_1) \wedge \\
\qquad \nexists\, tr' : \Sigma^*;\ \widehat{e} : \widehat{\Sigma} \bullet tr' = tr \frown \langle \widehat{e} \rangle \wedge tr' \in traces(P_1)\}
\end{array}
$$

Intuitively, a trace satisfies the liveness constraint if and only if all enabled signals have been engaged. Despite the filter function, the mature semantics models of CSP are maintained. The notion of signal captures (localized) liveness conditions in the same way as hot locations do. In the following, modality on location is handled universally in our refined modeling of LSC using CSP. That is,

events labeled with hot locations are modeled as $signals$, whereas events labeled with cold locations are modeled as ordinary CSP events.

Example 5.2.1 (Signals) Let $\widehat{\text{SKIP}}$ be the short form for $\widehat{\checkmark} \rightarrow \text{STOP}$. The following shows how to compute those traces that satisfy the liveness condition:

$$\mathcal{F}(\widehat{\text{SKIP}}) = \{(\langle\widehat{\checkmark}\rangle, X) \mid X \subseteq \mathcal{F}(\text{STOP})\} = \{\langle\checkmark\rangle\}$$
$$\mathcal{F}(\widehat{e} \rightarrow P) = \{(\langle\widehat{e}\rangle \frown s, X) \mid (s, X) \in \mathcal{F}(P)\}$$

The trace of $\widehat{\text{SKIP}}$ does not include the empty one because the event $\widehat{\checkmark}$ shall be engaged eventually. Similarly, a signal-prefixing shall not idle infinitely. **end**

5.3 Refined CSP Modeling of LSC

Our modeling of LSC using CSP in Section 4.1 is based on the assumption that there is no overlapping activations of the same chart. Using a finite pool of such processes, we may allow finite overlapping activations. It is reasonable since our objective in the last chapter is efficient verification of LSC models using FDR. In this section, we refine our modeling so that infinite overlapping activations are not restricted as one of the principles of synthesis is that the synthesized design shall be minimally restrictive so that further refinement is possible.

During a system run, a universal chart may be activated more than once and some of the activations may overlap. Therefore, it is necessary to distinguish different activations by associating each one with a unique identifier. Let $y : 1 \mathinner{..} n$ be the index of the y-th activation of a chart u. Each chart is associated with four groups of special events, $\Sigma'(u, x, y) \mathrel{\widehat{=}} \{ter_u.x.y, hcv_u, syn_u.x.y, fork_u.y\}$, where x is an optional identifier of the sub-chart. The special event $ter_u.x$ and $syn_u.x$ used in Section 4.1.3 are attached with y so that they are synchronized only among participating instances of the y-th activation of the chart. Event hcv_u is engaged when a hot condition is violated. It is irrelevant if the hot condition is violated in a sub-chart or a particular activation. Event $fork_u.y$ is used to fork a new activation of chart u. Let $\Sigma'(u)$ be the set of special events associated with chart u, i.e., $\Sigma'(u) \mathrel{\widehat{=}} \bigcup_x \bigcup_y \Sigma'(u, x, y)$. Let Σ' be the set of all special events, i.e., $\Sigma' \mathrel{\widehat{=}} \bigcup_u \Sigma'(u)$.

The process for location l on instance i in the main chart of the y-th activation of chart u is denoted as $MainLoca(u, i, l, y)$. Let $MainLoca(u, i, l+1, y)$ be the process of the next.

R1' $MainLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow MainLoca(u, i, l+1, y)) \langle\!\langle\ cond(l)\ \rangle\!\rangle (ter_u.y \rightarrow Terminated_u)$
where $temp(cond(l)) = Cold$ and $l \neq l_i^{max}$

If the condition labeled with l evaluates to true, the system engages in the event and proceeds to the next location, otherwise, event $ter_u.y$ is engaged to signal all other instances in the chart to termination. Processes for all other instances in the activation of the chart are interrupted by $ter_u.y$ to terminate so that the activation of the chart terminates.

R2' $MainLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow syn_u.y \rightarrow Completed_u) \langle\!\langle\ cond(l)\ \rangle\!\rangle (ter_u.y \rightarrow Terminated_u)$
where $temp(cond(l)) = Cold$ and $l = l_i^{max}$

After engaging in the event, event $syn_u.y$ is synchronized by all instances in the (activation of the) chart before any of them completes.

R3' $MainLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow MainLoca(u, i, l+1, y)) \langle\!\langle\ cond(l)\ \rangle\!\rangle (hcv_u \rightarrow Aborted)$
where $temp(cond(l)) = Hot$ and $l \neq l_i^{max}$

Event hcv_u is engaged if the hot condition is violated so that all other instances in the (activation of the) chart are signaled to deadlock (refer to **R10'**). Lastly,

R4' $MainLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow syn_u.y \rightarrow Completed_u) \langle\!\langle\ cond(l)\ \rangle\!\rangle (hcv_u \rightarrow Aborted)$
where $temp(cond(l)) = Hot$ and $l = l_i^{max}$

Let $PreLoca(u, i, l, y)$ be the process constructed for location l on instance i in the y-th activation of the pre-chart of chart u.

R5' $PreLoca(u, i, l_i^0, y) \mathrel{\widehat{=}}$
$(\Box\ e : \Sigma_u^i \rightarrow PreLoca(u, i, l_i^0, y))$
$\Box\ (fork_u?y \rightarrow Forked\ |[\ \Sigma_u^i\ ]|\ PreLoca(u, i, l_i^0, y+1))$
where instance i is not in the pre-chart of u and
$Forked \mathrel{\widehat{=}} \mu X \bullet ((syn_u.y \rightarrow MainLoca(u, i, l_i^0, y)) \Box$
$(\Box\ e : \Sigma_u^i \rightarrow X)) \triangledown_{ter_u.y}\ Terminated_u$

Before synchronizing the entering of the main chart and then behaving as specified by the main chart, the instance may engage in any event in Σ_u^i or synchronize on event $fork_u.y$ to fork a new copy of the process (in case this chart is activated by engaging in events associated with other instances). The activation terminates whenever a $ter_u.y$ event is engaged (due to either violation of a cold condition or engaging in an unexpected event in the pre-chart).

R6' $PreLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow PreLoca(u, i, l+1, y)$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l)\} \rightarrow ter_u.y \rightarrow Terminated_u))$
$\langle\!|\ cond(l)\ |\!\rangle$
$ter_u.y \rightarrow Terminated_u$
where location l is neither the first location nor the last.

If the condition evaluates to false, the process signals all other instances in the chart and terminates. Otherwise, if the expected event is engaged, the process proceeds to the next location, else, the process engages in an unexpected event and puts no further constraint on the system (L1 in Chapter 2).

R7' $PreLoca(u, i, l, y) \mathrel{\widehat{=}}$
$(event(l) \rightarrow syn_u.y \rightarrow MainLoca(u, i, l_i^0, y)$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l)\} \rightarrow ter_u.y \rightarrow Terminated_u))$
$\langle\!|\ cond(l)\ |\!\rangle$
$ter_u.y \rightarrow Terminated_u$
where the location is not the first location but is the last.

After engaging in the event, the instance waits for the synchronization and then proceeds to the first location of the main chart.

R8' $PreLoca(u, i, l_i^0, y) \mathrel{\widehat{=}}$
$(event(l_i^0) \rightarrow fork_u!y \rightarrow ((PreLoca(u, i, l_i^1, y) \triangledown_{ter_u.y} Terminated_u)$
$|[\,\Sigma_u^i\,]|\ PreLoca(u, i, l_i^0, y+1))$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l_i^0)\} \rightarrow PreLoca(u, i, l_i^0, y)))$
$\Box\ (fork_u?y \rightarrow ((Forked \triangledown_{ter_u.y} Terminated_u)$
$|[\,\Sigma_u^i\,]|\ PreLoca(u, i, l_i^0, y+1)))$
$\langle\!|\ cond(l_i^0)\ |\!\rangle$
$PreLoca(u, i, l_i^0, y)$
where the location is the first but not the last and
$Forked \mathrel{\widehat{=}} (event(l_i^0) \rightarrow PreLoca(u, i, l_i^1, y))$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l_i^0)\} \rightarrow Forked)$

A new process is forked whenever an expected event is engaged. This way, system runs that trigger overlapping activations of the same chart are properly constrained. The special events are not synchronized between different activations. Lastly,

R9' $PreLoca(u, i, l_i^0, y) \mathrel{\widehat{=}}$
$(event(l_i^0) \rightarrow fork_u!y \rightarrow syn_u.y \rightarrow$
$(MainLoca(u, i, l_i^0, y) \triangledown_{ter_u.y} Terminated_u)$
$|[\Sigma_u^i]|\, PreLoca(u, i, l_i^0, y+1)$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l_i^0)\} \rightarrow PreLoca(u, i, l_i^0, y)))$
$\Box\ (fork_u?y \rightarrow ((Forked \triangledown_{ter_u.y} Terminated_u)$
$|[\Sigma_u^i]|\, PreLoca(u, i, l_i^0, y+1)))$
$\langle\!|\ cond(l_i^0)\ |\!\rangle$
$PreLoca(u, i, l_i^0, y)$
where the location is the first and the last and
$Forked \mathrel{\widehat{=}} (event(l_i^0) \rightarrow syn_u.y \rightarrow MainLoca(u, i, l_i^0, y)$
$\Box\ (\Box\ e : \Sigma_u^i \setminus \{event(l_i^0)\} \rightarrow Forked))$

Whenever a chart is activated, the subsequent behavior of the system is constrained by both the process (for this activation) and the newly forked process (for any future activation) and, therefore, remains valid (**R5',R8',R9'**). The process $PreLoca(u, i, 0, y)$ allows, in general, infinite overlapping activations of the same chart. Let $Instance(u, i)$ be the process for instance i in chart u.

R10' $Instance(u, i) \mathrel{\widehat{=}} PreLoca(u, i, 0, 0) \triangledown_{hcv_u} Aborted$

The process deadlocks whenever a hot condition is violated. Each chart consists of a finite number of instances. Let $Chart(u)$ be the process for chart u.

R11' $Chart(u) \mathrel{\widehat{=}} \|_i (Instance(u, i), \Sigma_u^i \cup \Sigma'(u))$

The process is an alphabetized parallel composition of the processes of all instances in the chart. Whenever a hot condition is violated, the process deadlocks and, therefore, the system deadlocks. An LSC specification consists of a finite number of universal charts, each constraining its visible events. Let $\mathcal{I}$ be the process synthesized from the LSC specification. The process is the alphabetized parallel composition of the processes of all universal charts in the specification.

R12' $\mathcal{I} \mathrel{\widehat{=}} ((\|_{u \in \mathcal{S}} (Chart(u), \Sigma_u \cup \Sigma'(u)))\ \setminus \Sigma') \parallel \textsc{Run}$

The RUN portion is added to make sure events which are not visible to any of the universal charts (but may appear in some existential chart) can occur freely. Process $\mathcal{I}$ is an implementation of $\mathcal{S}$. Formally,

Theorem 5.3.1 $\mathcal{F}(\mathcal{I}) \subseteq \bigcap_{u \in \mathcal{S}} \mathcal{L}_\mu(u)$

Skeleton of proof: *From the construction of $Chart(u)$, it is clear that only behaviors satisfying the chart are allowed. Whenever a chart is activated, a new copy is forked to monitor subsequent possible activation of the chart(**R5',R8',R9')** and because all activation shares the same set of visible events, system execution is constrained by all activations. Because invisible events are not constrained, they are free to occur between any consecutive occurrence of visible events. Because events labeled with hot locations are mapped to signals, system runs stuck at a hot location are filtered by the function $\mathcal{F}$. We skip the full proof as it is extremely lengthy.* □

Example 5.3.2 (Process synthesis) Part of the chart *Talk* (presented in Example 4.1.2) (instance $env, user, cover$) is interpreted as the CSP processes presented in Figure 5.2. The full construction is available online [148]. **end**

5.4 Synthesis

This section is devoted to our solution for the synthesis problem. We first handle closed systems and then discuss how to extend our approach to solve a restatement of the distributed synthesis problem for open systems. Our synthesis makes use of the algebraic laws of CSP, i.e., L1-5 in Section 2.2 and the following derived ones. Law L6 is a direct consequence of law L4 and L5. Law L7 is the generalized form of law L6.

$$(P_1 \ {}_X\|_Y \ P_2) \ {}_{X \cup Y}\|_{Z \cup W} \ (P_3 \ {}_Z\|_W \ P_4) = (P_1 \ {}_X\|_Z \ P_3) \ {}_{X \cup Z}\|_{Y \cup W} \ (P_2 \ {}_Y\|_W \ P_4) \quad - \text{L6}$$
$$\|_{i=1}^{m}(\|_{j=1}^{n}(P_i^j, \Sigma_i^j), \bigcup_j \Sigma_i^j) = \|_{j=1}^{n}(\|_{i=1}^{m}(P_i^j, \Sigma_i^j), \bigcup_i \Sigma_i^j) \quad - \text{L7}$$

It is important that during synthesis the global state machine is never constructed. That is, we need to identify a local process, equipped with local liveness conditions, for each object in the system without first constructing the global one. In the following, we prove that in our context, it is sound to associate the liveness condition (modality on locations) with the local processes instead of the global process. Equivalently, we want to show that the following lemmas hold for all $P_1, P_2 : \mathcal{P}$.

$$
\begin{array}{ll}
MainLoca(talk, env, 0, y) \mathrel{\widehat{=}} talk \rightarrow syn.talk.y \rightarrow \textsc{Run} & \text{– by } \mathbf{R4'} \\
MainLoca(talk, user, 0, y) \mathrel{\widehat{=}} talk \rightarrow syn.talk.y \rightarrow \textsc{Run} & \text{– by } \mathbf{R4'} \\
MainLoca(talk, cover, 0, y) \mathrel{\widehat{=}} syn.talk.y \rightarrow \textsc{Run} & \text{– by } \mathbf{R4'} \\
MainLoca(talk, chip, 0, y) \mathrel{\widehat{=}} & \\
\qquad speakOff \rightarrow displayTime \rightarrow syn.talk.y \rightarrow \textsc{Run} & \\
MainLoca(talk, speaker, 0, y) \mathrel{\widehat{=}} speakOff \rightarrow \textsc{Run} & \\
MainLoca(talk, displayer, 0, y) \mathrel{\widehat{=}} & \\
\qquad displayTime \rightarrow setDisplayTime \rightarrow \textsc{Run} & \\
PreLoca(talk, env, 0, y) \mathrel{\widehat{=}} & \\
\quad (talk \rightarrow PreLoca(talk, env, 0, y)) \Box & \\
\qquad (fork.talk.y \rightarrow & \\
\qquad\quad (\mu X \bullet (syn.talk.y \rightarrow MainLoca(talk, env, 0, y) & \\
\qquad\qquad \Box\ talk \rightarrow X) & \\
\qquad\quad \triangledown_{ter.talk.y} \textsc{Run}) \,|[\, talk \,]|\, PreLoca(talk, env, 0, y+1)) & \text{– by } \mathbf{R5'} \\
PreLoca(talk, user, 0, y) \mathrel{\widehat{=}} & \\
\quad (open \rightarrow fork.talk.y \rightarrow syn.talk.y \rightarrow & \\
\qquad ((MainLoca(talk, user, 0, y) \triangledown_{ter.talk.y} \textsc{Run}) & \\
\qquad\quad |[\, open, close, talk \,]|\, PreLoca(talk, user, 0, y+1))) & \\
\quad \Box\ (close \rightarrow PreLoca(talk, user, 0, y)) & \\
\quad \Box\ (talk \rightarrow PreLoca(talk, user, 0, y)) & \\
\quad \Box\ (\mu X \bullet fork.talk.y \rightarrow & \\
\qquad (((open \rightarrow syn.talk.y \rightarrow MainLoca(talk, env, 0, y)) & \\
\qquad\quad \Box\ (close \rightarrow X \ \Box\ talk \rightarrow X)) \triangledown_{ccv.talk.y} \textsc{Run}) & \\
\qquad\quad |[\, open, user, talk \,]|\, PreLoca(talk, user, 0, y+1)) & \text{– by } \mathbf{R9'} \\
PreLoca(talk, cover, 0, y) \mathrel{\widehat{=}} & \\
\quad (open \rightarrow fork.talk.y \rightarrow ((PreLoca(cover, 1, y) \triangledown_{ter.talk.y} \textsc{Run}) & \\
\qquad |[\, open, close, coverOpened \,]|\, PreLoca(talk, user, 0, y+1))) & \\
\quad \Box\ (close \rightarrow PreLoca(talk, cover, 0, y)) & \\
\quad \Box\ (coverOpened \rightarrow PreLoca(talk, cover, 0, y)) & \\
\quad \Box\ (\mu X \bullet fork.talk.y \rightarrow ((open \rightarrow PreLoca(talk, cover, 1, y)) & \\
\qquad \Box\ (close \rightarrow X) \ \Box\ (coverOpened \rightarrow X)) \triangledown_{ter.talk.y} \textsc{Run}) & \text{– by } \mathbf{R8'} \\
PreLoca(talk, cover, 1, y) \mathrel{\widehat{=}} & \\
\quad (coverOpened \rightarrow syn.talk.y \rightarrow MainLoca(talk, cover, 0, y)) & \\
\quad \Box\ (close \rightarrow ter.talk.y \rightarrow \textsc{Run}) \ \Box\ (open \rightarrow ter.talk.y \rightarrow \textsc{Run}) & \text{– by } \mathbf{R7'}
\end{array}
$$

Figure 5.2: Synthesized processes

Lemma 5.4.1 $\mathcal{F}(P_1 \ {}_X\|_Y \ P_2) \supseteq \mathcal{F}(P_1) \ {}_X\|_Y \ \mathcal{F}(P_2)$

Proof: *This lemma is proved by the following:*

$$
\begin{aligned}
&\forall\, tr : \Sigma^* \bullet tr \in \mathcal{F}(P_1) \ {}_X\|_Y \ \mathcal{F}(P_2) \\
&\Rightarrow \nexists\, e_1 : \tilde{\Sigma} \bullet (tr \upharpoonright \Sigma_{P_1}) \frown \langle e_1 \rangle \in traces(P_1) \wedge \nexists\, e_2 : \tilde{\Sigma} \bullet (tr \upharpoonright \Sigma_{P_2}) \frown \langle e_2 \rangle \in traces(P_2) \\
&\Rightarrow \nexists\, e : \tilde{\Sigma} \bullet tr \frown \langle e \rangle \in traces(P_1 \ {}_X\|_Y \ P_2) \\
&\Rightarrow tr \in \mathcal{F}(P_1 \ {}_X\|_Y \ P_2) \qquad \square
\end{aligned}
$$

Intuitively, Lemma 5.4.1 states that if both components cannot engage in a signal at certain point of execution, then the composition cannot engage in the signal either. The reverse of Lemma 5.4.1 is not true. A counter example is: $\langle\rangle$ is a trace of $\mathcal{F}(P_1 \ {}_{\{\widehat{a},b\}}\|_{\{\widehat{a},c\}} \ P_2)$ but not $\mathcal{F}(P_1) \ {}_{\{\widehat{a},b\}}\|_{\{\widehat{a},c\}} \ \mathcal{F}(P_2)$, where $P_1 \mathrel{\widehat{=}} (\widehat{a} \rightarrow \textsc{Stop} \ \Box \ b \rightarrow \textsc{Stop})$ and $P_2 \mathrel{\widehat{=}} c \rightarrow \textsc{Stop}$.

Lemma 5.4.2 $\mathcal{F}(\|_{i=1}^{m}(\|_{j=1}^{n}(P_i^j, \Sigma_i^j), \bigcup_j \Sigma_i^j)) \supseteq \|_{j=1}^{n}(\mathcal{F}(\|_{i=1}^{m}(P_i^j, \Sigma_i^j)), \bigcup_i \Sigma_i^j)$ $\square$

Lemma 5.4.2 can be proved straightforwardly using law L7 and the generalized form of Lemma 5.4.1. It states that we may rewrite the global liveness condition in terms of local liveness conditions soundly, i.e., the accepting states of each object in the system can be identified locally without referring to the global state. We are now ready to synthesize distributed processes which group the local behaviors of each object. For simplicity, we assume that all events appear in at least one of the universal charts.

$$
\begin{aligned}
\mathcal{F}(\mathcal{I}) &= \mathcal{F}(\|_{u \in \mathcal{S}}(Chart(u), \Sigma_u \cup \Sigma'(u)) \setminus \Sigma') && \text{– \textbf{R12'}} \\
&= \mathcal{F}(\|_{u \in \mathcal{S}}((\|_i Instance(u,i), \Sigma_u^i \cup \Sigma'(u)), \Sigma_u \cup \Sigma'(u)) \setminus \Sigma') && \text{– \textbf{R11'}} \\
&= \mathcal{F}(\|_i((\|_{u \in \mathcal{S}}(Instance(u,i), \Sigma_u^i \cup \Sigma'(u))), (\textstyle\bigcup_i \Sigma_u^i) \cup \Sigma')) && \text{– L7} \\
&\supseteq \|_i(\underline{\mathcal{F}((\|_{u \in \mathcal{S}}(Instance(u,i), \Sigma_u^i \cup \Sigma'(u))))}, (\textstyle\bigcup_i \Sigma_u^i) \cup \Sigma') && \text{– Lemma 5.4.2}
\end{aligned}
$$

We remark that the underlined portion of the process identifies the local behavior of an object in the system equipped with local liveness conditions (and $(\bigcup_i \Sigma_u^i) \cup \Sigma'$ is its alphabet). The soundness is an immediate consequence of Theorem 5.3.1. If there are events that do not appear in any of the universal charts (but do in some existential chart), we may localize them to the corresponding instance processes straightforwardly.

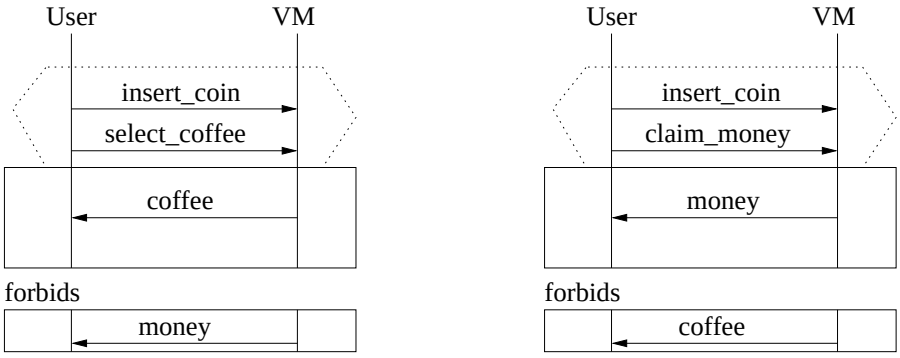


Figure 5.3: Unsatisfiable universal charts

So far, environmental objects are not distinguished from system objects. For instance, in Example 5.3.2, $user$ is considered as part of the system and the local process capturing its behaviors is synthesized in the same way as for object $cover$. Thus, we handle only closed systems but not open systems, i.e., systems that interact with the environment frequently. The synthesis problem for closed systems is often referred to as satisfiability, i.e., whether the language of a specification is non-empty, or equivalently if considering the environment as part of the system, whether there is a benevolent environment in which some implementation can be deployed in order to fulfill the specification. Synthesis for open systems, however, asks whether there is an implementation that can be deployed in any malevolent environment. In literature, the synthesis problem for open systems has long been recognized as a hard problem. It is even harder to synthesize distributed implementations without constructing the global state machine, i.e., undecidable in almost all interesting settings [152, 98, 122, 99]. Thus, we take a lightweight approach to tackle the problem.

Figure 5.3 illustrates the intuition behind our method. It shows two simple universal charts of a vending machine. This example is borrowed from [18], where it is used to illustrate the difference between synthesis of closed systems and synthesis of open systems. These two charts are unsatisfiable under the assumption that the implementation should deploy in any environment and $user$ is considered as part of the environment. For instance, considering the following sequence of

environmental events $\langle insert\_coin, select\_coffee, claim\_money \rangle$, neither of the universal charts can be satisfied. In practice, however, the $claim\_money$ event is typically blocked after event $select\_coffee$ and before event $insert\_coin$. In general, when system engineers design systems, implicit assumptions on the environment are often made (enforced later by blocking the user-interface at a certain time, using a queue to delay the arrival of the environmental events, etc.). Therefore, instead of synthesizing an implementation that works in any environment (which is certainly hard to do and unlikely to be successful), we synthesize one that works in the intended environment. In other words, we deal with a restatement of the synthesis problem 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.

In our method, objects are partitioned into either environmental objects or system objects. Events are also partitioned into either environmental events, written as E, 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 which requires the participation of an environmental object. The system designer is asked for a modeling of the intended environment, preferably using universal charts, which captures all implicit assumptions on the environment. We may then synthesize implementations that can be deployed in the intended environment or any refinement of it. Different from dealing with closed systems, the implementation should not restrict the intended environment in any way.

Given the modeling of the environment, local processes for the environmental objects are firstly synthesized in the same way that system objects are synthesized. We then verify that the synthesized process for the environment (alphabetized parallel composition of all the environment objects), denoted as Env, simulates the user-supplied modeling, e.g., ENV.

$$ENV \sqsupseteq Env \setminus ((\Sigma \setminus E) \cup \Sigma')$$

By hiding all internal communications and local actions and special synchronization, an implicit assumption, i.e., the internal computation is infinitely faster than the incoming of external stimuli, is enforced[1]. Using FDR, we may automatically verify the refinement relation of the two pro-

[1] Without the assumption, forbidden environmental events will not be possible.

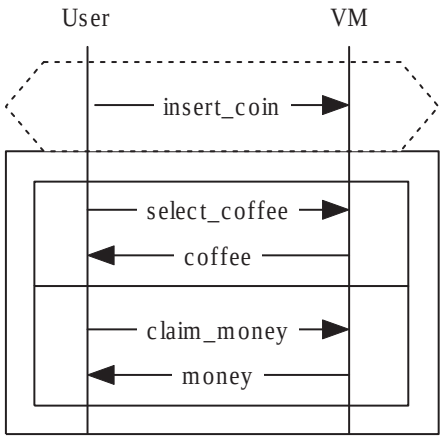


Figure 5.4: Environment modeling

cesses. This way, we make sure the implementation behaves correctly in the intended environment. From another point of view, the processes synthesized for environmental objects are indeed system processes which monitor the interaction between the environment and the system and trigger the appropriate special events at the proper point of execution. The refinement relationship therefore ensures that no interaction is missed.

Example 5.4.3 (Environment modeling) In the vending machine example, the assumption on the environment (users) can be modeled as the universal chart presented in Figure 5.4. After inserting the coin, the user shall either request coffee and wait for the coffee or claim the money and wait for it. Thus, the event $claim\_money$ is temporally disabled after the event $select\_coffee$. Semantically, the chart is equivalent to the following CSP process.

$$\begin{aligned} ENV \mathrel{\widehat{=}} & \; insert\_coin \rightarrow \\ & (select\_coffee \rightarrow coffee \rightarrow ENV \;\Box\; claim\_money \rightarrow money \rightarrow ENV) \end{aligned}$$

end

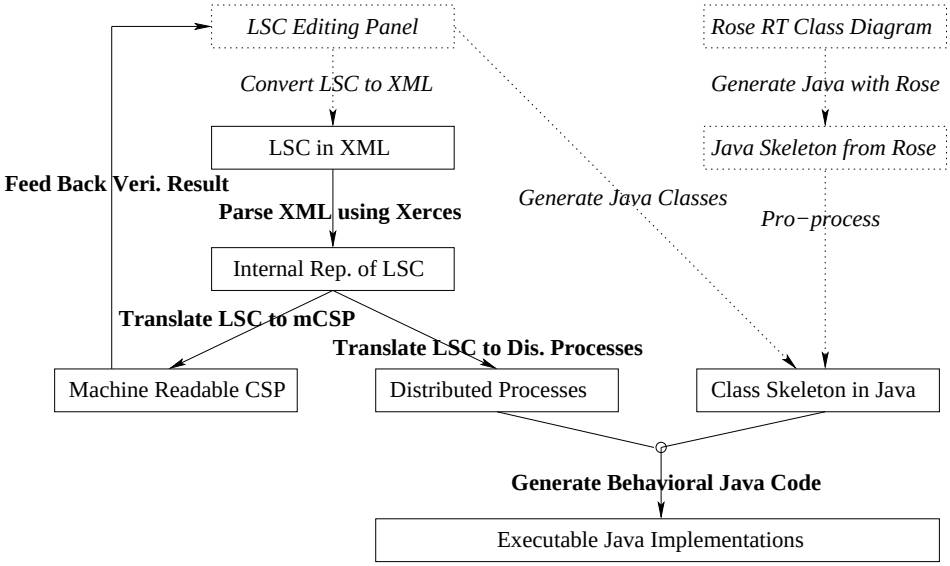


Figure 5.5: Workflow of the synthesis

5.5 Generating Implementations

An experimental tool has been implemented using XML and JAVA technology to automate our approach. One of the benefits of using CSP as an intermediate language is that there exist CSP-based process oriented design patterns for concurrency implemented in JAVA, i.e., in programming engineers' terms, JAVA libraries for CSP. Two libraries are available, Communicating Threads for JAVA (CTJ)[2] and CSP for JAVA (JCSP) [158]. We implemented our approach using the JCSP package mainly for its support of barrier synchronization. After identifying the local behaviors of each object, executable codes are generated by translating the distributed processes to JAVA programs making use of CSP-like constructs provided by JCSP.

The schematic workflow of the synthesis is illustrated in Figure 5.5. An italic font indicates works under development, e.g., a user-friendly drawing panel for user to introduce and refine LSC models and associate objects with local data variables, a pre-processing module to translate the JAVA code

[2] `www.ce.utwente.nl/javapp/information/Communicating_Java_Threads/Default.html`

generated from class diagrams drawn in Rational Rose [124], etc. The class skeleton is either generated from user inputs to the drawing panel (where users have to introduce the object type before adding an object instance to the system, introduce a local data variable before using it in the condition, etc.), or generated from class diagrams using Rose and pre-processed. The data aspects of an object are defined as a separate class. That way, we allow multiple instances of the same object type in the system.

Every local action or condition is implemented as a method in the respective class. Therefore, the system makes a method call whenever a local action or a condition is encountered during execution. The implementation detail of the methods is supplied by the user. For each object in the class, a separate class is defined to realize its local dynamic behaviors. Each object is associated with a set of channels for communicating with the rest of system and a set of synchronization barriers to realize the CSP-style synchronization between $Instance(u, i)$. Each communication event is associated with a channel definition. Its occurrence in the charts is translated into a $read()$, $write()$ operation on the respective channel. In JCSP, we may specify the capacity of the channel as either 0 or more, which saves us the work of modeling the buffers for asynchronous communication.

Each shared event between $Instance(u, i)$, either a communication event or local action, is associated with a synchronization barrier. The shared event is only engaged after the respective barrier is synchronized by all instances whose alphabet includes the event. The special events, ter_u, hcv_u, syn_u, and $fork_u$, are implemented as synchronization barriers at the system level since they synchronize different objects. We remark that hcv_u is redundant for the purpose of simulation as we may terminate the JAVA virtual machine whenever a hot condition is violated and proper information is displayed. Simple timing requirements in LSC like $settimer$ and $timeout$ are supported using the $CSTimer$ offered by JCSP.

Example 5.5.1 (Code generation) The high-level program generated for object $Chip$ is presented in Figure 5.6. There is a direct mapping between the programs and the processes. For instance, the first part of the top-most class contains channel definition, one for each event in the alphabet. After that, there is declaration for each object in the system. The system is the parallel composition of the

```
class PhoneSystem implements CSProcess {
    //Barriers for synchronization of entering and exiting
    private Barrier barrier_OpenCover = new Barrier ();
    private Barrier barrier_CloseCover = new Barrier ();
    private Barrier barrier_Receive = new Barrier ();
    private Barrier barrier_Talk = new Barrier ();

    public void run () {
       //Channels connecting objects
       One2OneChannel open = new One2OneChannel ();
       One2OneChannel coverOpened = new One2OneChannel ();
       ......
       //Components in the system
       User inst_User = new User (open, close, talk);
       Cover inst_Cover = new Cover (open, coverOpened, close, coverClosed);
       Chip inst_Chip = new Chip (coverOpened, displayMenu, coverClosed,
            displayTime, incomingCall, startRing, displayCallerID, speakerOff);
       Speaker inst_Speaker = new Speaker(startRing, speakerOff);
       Display inst_Display =
                new Display(displayMenu,displayTime,displayCallerID);
       Env inst_Env = new Env(incomingCall, talk);
       //System initialization
       CSProcess[] parArray = new CSProcess[]
           {inst_User,inst_Cover,inst_Chip,inst_Speaker,inst_Display,inst_Env};
       Parallel sys = new Parallel (parArray);
       sys.run();
    }
} class Cover implements CSProcess {
    //Channel and barrier definitions
    ......
    //Sub-processes
    private Cover_OpenCover OpenCover;
    private Cover_CloseCover CloseCover;
    private Cover_Talk Talk;
    //Data
    private Cover_Data data = new Cover_Data();
    //Channels
    public Controller (One2OneChannel open, One2OneChannel coverOpened,
            One2OneChannel close, One2OneChannel coverClosed) {
       ......
       OpenCover = new Cover_OpenCover (data,open,coverOpened);
       CloseCover = new Cover_CloseCover (data,close,coverClosed);
       Talk = new Cover_Talk (data,open,coverOpened,close);
    }
    public void run () {
       new Parallel (new CSProcess[]{OpenCover,CloseCover,Talk}).run();
    }
}
```

Figure 5.6: Example synthesized JAVA program

instance threads. The dynamic behaviors of each object is encapsulated in its class definition. **end**

5.6 Summary

Compared with Hoare's grand challenge on verifying compilers [78], mechanized generation of programs from high-level specification is an alternative and equally challenging approach to correct programs. This work can as well be viewed as a way of achieving Harel's dream as in [66], i.e., synthesizing codes all the way from scenarios. In [17], Bontemps and Schobbens showed that both verification and synthesis of LSC are computationally expensive in their theoretical study. For the verification problem, our solution is to make use of existing mature model checker instead of building one from scratch which allows applying mature techniques for the hard task. Our solution to the synthesis problem replies on using a set of additional events. The key idea is of using the bounded set of synchronous events to monitor global execution locally, and yield a distributed design without constructing the global state machine. In general, our approach is sound and as complete as possible (some assumptions due to practical concerns may harm the completeness, e.g., finite overlapping activations of the same chart). The main contribution of this work includes a complete system engineering method that automates the generation of implementation all the way from LSC, a set of generalized interpretation rules, and a lightweight approach to handle open systems, etc.

Being based on LSC, our method avoids implied scenarios. It is explained in the following using CSP notions. Let $\{M_j\}$ where $1 \leq j \leq n$ be the set of MSCs. Let M_j^i where $1 \leq i \leq m$ be the process capturing the behavior of instance i in the chart M_j. An implementation of the specification shall therefore exhibit exactly the following behaviors:

$$(\|_{i=1}^{m}(M_1^i, \Sigma_1^i)) \,\Box\, (\|_{i=1}^{m}(M_2^i, \Sigma_2^i)) \,\Box \cdots \Box\, (\|_{i=1}^{m}(M_n^i, \Sigma_n^i))$$

where Σ_j^i contains exactly the events of the instance appeared in the chart M_j. The distributed object system inferred from a set of MSCs should be composed of finite state processes modeling each of the objects appeared in the scenarios. Each object should exhibit as sequences of events at least all scenarios projected to the time line of that component. Formally, the behavior of an object shall at least exhibit the behaviors captured by the following expression: $M_1^i \,\Box\, M_2^i \,\Box \cdots \Box\, M_n^i$. The

existence of implied scenarios may be explained using CSP algebraic laws as the following (proved by **T6** and **T8** in Section 3.2.2):

$$\begin{aligned} &traces((\|_{i=1}^{m}(M_1^i, \Sigma_1^i)) \,\Box\, (\|_{i=1}^{m}(M_2^i, \Sigma_2^i)) \,\Box \cdots \Box\, (\|_{i=1}^{m}(M_n^i, \Sigma_n^i))) \\ &\quad \subseteq traces(\|_{i=1}^{n}(M_1^1 \,\Box\, M_2^1 \,\Box \cdots \Box\, M_n^1, \Sigma^i))) \end{aligned}$$

where Σ^i contains all events of object i. The occurrence of additional scenarios is because the scenario-based model describes allowed system behaviors from a global, system-wide perspective, whereas in the distributed object processes each agent acts locally based on local information. Contrasted with MSC, which captures only examples of system behaviors, an LSC universal chart specifies mandatory behaviors. In other words, a universal chart constrains all behaviors of the system. Therefore, the precise behaviors of an implementation are captured by the parallel composition (in contrast to choice) of the universal charts (refer to **R12'**). If an instance i is missing from an MSC M_j, no event regarding this instance can be engaged in the scenario, i.e., $\Sigma_j^i = \varnothing$. However, the semantics of universal charts state that a universal chart constrains only its visible events and invisible events can occur infinitely between any two consecutive occurrence of visible events, i.e., $M_j^i = \textsc{Run}_{\Sigma_j^i}$. This serves as the basis of our transformation in Section 5.4.

As for related works, the synthesis problem of MSC has been studied extensively [5, 3, 87, 153, 86, 87, 72]. The synthesis problem of LSC was initially discussed by Harel and Kugler in [68], in which they tackled the problem by defining the notion of consistency of LSC models. Their approach starts with constructing a *global system automaton* and decomposes it by different means (refer to [68] for details). 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 order semantics. The characteristic of our work is that we use CSP algebraic laws to identify local behaviors of each object without ever constructing the global state machine.

In [18], Bontemps, Schobbens and Löding 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 consistency of their LSC. Their work is later extended to handle all LSC constructs but unbounded loop in [13]. In our

approach, almost all LSC constructs are supported except complex time-related ones, which deserve a complicated discussion and thus are left to the future works. We remark that the same result can be derived using automata (e.g., Büchi Automata [23]) with a painfully complicated procedure.

In [17], Bontemps and Schobbens investigated the complexity of various problems associated with LSC. The results are pretty negative, i.e., they showed that centralized model-checking of LSC is Co-NP-complete, the distributed model-checking is PSPACE complete and the distributed realization problem is undecidable. In our work, we use a set of special events (bounded by the maximum number of overlapping activation of the universal charts and the number of the universal charts) to avoid undecidability. Thus, our work can be viewed as a lightweight approach. In [69], Harel, Kugler and Pnueli re-investigated the synthesis problem of LSC by adopting a lightweight approach as well, i.e., they generate Statecharts from LSC and then verify them for correctness, and thus avoid undecidability. A similar approach is evidenced in [16], where Bontemps and Egyed proposed a technique coupling translation and verification to cope with undecidability. We remark that such an approach certainly works for our approach as well except that we must deal the complexity of model-checking of complicated distributed systems. In addition, there is the work in [85], which synthesizes a timed Büchi Automaton from a single chart only. What makes our goal both harder and more interesting is in the treatment of a set of charts, not just a single one.

Besides, a remotely related problem known as controller synthesis has been studied for many years both from a computer science and control-theoretic perspective [27, 23, 121, 122, 99, 123]. However, the research on controller synthesis has been focused on automata but not scenario-based specification languages like LSC.

Chapter 6

Synthesis from State-based Specification

The Caterpillar was the first to speak.
'What size do you want to be?' it asked.
'Oh, I'm not particular as to size,'
Alice hastily replied;
'only one doesn't like changing so often, you know.'
- *Alice's Adventures in Wonderland*, Lewis Carroll

In the last chapter, we offered a mechanized way of generating prototype implementations for system engineering starting with scenario-based specification. The approach is of special interest because scenario-based diagrams are widely used as a specification language in early stage of system development. However, both LSC and MSC have limited expressiveness in specifying data and functional aspects of complex systems. Formal specification languages like Z/Object-Z offer an alternative state-based high-level system modeling. They can be used in early stages of system development to specify a data and functional model of the system, and therefore serve as another good starting point for synthesis of implementation.

The notion of separation of concerns is a common technique to fight complexity in system development. A practical approach is to focus on system functionalities before modeling the dynamic

control flow of the system. An early stage data model typically contains a set of objects/classes, data variables and the associated abstract operations in each class. Those models can be documented using Class Diagrams or formally modeled as Object-Z [161, 137] specifications. In this chapter, we investigate ways of synthesizing implementable designs (i.e., a control program in the form of finite state machines) from Object-Z specifications.

6.1 Introduction

Object-Z with history invariants can present precise and abstract models for complex systems. A system design in Object-Z is relieved from behavioral aspects of the system. The system behavior patterns are implicitly embedded within state/operational constraints and, additionally, history invariants. However, without explicit system behavior representations, it is difficult to implement such abstract models. Thus, we propose a sound and systematic approach to automatically extract explicit implementable system behaviors, as a control program to restrict the sequences of invocation of operations, from Object-Z specifications. The ultimate goal of our work is to generate implementations from high-level designs in Object-Z automatically.

An Object-Z specification captures safety requirements by specifying class invariants and pre/post-conditions for data operations. Liveness requirements are captured by history invariants. We generate finite state machines that are guaranteed to satisfy both sets of requirements. Additionally, because Object-Z distinguishes external variables (variables followed by a question mark) from state variables, it can be used to model open systems. Crucial requirements for open systems are also to be satisfied by the synthesized state machines, i.e., the state machines should not introduce fresh deadlocks and should work correctly in any environment. We call such state machines realizations of the Object-Z specification.

In order to handle Object-Z specifications with infinite data space, a predicate abstraction schema is developed to build an abstract finite state machine from an Object-Z specification. All behaviors of the concrete Object-Z specification is allowed in the abstract state machine. The number of abstract states is bounded by the number of predicates for abstraction. A weak abstract relation is

used so that the abstraction can be automated by general theorem provers like PVS [117] paying a reasonable price. Furthermore, the raw state machine is refined to satisfy additional requirements. Finally, an Object-Z specification is realized as a finite state machine with its transitions as guarded function calls. The soundness is proved by showing that there is a fair simulation relation from the realization to the specification. A tool is implemented in JAVA to demonstrate our method.

The reason why our approach is beneficial is twofold. Firstly, finite state machines are closer to implementations than Object-Z models, i.e., they are implementable. In our setting, a complete implementation of the system may be generated if the implementation of each operation in isolation is supplied. This conforms to one of the principles of object-oriented analysis and design, i.e., procedural thinking should be postponed as long as possible. Secondly, our realization is "minimally" restrictive so that further refinements are possible without breaking any of the requirements.

6.2 Extracting Raw State Machine

In this section, we discuss how to extract a finite state machine realization from an Object-Z class. A finite state machine is an abstract machine that has only a finite constant amount of memory. It can be viewed as a flattened UML Statechart. There are finite many states and each state has transitions to states. Transitions are triggered by observable events. Additionally, there are one or more initial states and final states.

Definition 13 *A state machine*[1] *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 function and* I *labels each state with a Boolean formula over a given set of propositions.*

The Boolean formula labeled with a state is also referred to as state invariant. Graphically, an initial state is indicated by an arrow from nowhere. A double-lined circle represents an accepting state. A

[1] It is also referred as labelled Kripke Structure.

run of the 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: $\forall\, i : \mathbb{N} \mid i \geq 1 \bullet (s_i, e_i, s_{i+1}) \in T$ and $s_1 \in S_0$. An accepting run is a finite run ending with an accepting state or an infinite one where some 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 14 *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 $M_1 \parallel M_2$ 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 conditions:*

- $(s_1, s_2) \in M.S \wedge (s_1, e, s_1') \in M_1.T \wedge e \notin M_2.\Sigma \Rightarrow ((s_1, s_2), e, (s_1', s_2)) \in M.T$
- $(s_1, s_2) \in M.S \wedge (s_2, e, s_2') \in M_2.T \wedge e \notin M_1.\Sigma \Rightarrow ((s_1, s_2), e, (s_1, s_2')) \in M.T$
- $(s_1, s_2) \in M.S \wedge (s_1, e, s_1') \in M_1.T \wedge (s_2, e, s_2') \in M_2.T$
  $\Rightarrow ((s_1, s_2), e, (s_1', s_2')) \in M.T$

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 15 *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:*

C1 $\forall\, s : M_1.S_0 \bullet \mathcal{R}(s) \in M_2.S_0$
C2 $\forall (s_1, e, s_2) \in M_1.T;\ s_1' : M_2.S \mid \mathcal{R}(s_1) = s_1' \bullet$
$\exists\, s_2' : M_2.S \bullet (s_1', e, s_2') \in M_2.T \wedge \mathcal{R}(s_2) = s_2'$
C3 $\forall\, s : M_1.F \bullet \mathcal{R}(s) \in M_2.F$

Informally, **C1** states that there is one initial state in M_2 corresponding to every initial state in M_1. **C2** states if M_1 can engage in an event at certain state, M_2 should be able to simulate the transition at the corresponding state. **C3** guarantees that all final states in M_1 are simulated in M_2. A similar definition appeared in [41]. Later development can be found in [75]. 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 [75].

6.2.1 Predicate Abstraction

As introduced in Chapter 2, the operations of a class form a named collection of relations, which determine a transition system in which an operation may fire exactly when its Z precondition is satisfied. Due to the blocking semantics of Object-Z, an operation is blocked outside its precondition. The semantic model of an Object-Z class consists of all the sequences of operations/events which can be performed by objects of the class. The implicit behavioral model of an Object-Z class can be expressed as the following CSP process: let $Behavior$ be the process capturing all possible behaviors of instances of the class,

$$
\begin{aligned}
&Behavior \mathrel{\widehat{=}} \mu\, R \bullet \\
&\quad ([\mathrm{pre}(Operation_1)] \bullet Operation_1 \;\Box \\
&\quad [\mathrm{pre}(Operation_2)] \bullet Operation_2 \;\Box \\
&\quad \ldots \Box \\
&\quad [\mathrm{pre}(Operation_n)] \bullet Operation_n);\; R
\end{aligned}
$$

The state space of an Object-Z class may be infinite. For example, a $Queue$ object may contain infinite items. However, an implementable control structure may only contain a finite number of control states. It restricts the behaviors of an object (specified by an Object-Z class) based abstract interpretations of the data variables. For instance, Figure 6.1 is an abstract interpretation of $Queue$ objects in which only the number of items (not the actually content) in the queue is concerned. We present a method to calculate predicate abstraction of an Object-Z class.

Given a finite set of predicates P (in terms of the state variables) for abstracting an Object-Z class, the set of abstract states, denoted as S_a, contains conjunctions of subsets of the predicates in P:

$$S_a \mathrel{\widehat{=}} \{x \mid \exists\, X \subseteq P \bullet x = \bigwedge(X \cup \{\neg\, e \mid e \in P \setminus X\})\}$$

An abstract state groups all possible valuation of the state variables satisfying the predicate x. For instance, the state labeled with $\#items > max$ in Figure 6.1 groups all instances of state schema

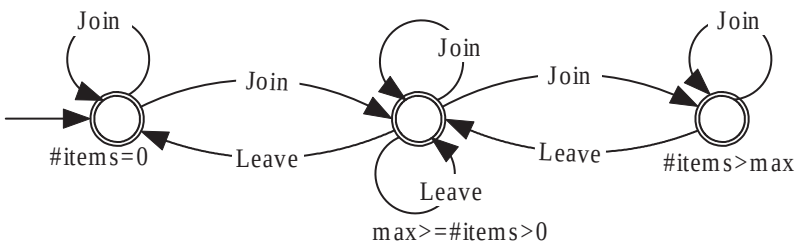


Figure 6.1: Abstraction of $Queue$

in $Queue$ where the number of items in $items$ is greater than max. For simplicity, we require that the set of predicates for abstraction includes the predicate in the initial schema.

Example 6.2.1 (Abstract states) Let $P \mathrel{\widehat{=}} \{\#items = 0, \#items \leq max\}$. The set of abstract states is (assuming $max > 0$):

$$S_a \mathrel{\widehat{=}} \{\#items = 0, max \geq \#items > 0, \#items > max\}$$

The abstract state $\#items = 0 \wedge \#items > max$ has been removed because it is infeasible. The abstract initial state of $Queue$ class is $\mathcal{S}(\textsc{Init}) \mathrel{\widehat{=}} \#items = 0$. **end**

Given an operation, it is necessary to find out the abstract states where an operation can be invoked without violating its precondition and the abstract states which can be reached by applying the operation. We define a function $\mathcal{W}$ to compute the weakest formula over P which implies a given predicate p.

$$\begin{array}{|l} \mathcal{W} : Predicate \rightarrow \mathbb{P}\, Predicate \\ \hline \forall\, p : Predicate \bullet \mathcal{W}(p) = \{x \in S_a \mid x \Rightarrow p\} \end{array}$$

The motivation is that if p is the precondition of an operation, then $\mathcal{W}(p)$ is the largest set of abstract states where the operation can be invoked without violating its precondition. In addition, we define a function $\mathcal{S}$ to compute the abstract states where a given predicate might be true.

$$\begin{array}{|l} \mathcal{S} : Predicate \rightarrow \mathbb{P}\, Predicate \\ \hline \forall\, p : Predicate \bullet \mathcal{S}(p) = S_a \setminus \mathcal{W}(\neg\, p) \end{array}$$

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 by applying the operation at the state. Function $\mathcal{S}$ works by pruning all states where the predicate is proved to be false. Thus, all states where the predicate is true are present in the result, together with states where we are uncertain if the predicate is true.

Function $\mathcal{S}$ is used to automatically construct abstractions of an Object-Z specification. The INIT schema is abstracted as $\mathcal{S}(\text{INIT})$, so that every possible initial state is grouped in the abstract initial state. We calculate abstraction of an operation by abstracting its precondition and postcondition. The precondition is replaced by $\mathcal{S}(\text{pre } Operation)$, i.e., all abstract states where the operation might be applied. We remark that this way the abstract finite state machines allows more behaviors (than using $\mathcal{W}(\text{pre } Operation)$). It remains sound because of the blocking semantics (contrasted with Z semantics of precondition), i.e., it is no harm to apply an operation outside its domain.

Example 6.2.2 (Abstract precondition) The abstract precondition of operation $Leave$ is:

$$\begin{aligned}
\mathcal{S}(\text{pre } Leave) &\mathrel{\widehat{=}} \mathcal{S}((\exists\, items' : \text{seq}\, Package;\ item! : Package \bullet \\
&\qquad items = \langle item! \rangle \frown items') \setminus \{item!\}) && \text{– def. of pre} \\
&\mathrel{\widehat{=}} S_a \setminus \mathcal{W}((\forall\, items' : \text{seq}\, Package;\ item! : Package \bullet \\
&\qquad items \neq \langle item! \rangle \frown items') \setminus \{item!\}) && \text{– def. of } \mathcal{S} \\
&\mathrel{\widehat{=}} S_a \setminus \{\#items = 0\} && \text{– def. of } \mathcal{W} \\
&\mathrel{\widehat{=}} \{max \geq \#items > 0, \#items > max\}
\end{aligned}$$

Thus, operation $Leave$ is applicable only at the two abstract states where $\#items > 0$. **end**

For each abstract state $s_a : S_a$ satisfying the abstract precondition, we calculate the abstract postcondition as $\mathcal{S}(\text{post}(Operation, S_a))$ so that all possible post-states are reachable in the abstract finite state machine.

Example 6.2.3 (Abstract postcondition)

$$\begin{aligned}
&\mathcal{S}(\text{post}(Leave, max \geq \#items > 0)) \\
&\quad \mathrel{\widehat{=}} \mathcal{S}(\#items \leq max \wedge \#items > 0 \wedge items = \langle item! \rangle \frown items') \\
&\quad \mathrel{\widehat{=}} S_a \setminus \mathcal{W}(\#items > max \vee \#items \leq 0 \vee items \neq \langle item! \rangle \frown items') \\
&\quad \mathrel{\widehat{=}} S_a \setminus \{\#items' > max\} \\
&\quad \mathrel{\widehat{=}} \{max \geq \#items' > 0, \#items = 0\}
\end{aligned}$$

The above computes the postcondition of operation $Leave$ at the abstract state where $max \geq \#items > 0$. **end**

We abstract every operation in the class to construct an abstract graph. For instance, the abstraction of the $Queue$ class defines the state transition system in Figure 6.1. Note that abstraction introduces non-determinism and spurious sequences of operations. For example, applying the $Join$ operation at the middle state may result in a state where the number of items in the queue is larger than max or no larger than max.

However, both function $\mathcal{W}$ and $\mathcal{S}$ in our context (first order logic) 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 $\mathcal{W}$ shall contain at most the set of abstract states in $\mathcal{W}(p)$, whereas the approximation of the function $\mathcal{S}$ shall contain at least states in $\mathcal{S}(p)$. Therefore, our abstraction is robust with respect to Object-Z refinement, i.e., strengthening post-condition. In our prototype, we make use of the theorem prover PVS [117] to compute such approximations in order to construct an abstract state machine by paying a reasonable prize. Despite the limited power of proving, an abstract state transition system covers all possible sequences of operations of the concrete one.

Definition 16 *Given a set of predicates P, $M_a \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$ is an abstraction of the Object-Z class only if $S \mathrel{\widehat{=}} S_a$ and $S_0 \mathrel{\widehat{=}} \mathcal{S}(\textsc{Init})$ and $F \mathrel{\widehat{=}} S$ and Σ is the set of operation schemas and I labels a state with itself and $T \mathrel{\widehat{=}} \{(s_1, e, s_2) : S \times \Sigma \times S \mid s_1 \in \mathcal{S}(pre(e)) \wedge s_2 \in \mathcal{S}(post(e, s_1))\}$.*

6.2.2 Generating Raw State Machines

Our method begins with constructing a finite Büchi automaton from the history invariant. An efficient tool to convert LTL formulæ into optimized Büchi automata is Somenzi and Bloem's Wring [140]. For example, Figure 6.2 shows the Büchi Automaton constructed from the LTL formulæ in the $FairBoundedQueue$ class. Both states are initial states. The state labeled with $\#items = 0$ is a final state. Transitions are not labeled in Büchi Automata.

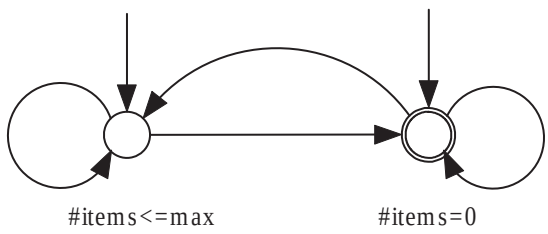


Figure 6.2: Büchi Automaton

Definition 17 *A Büchi automaton is a 5-tuple* (S, S_0, T, F, I) *where* S *is a finite set of states,* $S_0 \subseteq S$ *is a set of initial states,* $T : S \times S$ *is a transition relation,* $F \subseteq S$ *is a set of final states and* I *is a labeling function which labels a state with a Boolean formula.*

Meanwhile, a raw finite state machine is constructed from the Object-Z class as discussed in Section 6.2.1. We require that the predicates for abstraction include propositions in the history invariants and the initial schema. Every state in the raw state machine is a final state because in Object-Z semantics, an object may wait infinitely long before applying an enabled operation. The product of the state machine and the Büchi automaton is then constructed.

Definition 18 *A state machine* $(S, S_0, T, F, \Sigma, I)$ *is a product of a state machine* M *and a Büchi automaton* B *if it satisfies the following condition:* $\Sigma \mathrel{\widehat{=}} M.\Sigma$, $I \mathrel{\widehat{=}} M.I$ *and,*

$$
\begin{aligned}
&S \mathrel{\widehat{=}} \{(s_s, s_b) : M.S \times B.S \mid M.I(s_s) \Rightarrow B.I(s_b)\} \\
&S_0 \mathrel{\widehat{=}} \{(i_s, i_b) : M.S_0 \times B.S_0 \mid M.I(i_s) \Rightarrow B.I(i_b)\} \\
&T \mathrel{\widehat{=}} \{((s_s^1, s_b^1), e, (s_s^2, s_b^2)) : S \times \Sigma \times S \mid (s_s^1, e, s_s^2) \in M.T \wedge (s_b^1, s_b^2) \in B.T\} \\
&F \mathrel{\widehat{=}} \{(f_s, f_b) : M.F \times B.F \mid M.I(f_s) \Rightarrow \bigwedge B.I(f_b)\}
\end{aligned}
$$

Informally, a state in the Büchi automaton is unified with a state in the state machine if their labeling is consistent. Because all predicates in the history invariant are used for abstraction, the consistency testing of two states is a straightforward existence checking, i.e., whether the set of predicates labeled with a state is a subset of those of the other state. A state of the product is an initial state if and only if it is unified by two initial states. A labeled transition in the raw state machine is allowed

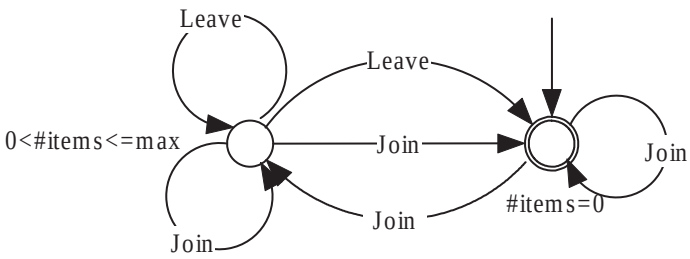


Figure 6.3: Product of the state machine and automaton

in the product if and only if there is a transition between the same starting state and ending state in the Büchi Automaton. For instance, Figure 6.3 is the product of the state machine in Figure 6.1 and the Büchi automaton in Figure 6.2.

6.3 Refining the Finite State Machine

A finite state machine is a (sound) realization of an Object-Z specification if the INIT schema is satisfied at every initial state, every operation is engaged with its precondition/postcondition fulfilled, and the history invariants are satisfied. For open systems, two additional requirements are crucial.

A_1: The finite state machine should not introduce any fresh deadlocks.

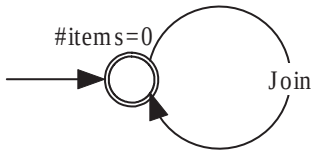
A_2: The finite state machine is not allowed to restrict the actions of the environment.

Both requirements have been discussed in various works of control theory [123, 98]. The first requirement is commonly referred as nonblocking. The second requirement is essential for systems constantly interacting with its environment. Informally, it requires that the state machine should be able to function correctly regardless of the environment. In this section, we present a systematic way of generating finite state machines that satisfy both the Object-Z specification and the two additional requirements.

6.3.1 Pruning Raw State Machines

The product of the raw state machine and the Büchi automaton satisfies the Object-Z specification with the history invariant. However, it may not be a valid realization of the Object-Z specification. There are two sources of possible errors. Firstly, because of requirement A_2, any finite state machine which satisfies the specification by restricting behaviors of the environment is not valid.

Example 6.3.1 (Problematic realization) The following is a problematic realization of instances of class $FairBoundedQueue$ presented in Section 2.1.2:



This finite state machine satisfies the safety properties of the Object-Z specification since it only contains part of the behaviors captured by the state machine in Example 2.1.10. By requiring that all $item?$ from the environment are expired, the queue remains empty all the time and thus trivially satisfies the history invariant. **end**

It is easy to see that synthesis of such a realization is not helpful at all. Such realization is removed systematically from the product by pruning states and transitions violating requirement A_1 and A_2. Secondly, abstraction introduces spurious sequences of events/operations. In the abstract state machine, an operation may be applied at states where its precondition is not satisfied or invocation of an operation may lead to states not satisfying the concrete postcondition. To solve the problem, each transition is equipped with a guard condition (if necessary) in the very last step. Formally,

Definition 19 *Let P be the product of M_a and the Büchi automaton B. A state machine $M \widehat{=}$*

$(S, S_0, F, \Sigma, T, I)$ *is a realization if it satisfies the following conditions:*

A1 $\forall (s_1, s_1') : M.S;\ s_2 : M_a.S;\ e : M_a.\Sigma \bullet (s_1, e, s_2) \in M_a.T \Rightarrow$
$\exists (s_3, s_3') : M.S;\ e' : M_a.\Sigma \bullet ((s_1, s_1'), e', (s_3, s_3')) \in M.T$

A2 $\forall (s_1, s_1'), (s_2, s_2') : M.S;\ s_3' : M_a.S;\ e : M_a.\Sigma \bullet$
$((s_1, s_1'), e, (s_2, s_2')) \in M.T \wedge (s_1, e, s_3) \in M_a.T \wedge$
e *is uncontrollable at* $s_1 \Rightarrow (\exists\, s_3 : M_a.S \bullet (s_1, s_1'), e, (s_3, s_3') \in M.T)$

Informally, **A1** states if a state is not a deadlock state, it shall not be a deadlock state in the realization. The operations enabled at a state are partitioned into two sets, controllable operations and uncontrollable operations. An operation is uncontrollable at a state if its postcondition depends on environmental inputs. For example, the $Join$ operation at the initial state of the state machine in Figure 6.3 is uncontrollable. An operation at a state is controllable if it is not uncontrollable. We remark that an operation may be controllable at a state but uncontrollable at another. This is different from works on supervisory control [123] as events in our context are complicated computations. **A2** states if a target state is reachable from a source state by applying an uncontrollable operation, the target state shall be reachable too in the realization.

In the following, we present the pruning algorithm that prunes states and transitions from the product recursively so as to construct a minimally restrictive (if possible) finite state design. For every reachable state s, we shall check if it satisfies requirement **A2**. If it does not, i.e., there is an uncontrollable action e at s whose post-states have been partially removed, all transitions from s labeled with e are pruned at once. Intuitively, an uncontrollable operation shall be forbidden at a state if allowing it may result in violation of the history invariant (given certain environment inputs). If a state is a fresh deadlock state, the state is pruned along with all its incoming and outgoing transitions. Pruning transitions may create new deadlock states. A state may violate **A2** after some of its immediate successor states get pruned since some of its outgoing transitions are pruned too. Therefore, the pruning must be applied recursively.

The algorithm is presented in Figure 6.4. Lines 1 to 4 declares 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. During pruning, variable $Path$ shall contain the states in the path from an initial state to the current state (inclusive). It is empty initially. State machine $Product$, Raw represents the

```
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 (!A2(Product, Raw, e, s))
10.                  prune all transitions labeled with e from Product;
11.              endif
12.          endfor
13.          if (!A1(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
}
```

Figure 6.4: Pruning algorithm

product and the abstract finite state machine 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 in $Successor$ but not in $Path$ or $Done$ is chosen at line 4. At line 7, we have another loop. The intuition is that the state shall be checked repeatedly until none of its decedent states is pruned. Lines $8 \,..\, 12$ verifies if the state satisfies **A2**. The function $A2(Product, Raw, e, s)$ returns true if all possible environment inputs to operation e at s is handled properly. Lines $13 \,..\, 16$ states that if the state is a fresh deadlock state, then the pruning backtracks by returning false, i.e., the parent state shall be checked again because one of its child states has been pruned. If the state satisfies both **A1** and **A2**, 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 re-examined. Otherwise, all decedent states have been pruned successfully and thus we are done with the state. Line 6 in procedure `prune` checks if there is a design after removing unreachable states and states leading to no accepting state from the pruned state machine. 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.

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 it are not pruned. The algorithm converges because the states and transitions are finite and it backtracks only when a state is pruned. We may further improve the efficiency making use of the fact that if a state satisfies both **A1** and **A2** and all states reachable from it do too, then it will never be pruned.

Example 6.3.2 (Pruning) If we specify the history invariant for $Queue$ as $\Box(\#items = 0)$, the product of the raw state machine and the Büchi automaton is the finite state machine in Example 6.3.1. After pruning, there is no initial or accepting state left (the transition is pruned because of violation of **A2** and the state is pruned because of violation of **A1**). Therefore, we conclude that there is no realization for such a specification. **end**

6.3.2 Calculating Guard Condition

The last step is to calculate a proper guard condition for each transition. A guarded transition can be applied only when its guard condition is satisfied. A guard condition guarantees that an operation is applied only when its precondition is satisfied. Moreover, part of a nondeterministic choice may get pruned in the pruning process. The remaining transitions are, therefore, constrained by restricting its postcondition. This is not directly implementable. Thus, a state guard is use to make sure that a transition is applied only when it will reach the desired postcondition.

Let $\mathcal{WP}$ be the weakest precondition operator introduced in [40]. Given an operation $Operation$ and a source state s_a and a state s_b that can be reached from s_a by applying $Operation$, the weakest precondition is defined as (a similar problem on the weakest precondition semantics of Z has been addressed in [25]):

$$\begin{aligned}&\mathcal{WP}(Operation, s_a, s_b)\\ &\quad \mathrel{\widehat{=}} (\exists\, State';\ outputs \bullet Operation) \wedge (\forall\, State';\ outputs \bullet (Operation \wedge s_a) \Rightarrow s_b)\end{aligned}$$

The first part of the condition guarantees the termination of the operation. The second part guarantees the postcondition. Intuitively, if the weakest precondition is satisfied by the valuation of the state variables before applying the operation, then the desired post-state is guaranteed to be reached.

Example 6.3.3 (Weakest precondition) The guard condition for $Join$ operation at the initial state of the state machine in Figure 6.3 to remain at the same state is:

$$\begin{aligned}&\mathcal{WP}(Join, \#items = 0, \#items = 0)\\ &\quad \mathrel{\widehat{=}} (\exists\, items' : \mathsf{seq}\, Package \bullet (expires(item?) \Rightarrow items' = items) \wedge\\ &\qquad (\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle)) \wedge\\ &\quad (\forall\, items' : \mathsf{seq}\, Package \bullet (\#items = 0 \wedge (expires(item?) \Rightarrow items' = items) \wedge\\ &\qquad (\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle)) \Rightarrow \#items' = 0)\\ &\quad \mathrel{\widehat{=}} \forall\, items' : \mathsf{seq}\, Package \bullet ((expires(item?) \wedge items' = \langle\, \rangle) \vee\\ &\qquad (\neg\, expires(item?) \wedge items' = \langle item? \rangle)) \Rightarrow \#items' = 0\\ &\quad \mathrel{\widehat{=}} \forall\, items' : \mathsf{seq}\, Package \bullet \#items' = 0 \vee expires(item?) \vee items' = \langle item? \rangle\\ &\quad \mathrel{\widehat{=}} expires(item?)\end{aligned}$$

The first deduction is due to the definition of weakest precondition and the second is due to the one point rule. Thus, the transition is guarded with $expires(item?)$. **end**

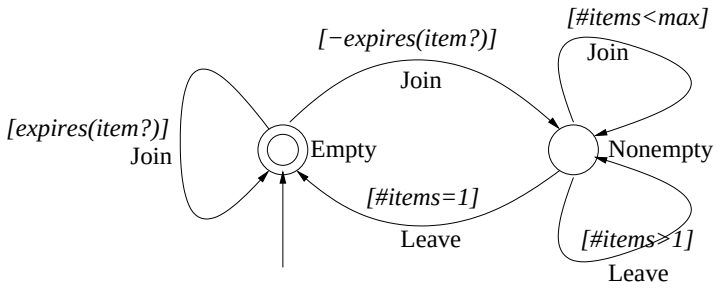


Figure 6.5: Realization of $FairBoundedQueue$

If the weakest precondition turns out to be $false$, it means that there is no way that we can guarantee that the transition ends up with the desired state. This is normally due to internal nondeterminism, i.e., some information is not present at the abstract level. Such transitions are pruned. The pruned state machine with guard conditions for $FairBoundedQueue$ is in Figure 6.5. States are labeled with names to improve readability.

Example 6.3.4 (Composed Object-Z class)

$$
\begin{array}{l}
\textit{Multiplexer} \\
\hline
\upharpoonright(\textsc{Init}, Join_1, Join_2, Transfer_1, Transfer_2, Leave) \\
\quad \begin{array}{l} input_1, input_2 : FairBoundedQueue \\ output : Queue \end{array} \\
\quad \textsc{Init} \\
\quad\quad input_1.\textsc{Init} \wedge input_2.\textsc{Init} \wedge output.\textsc{Init} \\
Join_1 \mathrel{\widehat{=}} input_1.Join \\
Join_2 \mathrel{\widehat{=}} input_2.Join \\
Transfer_1 \mathrel{\widehat{=}} input_1.Leave \parallel output.Join \\
Transfer_2 \mathrel{\widehat{=}} input_2.Leave \parallel output.Join \\
Leave \mathrel{\widehat{=}} output.Leave \\
\hline
\Box \# output.items \leq output.max
\end{array}
$$

We use a multiplexer example to show how our method works for composed classes. A multiplexer is made up of three bounded queues, two as incoming channels and one as an outgoing channel.

It can be viewed as a network router which gets packages from two different sources and forwards those which have not expired yet. All packages in the incoming channels are eventually forwarded to the outgoing channel. The history invariants include those inherited from the $FairBoundedQueue$. The predicates for abstraction include those in the history invariant and initial schema. They are:

$$\{\#input_1.items = 0, \#input_1.items \leq input_1.max \\ \#input_2.items = 0, \#input_2.items \leq input_2.max\}$$

Only operations defined or promoted in this class are concerned. For operations composed using operation operators, the process of calculating preconditions and postconditions can be simplified by considering the structure of an operation (refer to chapter 14 in [161]). We remark that an uncontrollable operation may become controllable when the object composes with other objects. For example, operation $output.Join$ is initially uncontrollable (at all states) when we consider $Queue$ class along. It becomes controllable as in operation $Transfer$ because all packages from either of the incoming channels are not expired. The final finite state machine realized from $Multiplexer$ is presented in Figure 6.6. **end**

6.4 Discussion

This section is devoted to a discussion on remaining issues on the approach, for instance the soundness, a prototype implementation and a practical implication of the approach.

6.4.1 Soundness

A state machine is a realization of an Object-Z specification if and only if it satisfies the following condition: all operations are applied when its precondition and postcondition are satisfied (A_3), all possible sequence of operations satisfies the history invariant (A_4), A_1 and A_2. A_3 is guaranteed by guarding each transition with a condition stronger than its precondition (the weakest precondition). In the process of pruning the product, all fresh deadlock states, and states and transitions violating A_2 are pruned. It is straightforward to verify that both A_1 and A_2 are satisfied. To prove A_4,

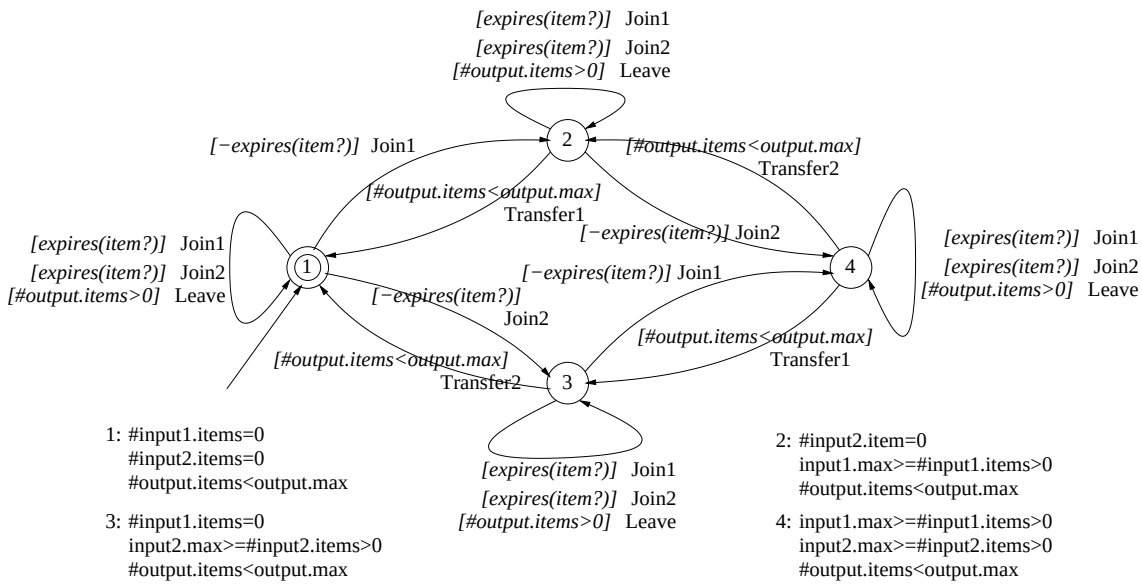


Figure 6.6: Realization of $Multiplexer$

we show that there is a fair simulation relation from our realization to the product of the state transition system defined by an Object-Z specification and the Büchi automaton representing its history invariant (the specification). The notion of fair trace-containment is robust with respect to LTL. Therefore, we may conclude that A_4 is satisfied.

Theorem 6.4.1 *Let M_c be the product of the (concrete) transition system determined by the Object-Z specification and the Büchi automaton representing the history invariant. Let M_r be a realization constructed using our method. M_c fairly simulates M_r.*

Proof. *We claim that the following total relation is a fair simulation relation from M_r to M_c.*

$$\mathcal{R} \mathrel{\widehat{=}} \{(r, c) : M_r.S \times M_c.S \mid c \text{ is a state where } M_r.\mathcal{I}(r) \text{ is true}\}$$

C1 *is an immediate consequence of the fact that the initial condition is included in the predicates for abstraction. In the abstraction process, an abstract state is identified as an initial state if and only if the initial condition is satisfied. Because the weakest condition calculated in the last step is stronger than the precondition, an operation is applied only when its predication is satisfied. Engaging in an*

operation 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 specifications. Though there may be infeasible pathes in the synthesized implementation, an operation may reach a successor state only if the postcondition is satisfied at the successor state, i.e., there is a corresponding transition in M_c. Thus, ***C2*** *is true. A state in M_r is a final state if it satisfies the fair constraint. All simulating states of the state satisfies the fair constraint (definition of $\mathcal{R}$). Thus,* ***C3*** *is true. We conclude that M_c fairly simulates M_r.* □

6.4.2 Automation

Our method is automated by experimental tool in JAVA. The inputs are an Object-Z class specification in its XML representation [150], along with an optional set of predicates for abstraction. By default, the predicates include those in the history invariant and the INIT schema. The predicate abstraction is automated with the help of PVS [117]. Lemmas are generated automatically from the Object-Z specification for calculating the abstract INIT schema, precondition and postcondition of each operation. In general, the number of lemmas is exponential to the number of the predicates. A number of tricks are used to reduce the abstract state space, e.g., removing false states by considering co-relation between the predicates. PVS is invoked in batch mode to prove the lemmas automatically without user interaction. We believe that it is unlikely that a user would like to prove the lemmas interactively for complex systems. To further speed up the abstraction so as to handle complex systems, a more loop-free proving strategy than $grind$ (the highest-level command in PVS) is used to prove each lemma in a limited amount of time.

PVS is used to automatically compute an approximation of the $\mathcal{S}$ function. Given a predicate p, we generate a PVS lemma for each abstract state to check if the predicates labeled with the state implies $\neg\, p$. A file containing all lemmas (in standard PVS syntax) is generated by our tool. Users may interactively prove the lemmas or more rationally let PVS do automatic proving. Either way, the proof result is written to a log file (in the latter case, the log file is named *orphaned-proofs.prf* by default). The log file is then processed to construct the abstract model. $\mathcal{S}(p)$ is computed as the set of abstract states where $\neg\, p$ is not implied true (the corresponding lemma is unsuccessfully

proved). A lemma is not proved either because it is not true or PVS is not powerful enough to prove it. Therefore, we compute an approximation of $\mathcal{S}(p)$. Given p as the postcondition of an operation, $\mathcal{S}(p)$ identifies the set of abstract states that may be reached by applying the operation.

A raw state machine is constructed from the proving result. It is then composed with the Büchi automata generated from Wring [140]. The product is pruned using our pruning algorithm. If there is at least one initial state and at least one reachable final state left, the pruned state machine is equipped with guard conditions and presented to users as a realization. However, computing the weakest precondition involves eliminating dashed variables. Variable elimination in our context is in general undecidable. Yet an interesting enough subset is decidable where there is no nonlinear integer arithmetic and no shielded variables occurring inside uninterpreted terms. PVS is currently lacking such a procedure. However, we can always use PVS to prove-check a manually constructed stronger guard.

More features on connecting our tool to existing tools for state machine like structures will be offered. For example, we plan to generate an XMI [130] representation of our state machines so that they can be exchanged and visualized using tools like Rational Rose [124]. We may also generate codes for Rhapsody [67] so that we may simulate the model and synthesize working code from the Object-Z specification if the implementation of each operation is supplied (and tested by checking the precondition and postcondition) by the user.

6.4.3 Event-based Controllers for State-based Plants

A good principle for modeling complex control systems is to separate system functionalities from control aspects in the early system design stage. For instance, a system engineer may typically identify the set of objects/classes in a system, data variables and operations in each class before experimenting with control flows to ensure critical system properties. Such early stage functional designs are typically documented as UML class diagrams [154] or mathematical models like Z/Object-Z [161, 50], B [1]. In later stages, event-based formalisms, e.g., CSP [79] and Pi-calculus [132], can be used to specify complicated control flows, i.e., order of applying the operations, conditions to

guard the invocation of operations, etc. Designing such control flows is time consuming and error-prone. Given a system functional state-based design and abstract important system properties, can event-based controllers be automatically generated so that the controller can orchestrate the system functions to satisfy the properties?

A related problem known as controller synthesis has been studied for many years [27, 23]. The problem of synthesizing controllers is of finding a controller that restricts the behavior of a given process in order to satisfy given constraints on sequences of actions executed by the process. A rich set of theories has been developed [121, 122, 99]. We believe that mature development on controller synthesis can be applied to automatically synthesize control flows based on a precisely defined system functional model. Such approach is beneficial because an event-based controller is implementable [133], contrary to state-based specifications. Moreover, a 'minimum' restrictive controller may be synthesized so that system engineers may further restrict it without violating the critical properties. For instance, data and functional requirements may be specified using Object-Z, as $plants$. We may then automatically synthesize prototype controllers in CSP to control the Object-Z specification. Without repeating the techniques, we illustrate the method using the vending machine example.

Example 6.4.2 (Object-Z plant) Figure 6.7 shows an Object-Z class modeling a typical vending machine. $Request$ and $Coins$ are user-defined primitive types representing possible user requests and acceptable coins to the machine. In the state schema, two state variables, $quota$ and req, are defined to record the amount inserted by a user and the user's current request. Irrelevant information like the location of the vending machine, total coins in the vending machine are abstracted away. Four operations are specified, namely $InsertCoin$, $ReleaseCoin$, $Request$ and $Dispatch$. Each operation is defined in terms of its effects on the state variables and inputs/outputs relations from/to the environment. A user may increase $quota$ by $InsertCoin$ or set req by $Request$. Or a user may ask the machine to $Dispatch$ an item or to $ReleaseCoin$. In the $InsertCoin$ schema, variable $coin?$ models the coins inserted. **end**

Our method is to reuse the abstraction schema to construct an abstract state machine from the

VendingMachine

$Request ::= Nil \mid Candy \mid Coke$

$Coins == \{10, 20, 50, 100\}$

$quota : \mathbb{Z}$

$req : Request$

INIT

$quota = 0 \land req = Nil$

InsertCoin

$\Delta(quota)$

$coin? : Coins$

$quota' = quota + coin?$

ReleaseCoin

$\Delta(quota, req)$

$quota > 0$

$quota' = 0 \land req' = Nil$

Request

$\Delta(req)$

$req? : Candy \mid Coke$

$req = Nil \land req' = req?$

Dispatch

$\Delta(req, quota)$

$req \neq Nil \land req' = Nil$

$(req = Candy \land quota' = quota - 50) \lor$

$\quad (req = Coke \land quota' = quota - 80)$

Figure 6.7: Object-Z specification of vending machine

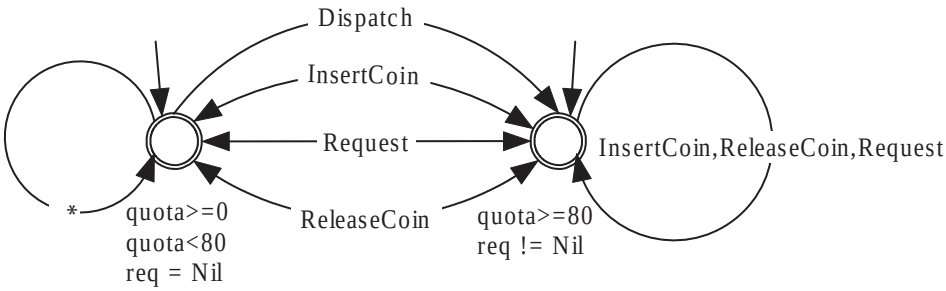


Figure 6.8: State machine specification

Object-Z model, compute the product of the abstract state machine and the abstract behavioral system requirements (which plays the same role as the history invariant), apply the pruning algorithm to construct a finite state controller, and lastly express the controller using CSP processes. In our approach, any $property$ that can be represented as finite state machine is acceptable. The simplest kind is an automaton or automata-like model (Kripke Structure, Finite State Machine with/without datapath). More importantly, temporal logic formulæ can be considered as $property$. In [26], Linear-time Temporal Logic is extended to refer to temporal properties of both state and event based on Labelled Kripke Structure, called State/Event LTL (SELTL). In our setting, temporal formulæ that concern both state and event information are allowed. For example, an invariant property concerning both state and event is $\Box(quota \geq 80 \wedge req \neq Nil \rightarrow Dispatch)$, which says whenever no less than 80 cents are inserted and the user has made a request, the vending machine dispatches. For simple invariant properties (specified as temporal logic formulæ using no negation and only universal quantification over computation sequences/trees), Büchi Automata with all states as accepting states can be constructed, which is treated as finite state machines.

Example 6.4.3 (Finite state machine specification) Figure 6.8 is a finite state machine specification. Two safety properties are captured. One is $\mathbf{G}(quota \geq 80 \wedge req \neq Nil \rightarrow Dispatch)$. The other is that the variable $quota$ shall always be non-negative so that negative profit for the vending machine is impossible. **end**

The same pruning algorithm is applied to decide whether there is a controller and if there is, synthesize one automatically. After pruning, we synthesize event-based controllers from the pruned finite state machine. Synthesizing CSP process expressions from the finite state machine is straightforward, e.g., [107]. An intuitive approach is to mimic the states, i.e., one process is defined for each node in the finite state machine. The main process is defined as a non-deterministic choice of the initial nodes.

Example 6.4.4 (Controller of the vending machine) The following is a CSP controller of the vending machine:

$$
\begin{aligned}
P_1 \quad &\widehat{=} ([quota + coin? < 80 \vee req = Nil] \bullet InsertCoin;\ P_1) \\
&\qquad \Box\ ([quota + coin? \geq 80 \wedge req \neq Nil] \bullet InsertCoin;\ P_2) \\
&\qquad \Box\ ([quota > 0] \bullet ReleaseCoin;\ P_1) \\
&\qquad \Box\ ([quota < 80] \bullet Request;\ P_1) \\
&\qquad \Box\ ([quota \geq 80] \bullet Request;\ P_2) \\
&\qquad \Box\ ([(req \neq Nil \wedge quota \geq 80) \vee (req = Candy \wedge quota \geq 50)] \bullet \\
&\qquad\qquad Dispatch;\ P_1) \\
P_2 \quad &\widehat{=} InsertCoin \Box ReleaseCoin \\
\text{MAIN} &\widehat{=} P_1
\end{aligned}
$$

The pruned state machine contains two states. The process capturing behaviors patterns at the initial state is written as P_1. All operations can be invoked at the state. The state guards guarantee that the state variables at the state satisfying the condition $quota \geq 0 \wedge \neg(quota \geq 80 \wedge req \neq Nil)$. The behavior patterns at the other state are captured in process P_2. The MAIN process is identified with the process for the initial state, in particular P_1. We remark that the CSP controller composed with the Object-Z specification constitutes to a TCOZ specification. **end**

6.5 Summary

The contribution of the work is twofold. Firstly, we developed a systematic method to abstract an Object-Z specification on a class base. Such a method is useful for verification of Object-Z specifications as well. Secondly, we developed an effective way of realizing an Object-Z specification as finite state machines, i.e., constructing a control program to guide the execution of the Object-Z

specification. By treating each transition as a function call and implementing each operation in isolation, we may generate executable codes from the specification. Moreover, an experimental tool is developed to realize the method.

A less restrictive controller would allow more possible further refinement. Our method works by pruning those sequences of operations that fail the specification or the additional requirements. Therefore, it is naturally 'minimally' restrictive. However, a minimum restrictive controller in general may not exist. An example can be found in [98].

Our work is related to works on abstraction and controller synthesis. Abstraction techniques are now widely considered useful and even necessary for successful verification. It has been discussed in various works on model-checking software, e.g., Graf's work on property preserving abstractions for transition systems [97] and Ball's work on abstraction of C programs[7]. Though partially inspired by Graf's work, our abstraction schema is highly coupled with Object-Z semantics. The abstraction schema is closely related to the work in [139], where Smith and Winter proposed a similar predicate abstraction for totalized Z specifications. Their aim is to verify safety temporal properties of Z specification. The difference between their abstraction and ours is that our predicate abstraction applies to Object-Z specifications (therefore, we do not assume operations to be totalized) and, more importantly, is automated by PVS. The latter is essential for complex systems.

Our work is also related to works on deriving an automata representation from Z/Object-Z for specification-based testing [38, 113, 77, 116]. Dick and Faivre in [38] derived an automata representation from a Z specification for generating test cases. Murray in [113] formally derived a finite state machine of an Object-Z specification for the same purpose. Their works focus on extracting a finite set of behaviors for testing (partial coverage). Our work focuses on extracting implementable finite state models from Object-Z specification. By contrast, we guarantee all behaviors are properly constrained.

The part of work on deriving a finite state representation from the data object is related to early works on using processes to represent data structures by Nierstrasz and Brinksma in [19, 115]. Our work is also inspired by works on controller synthesis both from computer science and control-

theoretic perspective. The line of work goes back to the realization problem [27] formulated by Church and later solved by Büchi and Landweber [23]. During the past decade, there has been a vigorous revival of this area. Various problems associated with partial observability, controllability and hierarchical control have been addressed as evidenced in [121, 122, 99]. However, previous works on controller synthesis are all based on automata-like structures with trivial data states. Our work applies to applications with complicated data and functional requirements.

Chapter 7

From Scenarios with Data to Implementations

'Do you mean that you think you can find out the answer to it?'
said the March Hare.
'Exactly so,' said Alice.
'Then you should say what you mean,'
the March Hare went on.
- *Alice's Adventures in Wonderland*, Lewis Carroll

Behavior modeling plays an important role in software engineering. It is the basis of system development methods like system specification, design, code generation, testing and verification. Two complementary approaches for modeling behavior have been shown to be useful in practice. One is interaction-based, which focuses on global interactions between system components, e.g., MSC, LSC. The other is state-based modeling, which concentrates on the internal states of individual components, e.g., Z and VDM [83]. In Chapter 5, we investigated ways of generating distributed processes from interaction-based modeling, namely LSC. In Chapter 6, we addressed the problem of synthesizing implementable designs from state-based modeling, namely Object-Z. Industrial scale

systems often have not only complex data structure but also intensive interactive behaviors. In this chapter, a combination of the two approaches is proposed so that we may synthesize implementations all the way from LSC models equipped with complex data structures.

7.1 Introduction

In order to formally specify complex systems, we propose a combination of interaction and state-based modeling, namely Live Sequence Chart and Z specification. That is, a complete system specification shall consist 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 of the combination is that it combines the modeling power of both and thus can be used to specify systems beyond the capability of either one. Moreover, such combined specifications contain sufficient information for synthesis of distributed implementable system design.

State-based modeling naturally complements interaction-based modeling, and thus it is no doubt that a smooth integration of them shall be beneficial. LSC is a rather rich extension to MSC that allows specification of not only possible behaviors, but also mandatory behaviors. We choose Z over other state-based modeling language because Z is widely known and accepted as well as well-developed in terms of specification, refinement, etc. The Z language is favored over Object-Z because Z is relatively simply structured and the class structure (as well as inheritance and polymorphism) in Object-Z may serve as an unnecessary complication.

Synthesis from specifications like scenario-based diagrams or various automata is showed to be extremely hard [121, 122, 68, 88]. Our problem is further complicated by the complex data structure underlying the scenarios. Due to the high complexity of the problem, our primary aim is to discover a practical way of synthesizing sound (and not necessarily complete) implementations. To the best of our knowledge, our work is the first attempt to synthesize low-level implementations from a combination of interactive-based modeling and state-based modeling.

We take a step-by-step approach. Firstly, a distributed object system is synthesized from the LSC

universal charts. The local actions in the charts are treated as abstract events, as we did in Chapter 5. The global state machine is never constructed during the steps so as to avoid state space explosion. Meanwhile, an abstract finite state machine is constructed from the Z model using automated predicate abstraction [7, 97], which allows us to grasp the behaviors of the objects based on a finite set of assertions. The abstraction method presented in Chapter 6 is augmented to cope with Z semantics. Secondly, the distributed object system is refined on an object basis to satisfy data-related requirements. Thus, the preconditions of the local actions (Z operations) and hot conditions in the LSC model will never 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 implementations by generating code from the refined finite state machine (the design). Our method is implemented as a JAVA application.

7.2 Integrating Live Sequence Chart and Z

State-based modeling language like Z and interaction-based modeling languages like LSC naturally complement each other. LSC lacks the expressiveness to capture complicated data and functional behaviors. Local actions are often ignored or treated as abstract events in the study of the verification and synthesis problem of LSC. Examples are the works in [68, 18] and our work in Chapter 5. Local data variables are often implicitly associated with the objects. They may appear in the conditions or get updated by the local actions. However, there is no way to specify exactly how the local actions update the local variables and what the data space of the object is, except using concrete implementations, which we think is undesirable as sequence diagrams are used in the early stage 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 representation, it is difficult to analyze or implement those abstract models. Z is not intended for timed or concurrent behaviors [161]. It lacks the expressiveness to capture dynamic interactive behaviors between the components in the system.

A combination of LSC and Z shall constitutes a powerful modeling language covering a wider

range of systems. Thus, we propose a simple yet effective integration of LSC and Z. We require that a combined system specification shall 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. In particular, each object in the LSC model with non-trivial 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 may only mention variables defined in the respective Z state schema in addition to external inputs.

System modeling shall start with identifying scenario-based system requirements, from which the universal charts are constructed. During the process, the system engineer slowly decides the data variable and local actions for each object. The designer's intension of the local action can be naturally documented as pre/postcondition pairs. Later, the designer may specify each local action using Z operation schema to formally state how each local action updates the data state. This way, a complete system specification is built. In the following, the same Light Control System is used as a running example to show how it may be specified using a combination of LSC and Z packages, and how an implementation may be synthesized from the specification.

Example 7.2.1 (Universal charts of Light Control System) Figure 7.1 captures a typical scenario 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 adjusting the light with appropriate illumination if the light is already on. Figure 7.2 illustrates another scenario of the LCS. 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. The forbidden events require 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. The rest of the scenarios are presented in Figure 7.3, in which the occupant may directly turn on/off the light by pushing the button or the system may adjust the illumination of the light. **end**

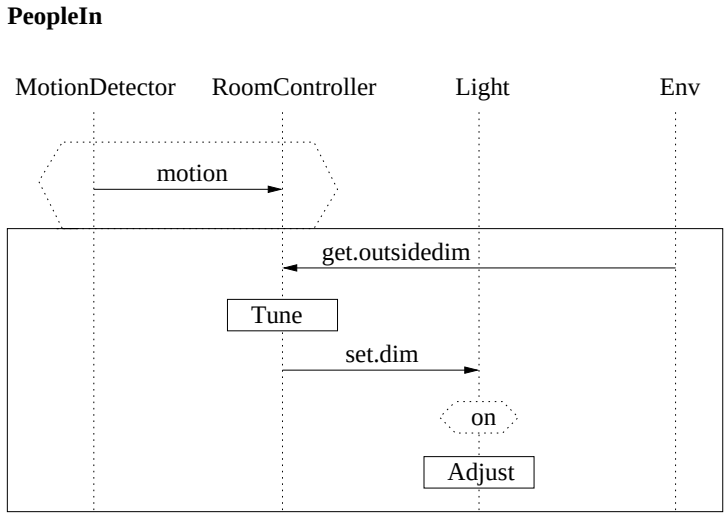


Figure 7.1: Scenario of the LCS: *PeopleIn*

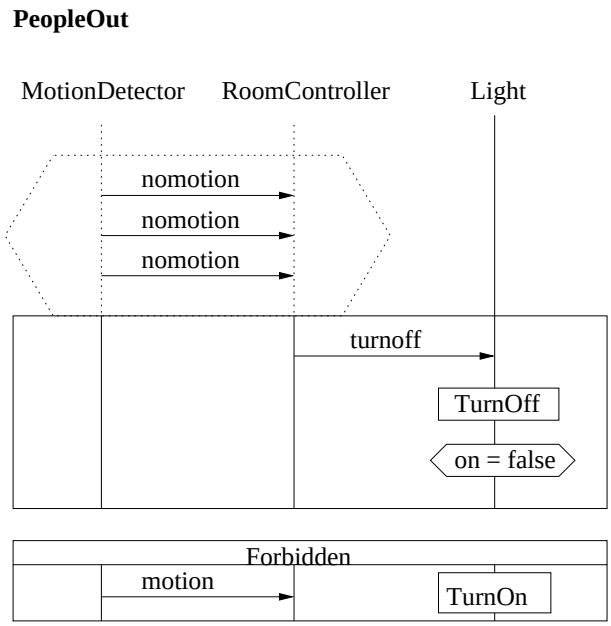


Figure 7.2: Scenario of the LCS: *PeopleOut*

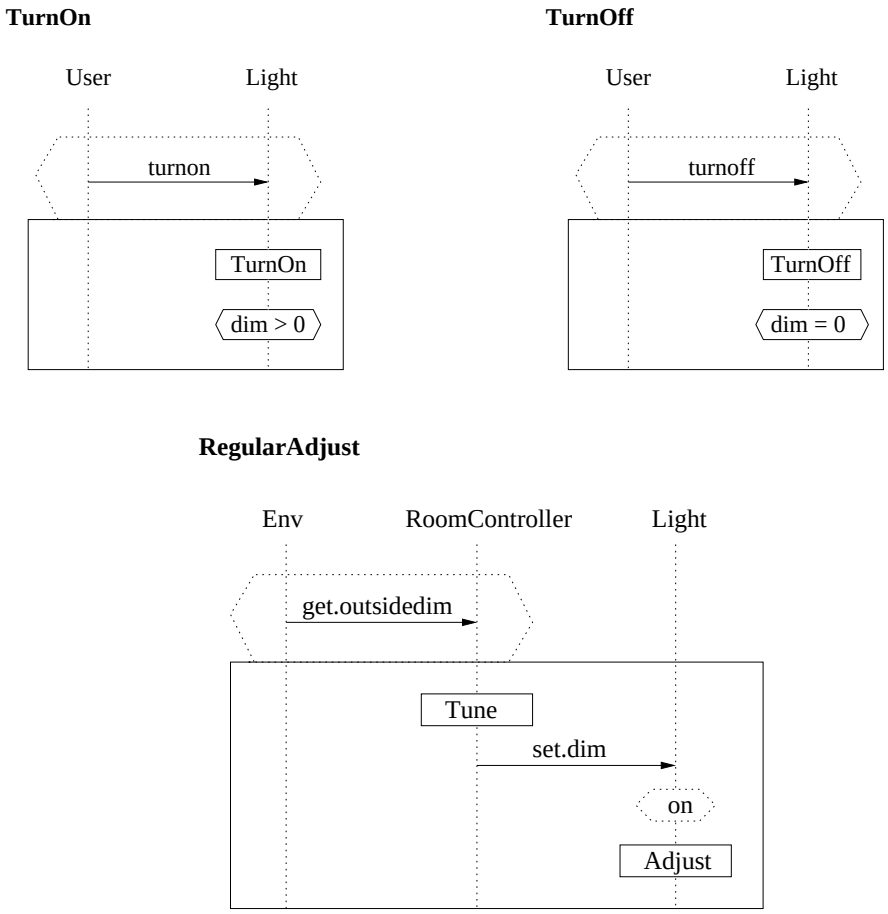


Figure 7.3: Scenarios of the LCS

After identifying the universal charts, the data variables and local computation of each object become clear. No local action is associated with instance $MotionDetector$, which suggests that it has trivial data state. The Z package associated with the $Light$ and $RoomController$ are illustrated in the following.

Example 7.2.2 (Z package of $Light$) The Z package of the *light* contains the following schemas.

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

$$\begin{array}{|l}\hline \textit{Adjust} \\ \Delta \mathit{Light} \\ dim? : \mathit{Illumination} \\ \hline on = \mathit{true} \land dim' = dim? \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{TurningOn} \\ \Delta \mathit{Light} \\ \hline on = \mathit{false} \land dim' = 100 \land on' = \mathit{true} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{TurningOff} \\ \Delta \mathit{Light} \\ \hline on = \mathit{true} \land dim' = 0 \land on' = \mathit{false} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{LightInit} \\ \mathit{Light}' \\ \hline dim' = 0 \land on' = \mathit{false} \\ \hline\end{array}$$

end

Example 7.2.3 (Z package of $RoomController$) The Z package of the *room controller* contains the following schemas.

$$\begin{array}{|l}\hline \textit{RoomController} \\ dim : 0 \mathinner{..} 100 \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{RoomControllerInit} \\ \mathit{RoomController}' \\ \hline dim' = 0 \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Tune} \\ \Delta \mathit{RoomController} \\ outsidedim? : 0 \mathinner{..} 100 \\ \hline (outsidedim? \leq 20 \land dim' = 100) \lor \\ \qquad (outsidedim? > 20 \land dim' + outsidedim? = 100) \\ \hline\end{array}$$

The variable dim in the state schema represents the light level (in *room controller*'s knowledge). Initially, it is of value 0. The operation $Tune$ computes the desired light level according to the outside light level. **end**

Example 7.2.4 (Combined specification of LCS) All instances in Figure 7.1,7.2,7.3 with non-trivial 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. **end**

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 and methods. Once both parts stabilize, the integrated specifications shall contain sufficient information on both data and control aspects of the system, which allows us to automatically synthesize implementable designs. Graphically, links from an instance in the chart to its Z state schema, and links from local actions to Z operation schemas shall be provided, e.g., the Z schema is shown in the popup window once the instance is highlighted and so are the operation schemas.

7.3 Synthesis of Distributed Object System

In this section, a distributed object system is synthesized from the universal charts. 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 synthesis is closely related to the construction in Chapter 5. However, because we have to store the data-related requirements for later refinement, finite state machines instead of CSP processes are constructed. State invariants are used to store data requirements. Moreover, using finite state machines allows us to reuse our work in Chapter 6.

There are a number of principles to identify a good synthesis strategy. Firstly, the synthesis should be robust with the notion of data refinement [161, 111] so that the synthesized design remains valid after refinement of the Z operations. Secondly, the global state machine should not need to be constructed in order to avoid the state explosion problem. This is essential for notations like LSC, which has a distributed nature and an underlying partial order semantics. Above all, the synthesized design should be consistent with the specification.

7.3.1 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 sub-chart of a main chart without hierarchy), let $M_m^i \widehat{=} (S, S_0, F, \Sigma, T, I)$ be a state machine synthesized from instance i in chart m. The basic idea is to construct one state for each location. Thus, S is the set of states corresponding to the set of 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 each location labeled with a cold condition, an additional state labeled with the negation of the condition is constructed so that if the condition is violated, the additional state is reached. The only transition enabled at the additional state is labeled with a synchronization barrier, which is used to terminate the (activation of) the chart. For each location labeled with a hot condition, the condition is labeled with the respective state and no additional state is added. This prevents behaviors that might violate 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 which 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 over the system. Such a state machine constrains a single activation of the basic chart.

A hierarchical chart can be flattened as finite state machines straightforwardly. Figure 7.4 presents a universal chart containing a conditional branch. It is part of the LSC specification of a lift control system. Whenever the lift approaches the next floor, the $shaft$ sends a message $arriving$ to $controller$. The $controller$ refreshes its knowledge of the current level by updating its local variable pos. A hot condition stating that the value of pos (a local variable representing the current level) 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.

Example 7.3.1 (State machine for $Shaft$) The state machine presented in Figure 7.5 captures the behaviors of $Shaft$ in the main. Events $Arrive.x.main$ and $Arrive.x.sub1$ are barriers used to syn-

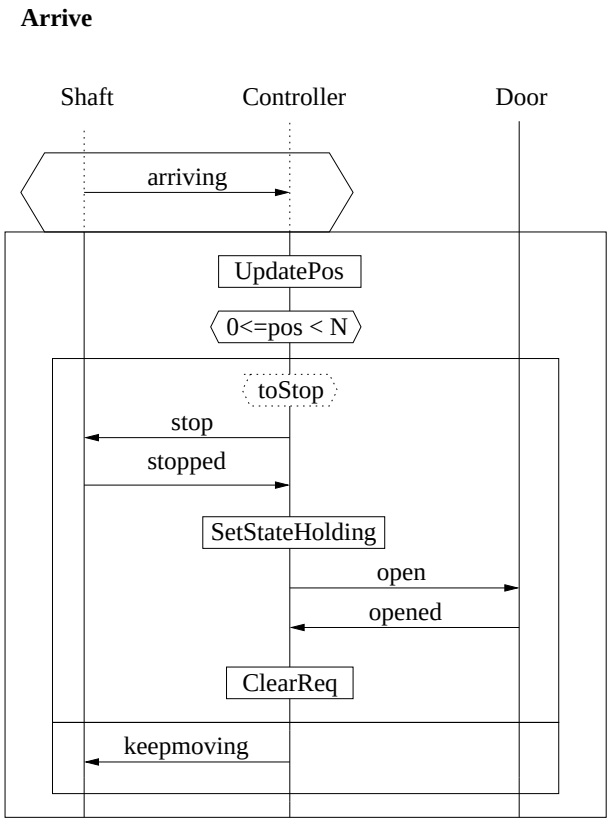


Figure 7.4: Scenario of Lift Control System

chronize the entering or exiting of the main chart or a sub-chart among all participating instances. Variable x is an identifier which distinguishes different activations of the same chart. Therefore, only participating instances in the same activation of the chart are synchronized. Whenever the chart completes (reaching the filled circle), all events in Σ_m^i can be engaged freely (indicated by a transition labeled with *). Only transitions labeled with visible events are constructed since transitions concerning invisible events are free to occur by the definition of parallel composition. **end**

Example 7.3.2 (State machine for $Controller$**)** The state machine presented in Figure 7.6 is synthesized for instance $Controller$. The hot condition is labeled with the state right after local action $UpdatePos$. After entering the sub-chart, two states are reached, one labeled with condition $toStop$

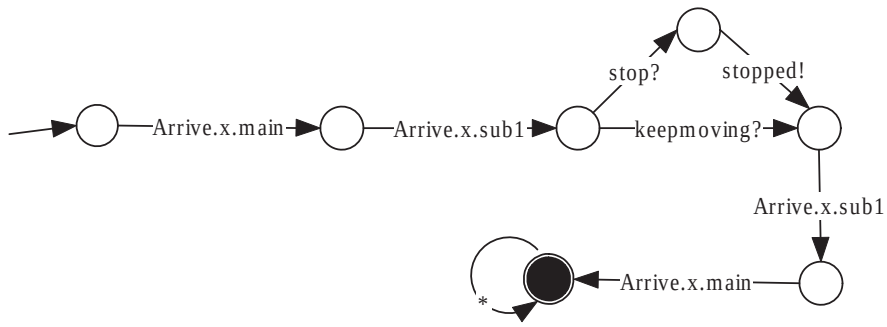


Figure 7.5: State machine for *Shaft*

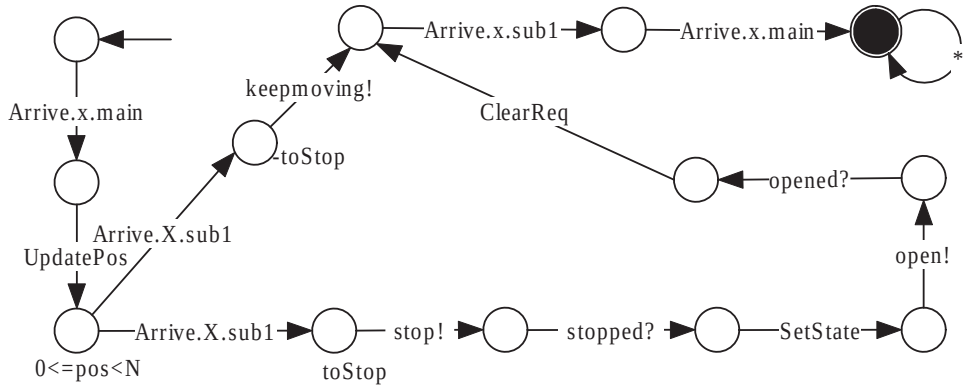


Figure 7.6: State machine for *Controller*

and the other labeled with its negation. Thus, the conditional branch is effectively flattened. In general, state machines for hierarchical charts can be constructed from the state machines for the sub-charts. **end**

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 sub-chart. The x component is necessary because there could be multiple or even infinite overlapping activations of the same chart. For instance, trace $\langle nomotion, nomotion, nomotion \rangle$ triggers

three overlapping activations of the chart *PeopleOut*. Event $u.x.y$ is used to synchronize the entering or exiting of sub-chart y in chart u among those participating instances. Event $u.x.y.conVio$ is engaged if and only if a cold condition in sub-chart y is violated in the x-activation of u. It is the only event which can be engaged at the state labeled with the negation of a cold condition. Other instances in the chart are ready to engage in this event all the time (a transition labeled with this transition is enabled at every state in the state machine for other instances).

The state machine for an instance in the pre-chart is similarly constructed. However, because a universal chart puts no constraint over the system before entering the main chart, the state machine synthesized from the pre-chart shall allow all possible behaviors, and at the same time monitor communication sequences that may match the pre-chart. Let $M_p^i \widehat{=} (S, S_0, F, \Sigma, T, I)$ be the state machine synthesized from instance i in the pre-chart 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, which is labeled with e. In addition, a transition (s_1, e', s_{max}) is constructed for every event e' in $\Sigma_u^i \setminus \{e\}$, where s_{max} is the state corresponding to the last location on instance i in the *main* chart (the filled one). Intuitively, the pre-chart progresses whenever an expected event is engaged, whereas an unexpected event aborts the activation of the chart. Because hot condition in pre-chart has no semantic meaning, all conditions in pre-charts are treated as cold conditions. Lastly, the state corresponding to the last location in the pre-chart is identified with the state corresponding to the first location in the main chart so that once the pre-chart is completed, the main chart is reached.

Example 7.3.3 (State machines for instances in PeopleOut) Figure 7.7 shows the state machines synthesized for instances in the chart showed in Figure 7.2. The alphabet of each state machine includes the forbidden events. The forbidden events are allowed to occur before entering the main chart. Once a communication sequence matches the pre-chart, the state machine synchronizes entering of the main chart. All states in the pre-chart are accepting as the state machine shall not constrain the system execution before entering the main chart. **end**

The state machines constructed so far only monitor a single activation of the chart. A trace which triggers multiple activations of the same chart is not properly constrained. For instance, the state

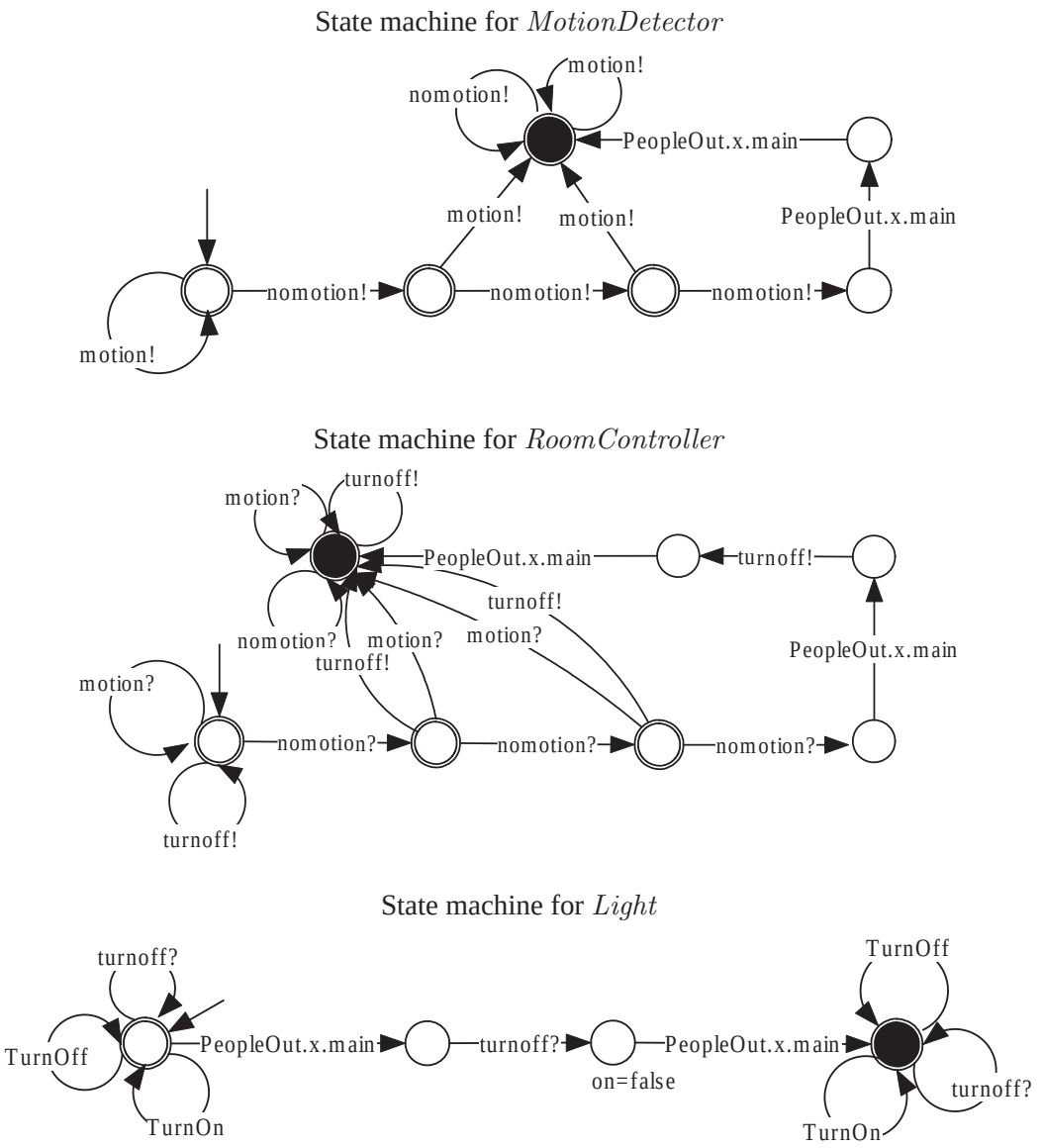


Figure 7.7: State machines for instances in *PeopleOut*

machines in Figure 7.7 may execute the following trace:

$$\langle nomotion, motion, nomotion, nomotion, nomotion, TurnOff \rangle$$

It is however not allowed by the chart in Figure 7.2 because the three consecutive $nomotion?$ triggers another activation of the chart. The remedy is to identify the filled state with the initial state so that the state machine is reused for later activations. However, such state machines still can not constrain overlapping activations. 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 [17], 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 non-overlapping activations. In practice, large number of overlapping activations is unlikely because system behaviors are increasingly restricted as the number of overlapping activations increases. There is often a natural limit on the number of overlapping activations. For instance, there could be at most three overlapping activation of chart *PeopleOut* because the main chart shall complete before the fourth $nomotion$ event. A simple analysis shall tell the maximum number of activations allowed by a chart.

Example 7.3.4 (Final state machine) The state machine presented in Figure 7.8 is synthesized from $RoomController$ in scenario *PeopleOut*. It monitors the x-activation of the chart. The state machine is augmented with a special synchronization barrier $fork.x$, which is used internally to activate a new copy of the state machine whenever it moves beyond the initial state. Because 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. 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 are computed as showed in Figure 7.9. The very last state (the one composed by three filled state) is identified with the initial state so as to allow non-overlapping activation. We remark that the final state machine can be further reduced using standard techniques

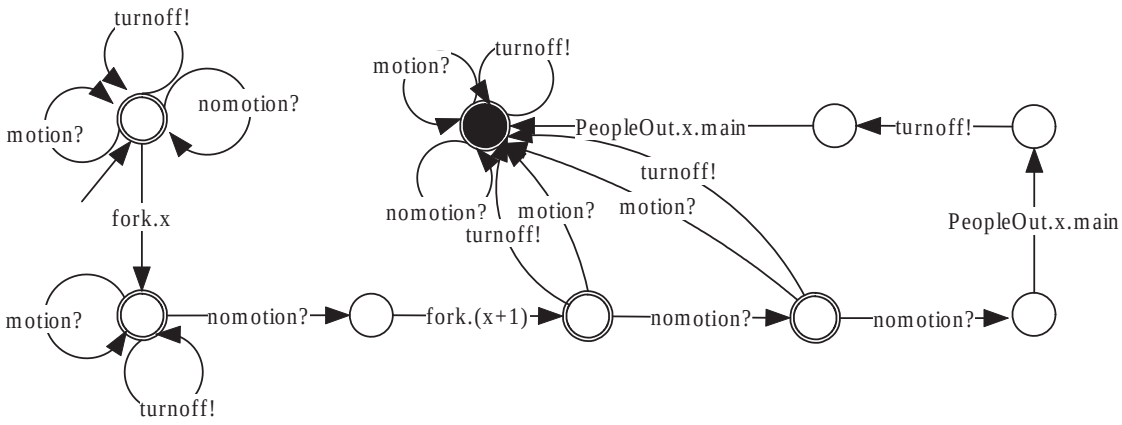


Figure 7.8: State machines for instance $RoomController$

like bi-simulation reduction [56], etc. For instance, all states labeled with event $fork$ are removed since they contribute nothing to system behaviors. **end**

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 rule is established. Let M_{LSC}^i be the local behaviors of an object i.

$$\|_u \|_i M_u^i \mathrel{\widehat{=}} \|_i \|_u M_u^i \mathrel{\widehat{=}} \|_i M_{LSC}^i$$

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. In previous chapters, we have formally defined a trace-based denotational semantics for LSC, and then developed a sound interpretation of LSC in the classic notion of CSP [79]. By transforming CSP interpretations of the LSC model using its algebraic laws, the local behaviors of each object are grouped together as a set of distributed processes. A bisimulation relation between

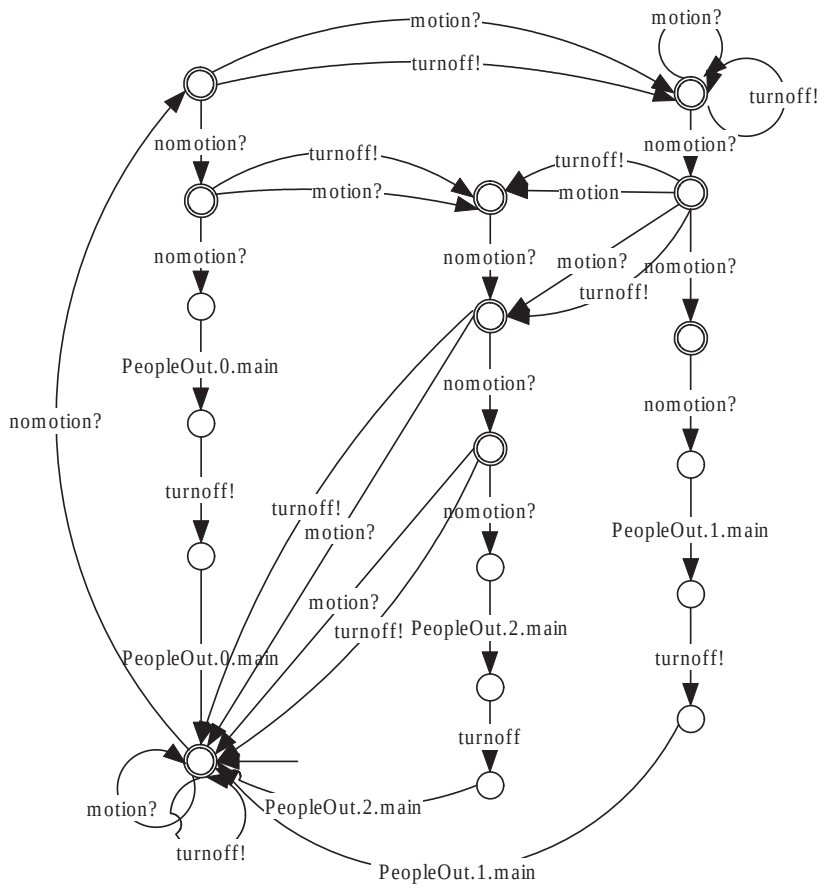


Figure 7.9: State machine synthesized for instance $RoomController$

the synthesized state machine and the transition system interpretation of the distributed processes would prove the soundness of the synthesis. Alternatively, we may define a similar set of algebraic laws in terms of finite state machines and prove the soundness directly.

So far, we handle only closed systems but not open systems. Synthesis for open systems asks whether there is an implementation that can be deployed in any malevolent environment. To avoid the undecidability of the distributed synthesis problem for open systems, the same lightweight approach presented in Section 5.4 is adopted. The synthesized state machine for the environment (parallel composition of all state machines for environment objects) is verified to be equivalent to

(or simulates) the user-supplied modeling of the environment.

7.4 Refinement of the Distributed Object System

In our combined specification, local actions are defined as operation schemas, which could be implemented by a series of computations constrained by pre/post-condition. It is necessary to refine the distributed object system so as to guarantee that a local action is only engaged with its precondition satisfied, a hot condition shall be satisfied in all circumstance, etc. However, it is difficult to tell if a certain assertion is true after a series of local computations simply because the state space of a Z specification may often be infinite. The problem is further complicated as Z operation schema may take inputs from the environment, which can not be controlled by the system. Our remedy is predicate abstraction, as applied in Chapter 6 for extracting finite state realizations of Object-Z specifications. Predicate abstraction allows us to interpret and then restrict the behaviors of an object based on an abstract view of the data variables, which is essential for our synthesis since an implementable control structure may only contain a finite number of control states.

The abstraction method used in Chapter 6 is amended for abstracting Z packages. In Z semantics, the result of applying an operation outside its precondition is divergence. Thus, in abstraction of a Z package, an operation must be applied at states where its precondition is satisfied. Moreover, in the abstraction interpretation, we guarantee that applying an operation may reach all states where the postcondition may be satisfied. This way, our abstraction is robust with respect to Z data refinement, i.e., weakening precondition and strengthening postcondition. The abstract machine is then used to refine the distributed object system synthesized from the LSC model on an object basis. Invocation of operations that might violate its precondition or result in a state violating a hot condition is systematically pruned.

In order to guarantee the correctness of the synthesized design, we require that the set of predicates for abstraction includes all conditions in the universal charts (as well as the predicate in the initial schema for simplicity). A finite state abstraction of a Z package is built by abstracting both its initial schema and its operation schemas. Because only sound designs are of interest, a local

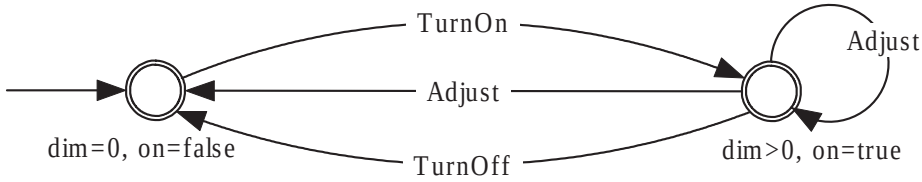


Figure 7.10: Abstraction of the $Light$ package

action shall be invoked only when we are certain no assertions will be violated. Thus, the precondition of the operation is abstracted as $\mathcal{W}(\text{pre}(Operation))$ and its postcondition is abstracted as $\mathcal{S}(\text{post}(Operation, s_a))$, where s_a is an abstract state satisfying the abstract precondition. Intuitively, by replacing the precondition with a more restrictive one, we make sure no precondition shall be violated. By replacing the postcondition with a less restrictive one, we make sure that no hot conditions shall be violated in all circumstances.

Definition 20 *Given a set of predicates P, $M_Z^i \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$ is an abstraction of the Z package associated with object i only if $S \mathrel{\widehat{=}} S_a$ and $S_0 \mathrel{\widehat{=}} \mathcal{W}(initial\ condition)$ and $F \mathrel{\widehat{=}} S$ and Σ is the set of operation schemas in the package and I labels a state with itself and $T \mathrel{\widehat{=}} \{(s_1, e, s_2) : S \times \Sigma \times S \mid s_1 \in \mathcal{W}(pre(e)) \wedge s_2 \in \mathcal{S}(post(e, s_1))\}$.*

Example 7.4.1 (Abstraction of Z package) Assume the set of predicates for abstracting the $Light$ package is $\{dim = 0, on = false, dim > 0\}$, the set of abstract states contains two states: $S_a \mathrel{\widehat{=}} \{dim = 0 \wedge on = false, dim > 0 \wedge on = true\}$. The abstract initial state is exactly the state where $dim = 0 \wedge on = false$. Operation $Adjust$ is abstracted by computing the following:

$$
\begin{aligned}
&\mathcal{W}(\text{pre}(Adjust)) \\
&\quad \mathrel{\widehat{=}} \mathcal{W}(\exists\, dim' : Illumination;\ on' : \mathbb{B} \mid dim' > 0 \Leftrightarrow on' = true \bullet \\
&\qquad on = true \wedge dim' = dim?) && \text{– def. of Precon.} \\
&\quad \mathrel{\widehat{=}} \{dim > 0 \wedge on = true\} && \text{– def. of } \mathcal{W}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}(\text{post}(Adjust, dim > 0 \wedge on = true)) \\
&\quad \mathrel{\widehat{=}} \mathcal{S}(dim > 0 \Leftrightarrow on = true \wedge dim' > 0 \Leftrightarrow on' = true \wedge \\
&\qquad dim > 0 \wedge on = true \wedge on = true \wedge dim' = dim?) \\
&\quad \mathrel{\widehat{=}} \{dim' = 0 \wedge on' = false, dim' > 0 \wedge on' = true\}
\end{aligned}
$$

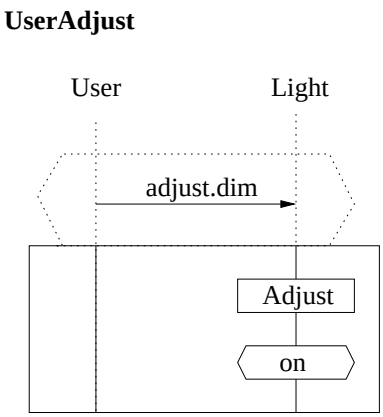


Figure 7.11: Scenario of the LCS: *UserAdjust*

Thus, the abstract operation $Adjust$ is enabled only at the abstract state where on is true, from which both abstract states can be reached. We skip the abstraction of the other operations in the package. Figure 7.10 shows the resultant state machine. **end**

After constructing the abstract state machine from the Z package, the product of M^i_{LSC} and M^i_Z is computed. By removing states labeled with false, we guarantee that no precondition or hot condition is violated. However, the problem is complicated by the uncontrollability of the environment because removing states may put restrictions over inputs from the environment, which is problematic. For instance, if we allow the user to adjust the illumination by setting it to certain values, captured by the universal chart in Figure 7.11. 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). Another important property for open systems is nonblocking, i.e., the design should not introduce any fresh deadlock. The pruning algorithm presented in Chapter 6 is reused to determine whether there is a satisfying design, and synthesizes one if possible by refining the product state machine.

Example 7.4.2 (Pruning state machine) Figure 7.12 presents the state machine for instance $Light$ in scenario *UserAdjust*. The product of the state machines in Figure 7.10 and Figure 7.12 is pre-

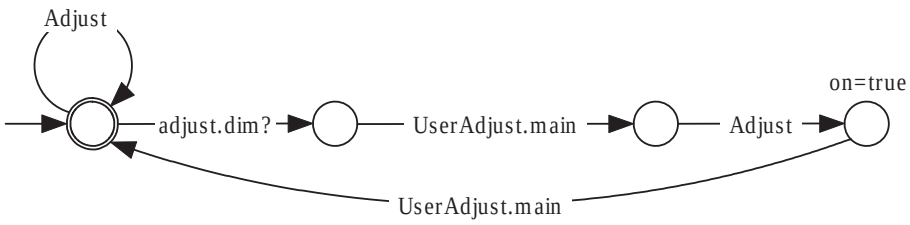


Figure 7.12: State machine synthesized from instance $Light$ in *UserAdjust*

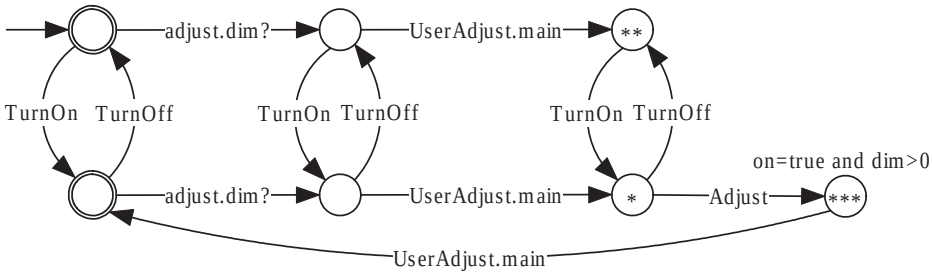


Figure 7.13: Product state machine

sented in Figure 7.13 (where one state labeled with false has been removed). The pruning algorithm is then applied. The * state is removed because $Adjust$ is uncontrollable at the state and the state labeled with $on = false \wedge dim = 0$ is not reachable from the * state by applying $Adjust$ while it does in Figure 7.10. Thus, line 10 of the algorithm presented in Section 6.3 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. After removing states leading to no accepting state, the resultant state machine is shown in Figure 7.14. It is a valid design for closed systems since there are one initial state and accepting states. Intuitively, the design guarantees that the chart *UserAdjust* is satisfied by requiring it is never activated. However, if $user$ is considered as part of the environment, then there is no way to prevent users from activating the chart by sending message $adjust.dim$. In our approach, the synthesized modeling of $User$ has failed to be a simulation of the default modeling (where users

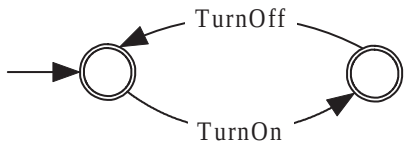


Figure 7.14: Pruned state machine

can initiate any communication at any time). Thus there is no design satisfying this chart. **end**

In the following, we briefly discuss the soundness of the techniques used in this section. In Section 7.3.1, 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 20, local actions can only be engaged within in 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 specification. Though there may be infeasible pathes in the synthesized implementation, an operation may reach a successor state only if the postcondition is satisfied at the successor state. The point is that using the weakened postcondition, we can detect possible violation of hot conditions early in the synthesis process (instead of at run-time). 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. Fair simulation implies fair trace containment. Thus, the pruned state machine is consistent with the specification.

7.5 Automation

We implemented a prototype to experiment with our approach using standard case studies. The experiment tools presented in Chapter 5 and Chapter 6 are reused. The input to our experimental

tool is an XML representation of the Z model and an XML representation of the LSC model. As discussed in Section 6.3.2, a transition in the pruned state machine may be constrained by restricting its postcondition in the pruned state machine, which is not implementable. Two different remedies have been explored. The first remedy is to guard each invocation of the action with a proper guard condition as we did in Section 6.3.2. For partially pruned nondeterministic choices, the transitions shall be guarded with the weakest precondition that guarantees the reachability of the desired state. After that, executable implementation can be synthesized straightforwardly with the implementation of each local actions supplied by users. As long as the implementation of local actions conforms to its precondition/postcondition specification, our synthesized prototype remains sound. However, a reasonable guard condition must not involve any primed variables. Computing the weakest precondition requires elimination of the primed variables, which is in general undecidable. Therefore, this remedy is unlikely to be fully automated. The other remedy is to generate a set of proof obligations for nondeterministic choices 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/post-condition so as to make sure the operation satisfies the more restrictive post-condition at the system states.

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. Later, we limit the number of overlapping activations of the same chart as a way to further reduce the size of the local state machines. For instance, all universal charts except *PeopleOut* allow no overlapping activations in the LCS example. 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 for concurrency [158] to generate structural prototypes.

To handle systems with infinite data space, we adopt predicate abstraction to construct an abstract view of system behaviors in terms of finite assertions. In general, the size of the abstract state machine is exponential in the number of predicates for abstraction. It is the most time-consuming operation in our method. However, it remains affordable because only one Z package is abstracted at a time and there are unlikely to be large number of conditions concerning one object. Our abstraction

method constructs an abstract state graph by paying a reasonable price. In our prototype, a sound approximation of the function $\mathcal{W}$ and $\mathcal{S}$ is used. To further speed up the abstraction as well as to guarantee termination of the proving, every lemma is proved in a limited amount of time. The time limit is set as a user option. The date aspect of the LCS example is slightly trivial. As for reference, in a vending machine example where there are state variables with infinite domain and multiple operation schemas, all together 190 lemmas are generated and all 105 provable lemmas are proved without user interaction in minutes. The lift control system is also modified to handle system with arrays of variables (refer to [148] for detail). In addition, a number of tricks have been used to reduce the abstract state space, for instance removing a false state by considering co-relation between the predicates and the state invariant before abstract. The complexity of our pruning algorithm is polynomial time in terms of the number of states. So are the operations we perform over the state machine. Thus, they are carried out in reasonably speedy fashion.

7.6 Summary and Discussion

In this work, we present 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. Firstly, we propose an intuitive integration of Z model and LSC model, which is capable of modeling systems with not only complicated data structures but also complex interactive behaviors. Secondly, we develop a systematic way of synthesizing distributed finite state designs all the way from the combined specifications. Thirdly, we developed an experimental tool to automate our method. One of the possible future works is to generate implementations other than JAVA programs from the synthesized design, for example SystemC [61] or Spec# [8]. We may as well formally explore the notion of refinement in terms of the combined interaction and state-based modeling. For instance, we may investigate how refinement in the B method may cooperate with refinement in LSC so that implementation can be deduced step by step from the combined specification.

The integration of the Z specification language and LSC is related to works on integrated specification languages [160, 102, 157, 138, 24, 36]. The characteristic of our work is that we provide

a synthesis method in addition to system specification. Our synthesis method may suffer from being over-restrictive sometimes. One of the reasons has already been mentioned in Section 7.3.1. Another reason is that because our pruning applies on an object basis, valid designs requiring cooperation of multiple system objects are not possible. For instance, inputs to an operation from other system components are controllable if we consider the global state machine. For example, in Figure 7.1, the value of dim from $RoomController$ is actually never 0 from the whole system's view. Disallowing such designs is a sacrifice we have to make if we do not construct the global state machine. The third reason is the limited power of proof systems. 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 the abstraction suggests that applying an uncontrollable operation may result in an undesired state from a given state whereas in fact it cannot, then the uncontrollable operation will be prohibited from happening. Nevertheless, our approach serves as a promising method to apply synthesis techniques to complicated system specifications, and it can be applied to other integrations of state-based and interaction-based modeling as well.

Chapter 8

Conclusion

'Would you tell me, please, which way I ought to go from here?'
'That depends a good deal on where you want to get to', said the cat.
'I don't much care where' said Alice.
'Then it doesn't matter which way you go', said the Cat.
'So long as I get somewhere, ' Alice added as an explanation.
'Oh, you're sure to do that,' said the Cat,
'if you only walk long enough.'
- *Alice's Adventures in Wonderland*, Lewis Carroll

In this chapter, we summarize the main contributions of this thesis and present possible directions for further research.

8.1 Contributions

The scheme of this thesis is to identify and study formal specification languages which are complementary to each other in terms of visualization, verification or synthesis. We explored many well-established languages and notations so as to identify the similarity and difference between them.

Transformation techniques are then used to connect those complementary ones for practical purposes. Our approach is however not restricted to particular languages or notations. It demonstrates general complementary relationships between logic-based formalisms and visual formalisms.

The works presented in this thesis can be fully integrated with other software products and processes all along system development life cycle. For instance, visualization (Chapter 3) offers graphical representation of logic-based system models in the specification stage. The work on verification reveals inconsistency of the system specification. The works on synthesis provides a constructive method for connecting the specification, design, and implementation stage. In the following, we discuss the detailed contributions of this thesis.

This thesis successfully demonstrated that though logic-based formalisms and visual formalisms may vary vastly in syntax, they may often share a common semantic basis like trace semantics. Based on the common semantic basis, sound transformation from logic-based specifications to diagrammatic notations allows visualization of logic-based models. The author believes that logic-based formalisms are a more precise and thus safer means for stating system requirements than diagrams. Mechanized visualization allows system engineering starting with logic-based formalism enjoy the visual power of modeling languages like UML.

Detecting inconsistency in system specifications is vital in the process development process. It is commonly known that the earlier the inconsistency and errors are exposed, the more resource and human effort are saved for implementation of the desired software system. This thesis developed verification methods for both logic-based formalisms and visual formalisms using transformation techniques. It has been shown that existing mature model checkers can be applied to formalisms other than those intended effectively.

Verification based on the transformation technique is inexpensive yet effective. It has been applied to a large scale of languages and notations. Our research has influenced research activities on applying formal methods languages and tools to the web domain. For instance, Liu *el at* developed a Timed Automata semantics for orchestration of web service so that Orc specification [112] can be transformed to Timed Automata and consequently verified by UPPAAL [47]. Dong *el at* developed

a tools environment for reusing formal methods tool for proving web ontology [45]. Sun *el at* developed a transformation from web services to LSC so as to use *Play-engine* for simulation and verification [151].

One of the ultimate goals for software engineering is to automatically generate low-level implementations from high-level specification. A main contribution of this thesis is the investigation on this automated development process. Systematic ways of generating prototypes from state-based specifications (e.g., Object-Z), or scenario-based diagrams (e.g., Live Sequence Chart), or combination of both (e.g., Live Sequence Chart combined with Z specification) have been developed. This thesis discussed the complexity dealing with the problem of synthesis and compared our methods with existing approaches. To the best of our knowledge, the synthesis work presented in this thesis is the first attempt to mechanically generate prototypes from specifications of systems with intensive interactive behaviors as well as complicated data and functional requirements.

8.2 Future Research Trends

The following topics, arising out of this thesis, seem worthy of future research.

This thesis developed a number of tools providing support for various tasks, which form a near-complete framework for system specifying, verifying, developing, and testing. Each link in Figure 1.1 illustrates an automated transformation in the name of either visualization or verification. Figure 1.2 shows the tools developed in the work of synthesis. The two figures serve as a blueprint of the framework we shall develop as one of the future works. The framework shall allow system specification or design using user favored modeling techniques like Z, Object-Z, CSP, MSC, LSC, or any combination of them. Thus, we shall develop friendly user interface for editing logic-based specifications as well as drawing diagrams. Alternatively, we shall support system designs created externally using existing popular tools like UML editing tools. For works on verification, we shall hide underlying reasoning details and connect analysis results to the level of user specification. Hints for refining the specification shall be highlighted properly, e.g., using different fonts for logic-based specification or using emphasized drawing for visual specification. For instance, verification results

of LSC models from FDR shall automatically feed back to the user. Any counter example will be displayed graphically so as to guide the refinement of the LSC model. Once this is done, users with little or no knowledge of CSP or FDR may benefit.

In the work on synthesis, few timing issues have been discussed. One of the challenging tasks is to investigate whether our works on synthesis extend to system specifications with qualitative timing behaviors. For instance, we shall investigate whether the approach presented in Chapter 5 handles LSC with typical timing events. Timed CSP seems to be a promising media to carry out the discussion since the symmetry and transitivity laws of parallel composition hold in Timed CSP as well. However, a global shared clock is inevitable in the context of LSC, which presents a real challenge for the distributed synthesis. Similarly, we shall extend our work presented in Chapter 6 so that the history invariant may contain explicit time variables. In general, timed synthesis remains as a tough research task [104].

In our works on synthesis of implementations, prototypes in JAVA are mechanically generated. We shall improve our code generation to aim at product quality programs. Issues like code optimization, code reusability, shall be taken into account. We remark that it would be as straightforward and of more use to generate implementation in programming languages other than JAVA. Two of them are of particular interest. One is SystemC [61]. The reasons are, SystemC supports high-level modeling, hardware-software partitions, and it is easy to implement different channel types in SystemC. The other is Spec# [8] because it offers a facility to write specifications that capture programmer intentions about how methods and data are to be used and the compiler emits run-time checks to enforce these specifications. This capability offers a sound way of enforcing (shared) hot conditions.

In Chapter 7, we briefly mentioned that our method is robust with respect to Z data refinement. A challenging task is to formally explore the notion of refinement in terms of the combined interaction and state-based modeling. For instance, what kinds of data refinement shall co-exist with the notion of refinement in LSC. The latter typically means expanding a sub-chart with more details. In particular, we may investigate how refinement in the B method [92] may cooperate with refinement in LSC so that implementation can be derived step by step from the combined specification.

Bibliography

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 304–313. ACM Press, 2000.

[4] R. Alur, T. A. Henzinger, and E. D. Sontag, editors. *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Rutgers University, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1996.

[5] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proceedings of the 10th International Conference on Concurrency Theory*, pages 114–129. Springer, 1999.

[6] J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 18(1), 1990.

[7] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI 2001*, pages 203–213, 2001.

[8] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*, pages 49–69, 2004.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In Alur et al. [4], pages 232–243.

[10] R. Berghammer and B. v. Karger. Formal Derivation of CSP Programs From Temporal Specifications. In *Mathematics of Program Construction*, pages 181–196, 1995.

[11] E. A. Boiten, J. Derrick, and G. Smith, editors. *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.

[12] C. Bolton and J. Davies. Activity Graphs and Processes. In W. Grieskamp, T. Santen, and W. Stoddart, editors, *Proceedings of IFM 2000*, pages 77–96. Springer, 2000.

[13] Y. Bontemps. *Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), April 2005.

[14] Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into Automata. In *"Workshop: Scenarios and State-Machines, ICSE'02"*, 2002.

[15] Y. Bontemps, P. Heymans, and P. Schobbens. From Live Sequence Charts to State Machines and Back: A Guided Tour. *IEEE Transactions on Software Engineering*, 31(12):999–1014, 2005.

[16] Y. Bontemps, P. Heymans, and P. Schobbens. Lightweight Formal Methods for Scenario-Based Software Engineering. In S. Leue and T. Systa, editors, *Scenarios*, volume 3466 of *Lecture Notes in Computer Science*, pages 174–192, 2005.

[17] Y. Bontemps and P. Schobbens. The Complexity of Live Sequence Charts. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005*, pages 364–378, 2005.

[18] Y. Bontemps, P. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, July 2004.

[19] E. Brinksma. From Data Structure to Process Structure. In K. G. Larsen and A. Skou, editors, *CAV 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 244–254. Springer, 1991.

[20] P. J. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, april 1999.

[21] P. J. Brooke and R. F. Paige. The Design of a Tool-Supported Graphical Notation for Timed CSP. In M. J. Butler, L. Petre, and K. Sere, editors, *Proc. Integrated Formal Methods 2002 (IFM'02)*, pages 299–318, Turku, Finland, May 2002.

[22] S. D. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. In S. D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer, 1984.

[23] J. R. Büchi and L. H. Landweber. Solving Sequential Conditions by Finite State Strategies. *Transactions on the American Mathematical Society*, 138:295–311, 1969.

[24] M. J. Butler and M. Leuschel. Combining CSP and B for Specification and Property Verification. In Fitzgerald et al. [57], pages 221–236.

[25] A. Cavalcanti and J. Woodcock. A Weakest Precondition Semantics for Z. *The Computer Journal*, 41(1):1–15, 1998.

[26] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In Boiten et al. [11], pages 128–147.

[27] A. Church. Logic, Arithmetic and Automata. In *Proceeding of International Congr. Math.*, 1960.

[28] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *POPL 1983*, pages 117–126, 1983.

[29] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Other working group members: R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T.Henzinger, G. Hozmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushbby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave.

[30] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 391–395. Springer, 2001.

[31] A. Coombes and J. A. McDermid. Using Diagrams to Give a Formal Specification of Timing Constraints in Z. In *Z User Workshop*, pages 119–130, 1992.

[32] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[33] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[34] J. Davies and C. Crichton. Using State Diagrams to Describe Concurrent Behaviour. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*, volume 2885 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2003.

[35] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In Alur et al. [4], pages 208–219.

[36] J. Derrick and E. A. Boiten. Combining Component Specifications in Object-Z and CSP. *Formal Aspect of Computing*, 13(2):111–127, 2002.

[37] R. Diaconescu and K. Futatsugi. An overview of CafeOBJ. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[38] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284. Springer, 1993.

[39] E. W. Dijkstra. Programming: From Craft to Scientific Discipline. In E. Morlet and D. Ribbens, editors, *International Computing Symposium 1977*, pages 23–30. North-Holland, 1977.

[40] E.W. Dijkstra. *A Discipline of Programming*. International Series in Computer Science. Prentice-Hall, 1976.

[41] D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *Computer Aided Verification*, pages 255–265, 1991.

[42] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 483–498. Springer, 2004.

[43] J. S. Dong, P. Hao, S. C. Qin, and X. Zhang. The Semantics and Tool Support of OZTA. In Lau and Banach [90], pages 66–80.

[44] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Tool for Verification of Timed CSP. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM 2006)*, 2006. To appear.

[45] J. S. Dong, Y. F. Li, J. Sun, and Y. Z. Feng. A Tools Environment for Developing and Reasoning about Ontologies. In *APSEC 2005*. IEEE Computer Society, 2005.

[46] J. S. Dong, Y. F. Li, J. Sun, J. Sun, and H. Wang. XML-Based Static Type Checking and Dynamic Visualization for TCOZ. In George and Miao [60], pages 311–322.

[47] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration via Timed Automata. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM 2006)*, 2006. To appear.

[48] J. S. Dong, S. C. Qin, and J. Sun. Generating MSCs from an Integrated Formal Specification Language. In Boiten et al. [11], pages 168–186.

[49] J. S. Dong, J. Sun, H. Wang, C. H. Lee, and H. B. Lee. Analysing Web Ontology in Alloy: A Military Case Study. In *SEKE 2003*, pages 542–546, 2003.

[50] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing Series. Macmillan, March 2000.

[51] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994. Also in a special issue of *Computer Standards and Interfaces* on Formal Methods and Standards, September 1995.

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

[53] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer System Sciences*, 18(2):194–211, 1979.

[54] D. A. Fisher. The Interaction Between the Preliminary Designs and the Technical Requirements for the DoD Common High Order Language. In *ICSE 1978*, pages 82–83, 1978.

[55] J. Fisher, D. Harel, E. J. A. Hubbard, N. Piterman, M. J. Stern, and N. Swerdlin. Combining State-Based and Scenario-Based Approaches in Modeling Biological Systems. In V. Danos

and V. Schächter, editors, *CMSB 2004*, volume 3082 of *Lecture Notes in Computer Science*, pages 236–241. Springer, 2004.

[56] K. Fisler and M. Y. Vardi. Bisimulation Minimization and Symbolic Model Checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[57] J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*. Springer, 2005.

[58] Formal System Europe. Failure Divergence Refinement. http://www.fsel.com/, 2003.

[59] The Apache Software Foundation. XERCES JAVA Parser 2.7.1 Release. http://xml.apache.org/xerces-j/index.html, 2005.

[60] C. George and H. K. Miao, editors. *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, volume 2495 of *Lecture Notes in Computer Science*. Springer, 2002.

[61] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[62] J. V. Guttaq, J. J. Horning, S. J. Garland, and K. D. Jones. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[63] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[64] P. Hao, J. S. Dong, S. C. Qin, J. Sun, and Y. Wang. Timed automata patterns: Projections from tcoz. 2006. Submitted for review.

[65] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[66] D. Harel. From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.

[67] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.

[68] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *International Journal on Foundations of Computer Science*, 13(1):5–51, 2002.

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

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

[71] D. Harel and R. Marelly. *Play-Engine User's Guide*, 2003.

[72] O. Haugen and K. Stølen. STAIRS - Steps to Analyze Interactions with Refinement Semantics. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003*, volume 2863 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2003.

[73] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, 1987.

[74] K. M. V. Hee, editor. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, 1994.

[75] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair Simulation. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 1997.

[76] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[77] R. M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[78] C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Joural of ACM*, 50(1):63–69, 2003.

[79] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

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

[81] ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.

[82] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[83] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International(UK) Ltd., 1990.

[84] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *WWW (Alternate Paper Tracks) 2003*, 2003.

[85] J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Charts. In T. Margaria and Y. Wang, editors, *TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 512–527. Springer, 2001.

[86] P. Kosiuczenko and M. Wirsing. Formalizing and Executing Message Sequence Charts via Timed Rewriting. *Electrical Notes on Theoretical Computer Science*, 25:1–25, 1999.

[87] K. Koskimies and E. Mäkinen. Automatic Synthesis of State Machines from Trace Diagrams. *Softw. Pract. Exper.*, 24(7):643–658, 1994.

[88] I. Kruger. Modeling and Synthesis with MSC Extensions for Broadcasting, Overlapping, Preemptive, and Triggered Collaborations. In *Workshop on Scenarios and State Machines at ICSE 2003*, 2003.

[89] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In N. Halbwachs and L. D. Zuck, editors, *TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2005.

[90] K. K. Lau and R. Banach, editors. *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*. Springer, 2005.

[91] L. Lavagno, G. Martin, and B. Selic. *UML for Real: Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.

[92] M. Leuschel and M. J. Butler. Automatic Refinement Checking for B. In Lau and Banach [90], pages 345–359.

[93] Y. F. Li, J. Sun, G. Dobbie, J. Sun, and H. Wang. Validating Semistructured Data Using OWL. In *Proceedings of International Conference on Advances in Web-Age Information Management (WAIM 2006)*, pages 520–531, 2006.

[94] S. Y. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, 1998.

[95] S. Y. Liu and J. Woodcock, editors. *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*. IEEE Computer Society, 2005.

[96] Y. Liu and J. Sun. Algorithmic Design Using Object-Z for Twig XML Queries Evaluation. In *International Workshop on Web Languages and Formal Methods (WLFM'05)*, 2005.

[97] 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*, 6:11–44, Jan 1995.

[98] P. Madhusudan and P. S. Thiagarajan. Branching Time Controllers for Discrete Event Systems. *Theoretical Computer Science*, 274:117–149, 2002.

[99] P. Madhusudan and P.S. Thiagarajan. A Decidable Class of Asynchronous Distributed Controllers. In L. Brim, P. Janar, M. Ketinsky, and A. Kuera, editors, *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, Lecture Notes in Computer Science, pages 145 – 160. Springer, 2002.

[100] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[101] B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

[102] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Press.

[103] B. P. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In Wing et al. [159], pages 1166–1185.

[104] O. Maler, A. Pnueli, and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In E. W. Mayr and C. Puech, editors, *STACS 1995*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 1995.

[105] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[106] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[107] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions Programming Languages and Systems*, 6(1):68–93, 1984.

[108] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proceedings of OOPSLA'02*, pages 83–100, 2002.

[109] S. Mauw and M. A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[110] D. May. OCCAM. *SIGPLAN Notices*, 18(4):69–79, 1983.

[111] R. Miarka, E. A. Boiten, and J. Derrick. Guards, Preconditions, and Refinement in Z. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2000.

[112] J. Misra and W. Cook. *Computation Orchestration: A Basis for Wide-Area Computing*. NATO ASI Series, 2005.

[113] L. Murray, D. A. Carrington, I. MacColl, J. McDonald, and P. A. Strooper. Formal Derivation of Finite State Machines for Class Testing. In Jonathan P. Bowen, Andreas Fett, and Michael G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 1998.

[114] M. Y. Ng and M. J. Butler. Tool Support for Visualizing CSP in UML. In George and Miao [60], pages 287–298.

[115] O. Nierstrasz. Regular Types for Active Objects. In *OOPSLA 1993*, pages 1–15, 1993.

[116] J. Offutt, S. Y. Liu, A. Abdurazik, and P. Ammann. Generating Test Data From State-based Specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[117] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer.

[118] F. Peschanski and D. Julien. When Concurrent Control Meets Functional Requirements, or Z + Petri-Nets. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003*, volume 2651 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2003.

[119] C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress 62*, pages 386–390, 1963.

[120] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer science*, pages 46–57, 1977.

[121] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM Symposium Principles of Programming Languages (POPL 1989)*, pages 179–190, 1989.

[122] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesize. In *Proceedings 31st IEEE Symposium on Foundation of Computer Science*, 1990.

[123] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Special Issue on Discrete Event Dynamic Systems*, 36(1):81–98, Jan 1989.

[124] Rational Ltd. Rational rose. http://www-306.ibm.com/software/rational/, 2004.

[125] G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. In L. Kott, editor, *ICALP 1986*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer, 1986.

[126] J. H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI 1991*, pages 293–305, 1991.

[127] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *International Conference on Software Engineering*, pages 105–118, 1992.

[128] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[129] A. Roychoudhury and P. S. Thiagarajan. Communicating Transaction Processes. In *ACSD 2003*, pages 157–166. IEEE Computer Society, 2003.

[130] F. Ruiz, A. Vizcaíno, F. García, and M. Piattini. Using XMI and MOF for Representation and Interchange of Software Processes. In *DEXA Workshops 2003*, pages 739–744. IEEE Computer Society, 2003.

[131] M. Saaltink. The Z/EVES System. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

[132] D. Sangiorgi and D. Walker. *The Pi-Calculus*. Cambridge University Press, 2004.

[133] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, Oxford University, 1998.

[134] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 640–675. Springer, 1992.

[135] S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings of the Chalmers Workshop on Concurrency, 1991*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[136] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.

[137] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[138] G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems-An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[139] G. Smith and K. Winter. Proving Temporal Properties of Z Specifications Using Abstraction. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *ZB 2003*, volume 2651 of *Lecture Notes in Computer Science*, pages 260–279. Springer, 2003.

[140] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulæ. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[141] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 1989.

[142] P. Stocks and D. A. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

[143] J. Sun and J. S. Dong. Extracting FSMs from Object-Z Specifications with History Invariants. In Liu and Woodcock [95], pages 96–105.

[144] J. Sun and J. S. Dong. Model Checking Live Sequence Charts. In Liu and Woodcock [95], pages 529–538.

[145] J. Sun and J. S. Dong. Synthesis of Distributed Processes from Scenario-Based Specifications. In Fitzgerald et al. [57], pages 415–431.

[146] J. Sun and J. S. Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.

[147] J. Sun and J. S. Dong. From Live Sequence Charts to Distributed Implementations. 2006. Submitted for review.

[148] J. Sun and J. S. Dong. Verfication and Synthesis of Live Sequence Charts. `http://www.comp.nus.edu.sg/~sunj/LSC2CSP.html`, 2006.

[149] J. Sun, J. S. Dong, J. Liu, and H. Wang. An XML/XSL Approach to Visualize and Animate TCOZ. In *APSEC 2001*, pages 453–460. IEEE Computer Society, 2001.

[150] J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13:329–356, 2002.

[151] J. Sun, Y. F. Li, H. Wang, and J. Sun. Visualizing and Simulating Semantic Web Services Ontologies. In Lau and Banach [90], pages 435–449.

[152] W. Thomas. On the Synthesis of Strategies in Infinite Games. In E.W. Mayr and C. Puech, editors, *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science, STACS '95*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, 1995. Springer.

[153] S. Uchitel and J. Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *ICSE 2001*, pages 188–197. IEEE Computer Society, 2001.

[154] UML Group. OMG UML Version 1.5. http://www.uml.org/, June 2002.

[155] H. Wang, J. S. Dong, J. Sun, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. In *Mutilagent and Grid Systems*, 2006. To appear.

[156] T. Wang, A. Roychoudhury, R. H. C. Yap, and S. C. Choudhary. Symbolic Execution of Behavioral Requirements. In B. Jayaraman, editor, *PADL 2004*, volume 3057 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2004.

[157] H. Wehrheim. Data Abstraction for CSP-OZ. In Wing et al. [159], pages 1028–1047.

[158] P. H. Welch, J. R. Aldous, and J. Foster. CSP Networking for JAVA (JCSP.net). In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (2)*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer, 2002.

[159] J. M. Wing, J. Woodcock, and J. Davies, editors. *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*. Springer, 1999.

[160] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[161] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.

Appendix A

Glossary of Z Notation

This appendix presents a glossary of the Z notation used in this thesis. The glossary is based on the glossary of Z notation presented in Hayes [73] with modifications to reflect more closely the more recent Z notation of Spivey [141].

Mathematical Notation

Definitions and declarations

Let x, x_k be identifiers and let T, T_k be non-empty, set-valued expressions.

$LHS == RHS$	Definition of LHS as syntactically equivalent to RHS.
$LHS[X_1, X_2, \ldots, X_n] == RHS$	Generic definition of LHS, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets.
$x : T$	A declaration, $x : T$, introduces a new variable *x* of type *T*.

$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$

List of declarations.

$x_1, x_2, \ldots, x_n : T$ $\quad == x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$

$[X_1, X_2, \ldots, X_n]$ Introduction of free types named $X_1, X_2, \ldots, X_n$.

Logic

Let P, Q be predicates and let D be a declaration or a list of declarations.

$true, false$ Logical constants.

$\neg\, P$ Negation: "not P".

$P \wedge Q$ Conjunction: "P and Q".

$P \vee Q$ Disjunction: "P or Q or both".

$P \Rightarrow Q$ $\quad == (\neg\, P) \vee Q$

Implication: "P implies Q" or "if P then Q".

$P \Leftrightarrow Q$ $\quad == (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Equivalence: "P is logically equivalent to Q".

$\forall\, x : T \bullet P$ Universal quantification: "for all x of type T, P holds".

$\exists\, x : T \bullet P$ Existential quantification: "there exists an x of type T such that P holds".

$\exists_1\, x : T \bullet P$ Unique existence: "there exists a unique x of type T such that P holds".

$\forall\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

"For all x_1 of type T_1, x_2 of type T_2, $\ldots$, and x_n of type T_n, P holds."

$\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

Similar to $\forall$.

$\exists_1\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

Similar to $\forall$.

$\forall\, D \mid P \bullet Q$ $\Leftrightarrow \forall\, D \bullet P \Rightarrow Q$

$\exists\, D \mid P \bullet Q$ $\Leftrightarrow \exists\, D \bullet P \wedge Q$

$t_1 = t_2$ Equality between terms.

$t_1 \neq t_2$ $\Leftrightarrow \neg\, (t_1 = t_2)$

Sets

Let X be a set; S and T be subsets of X; t, t_k terms; P a predicate; and D declarations.

$t \in S$ Set membership: "t is a member of S".

$t \notin S$ $\Leftrightarrow \neg\, (t \in S)$

$S \subseteq T$ $\Leftrightarrow (\forall\, x : S \bullet x \in T)$

Set inclusion.

$S \subset T$ $\Leftrightarrow S \subseteq T \wedge S \neq T$

Strict set inclusion.

$\varnothing$ The empty set.

$\{t_1, t_2, \ldots, t_n\}$ The set containing the values of terms $t_1, t_2, \ldots, t_n$.

$\{x : T \mid P\}$ The set containing exactly those x of type T for which P holds.

$(t_1, t_2, \ldots, t_n)$ Ordered n-tuple of $t_1, t_2, \ldots, t_n$.

$T_1 \times T_2 \times \ldots \times T_n$

Cartesian product: the set of all n-tuples such that the kth component is of type T_k.

$first(t_1, t_2, \ldots, t_n)$

$== t_1$

Similarly, $second(t_1, t_2, \ldots, t_n) == t_2$, etc.

$\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$

The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.

$\{D \mid P \bullet t\}$ The set of values of the term t for the variables declared in D ranging over all values for which P holds.

$\{D \bullet t\}$ $== \{D \mid true \bullet t\}$

$\mathbb{P}\, S$ Powerset: the set of all subsets of S.

$\mathbb{P}_1\, S$ $== \mathbb{P}\, S \setminus \{\varnothing\}$

The set of all non-empty subsets of S.

$\mathbb{F}\, S$ $== \{\, T : \mathbb{P}\, S \mid T \text{ is finite } \}$

Set of finite subsets of S.

$\mathbb{F}_1\, S$ $== \mathbb{F}\, S \setminus \{\varnothing\}$

Set of finite non-empty subsets of S.

$S \cap T$ $== \{x : X \mid x \in S \wedge x \in T\}$

Set intersection.

$S \cup T$ $== \{x : X \mid x \in S \vee x \in T\}$

Set union.

$S \setminus T$	$== \{x : X \mid x \in S \wedge x \notin T\}$ Set difference.
$\bigcap SS$	$== \{x : X \mid (\forall S : SS \bullet x \in S)\}$ Intersection of a set of sets; SS is a set containing as its members subsets of X, i.e. $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\bigcup SS$	$== \{x : X \mid (\exists S : SS \bullet x \in S)\}$ Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\#S$	Size (number of distinct members) of a finite set.

Numbers

$\mathbb{R}$	The set of real numbers.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$\mathbb{N}$	$== \{n : \mathbb{Z} \mid n \geq 0\}$ The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	$== \mathbb{N} \setminus \{0\}$ The set of strictly positive natural numbers.
$m \mathinner{..} n$	$== \{k : \mathbb{Z} \mid m \leq k \wedge k \leq n\}$ The set of integers between m and n inclusive.
$min\ S$	Minimum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $min\ S \in S \wedge (\forall x : S \bullet x \geq min\ S)$.
$max\ S$	Maximum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $max\ S \in S \wedge (\forall x : S \bullet x \leq max\ S)$.

Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$ $\quad == \mathbb{P}(X \times Y)$

The set of relations between X and Y.

$x \mathrel{\underline{R}} y$ $\quad == (x, y) \in R$

x is related by R to y.

$x \mapsto y$ $\quad == (x, y)$

$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$

$== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

The relation relating x_1 to y_1, x_2 to y_2, ..., and x_n to y_n.

dom R $\quad == \{x : X \mid (\exists\, y : Y \bullet x \mathrel{\underline{R}} y)\}$

The domain of a relation: the set of *x* components that are related to some *y*.

ran R $\quad == \{y : Y \mid (\exists\, x : X \bullet x \mathrel{\underline{R}} y)\}$

The range of a relation: the set of *y* components that some *x* is related to.

$R_1 \fatsemi R_2$ $\quad == \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \wedge y\, R_2\, z)\}$

Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$.

$R_1 \circ R_2$ $\quad == R_2 \fatsemi R_1$

Relational composition. This form is primarily used when R_1 and R_2 are functions.

$R^{\sim}$ $\quad == \{y : Y;\ x : X \mid x \mathrel{\underline{R}} y\}$

Transpose of a relation R.

$\mathrm{id}\, S$	$== \{x : S \bullet x \mapsto x\}$ Identity function on the set S.
R^k	The homogeneous relation R composed with itself k times: given $R : X \leftrightarrow X$, $R^0 = \mathrm{id}\, X$ and $R^{k+1} = R^k \fatsemi R$.
R^+	$== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$ $= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \wedge Q \fatsemi Q \subseteq Q\}$ Transitive closure.
R^*	$== \bigcup\{n : \mathbb{N} \bullet R^n\}$ $= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \wedge R \subseteq Q \wedge Q \fatsemi Q \subseteq Q\}$ Reflexive transitive closure.
$R(\!\mid S \mid\!)$	$== \{y : Y \mid (\exists\, x : S \bullet x \mathrel{\underline{R}} y)\}$ Image of the set S through the relation R.
$S \lhd R$	$== \{x : X;\ y : Y \mid x \in S \wedge x \mathrel{\underline{R}} y\}$ Domain restriction: the relation R with its domain restricted to the set S.
$S \mathrel{\lhd\!\!\!-} R$	$== (X \setminus S) \lhd R$ Domain subtraction: the relation R with the elements of S removed from its domain.
$R \rhd T$	$== \{x : X;\ y : Y \mid x \mathrel{\underline{R}} y \wedge y \in T\}$ Range restriction to T.
$R \mathrel{\rhd\!\!\!-} T$	$== R \rhd (Y \setminus T)$ Range subtraction of T.
$R_1 \oplus R_2$	$== (\mathrm{dom}\, R_2 \mathrel{\lhd\!\!\!-} R_1) \cup R_2$ Overriding; $R_1, R_2 : X \leftrightarrow Y$.

Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$	The function f applied to t.
$X \nrightarrow Y$	$== \{f : X \leftrightarrow Y \mid (\forall\, x : \mathrm{dom}\, f \bullet (\exists_1\, y : Y \bullet x\ \underline{f}\ y))\}$ The set of partial functions from X to Y.
$X \rightarrow Y$	$== \{f : X \nrightarrow Y \mid \mathrm{dom}\, f = X\}$ The set of total functions from X to Y.
$X \rightarrowtail\!\!\!\!\!\rightarrow Y$	$== \{f : X \nrightarrow Y \mid (\forall\, y : \mathrm{ran}\, f \bullet (\exists_1\, x : X \bullet x\ \underline{f}\ y))\}$ The set of partial one-to-one functions (partial injections) from X to Y.
$X \rightarrowtail Y$	$== \{f : X \rightarrowtail\!\!\!\!\!\rightarrow Y \mid \mathrm{dom}\, f = X\}$ The set of total one-to-one functions (total injections) from X to Y.
$X \twoheadrightarrow Y$	$== \{f : X \nrightarrow Y \mid \mathrm{ran}\, f = Y\}$ The set of partial onto functions (partial surjections) from X to Y.
$X \twoheadrightarrow Y$	$== (X \twoheadrightarrow Y) \cap (X \rightarrow Y)$ The set of total onto functions (total surjections) from X to Y.
$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$	$== (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$ The set of total one-to-one onto functions (total bijections) from X to Y.
$X \nrightarrow\!\!\!\!\!\rightarrow Y$	$== \{f : X \nrightarrow Y \mid f \in \mathbb{F}(X \times Y)\}$ The set of finite partial functions from X to Y.
$X \rightarrowtail\!\!\!\!\!\nrightarrow Y$	$== \{f : X \rightarrowtail Y \mid f \in \mathbb{F}(X \times Y)\}$ The set of finite partial one-to-one functions from X to Y.

$(\lambda\, x : X \mid P \bullet t)$ $== \{x : X \mid P \bullet x \mapsto t\}$

Lambda-abstraction: the function that, given an argument x of type X such that P holds, gives a result which is the value of the term t.

$(\lambda\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet t)$

$== \{x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$

disjoint$[I, X]$ $== \{S : I \nrightarrow \mathbb{P}\, X \mid \forall\, i, j : \mathrm{dom}\, S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \varnothing\}$

Pairwise disjoint; where I is a set and S an indexed family of subsets of X (i.e. $S : I \nrightarrow \mathbb{P}\, X$).

S <u>partitions</u> T $== S \in \mathrm{disjoint} \wedge \bigcup \mathrm{ran}\, S = T$

Sequences

Let X be a set; A and B be sequences with elements taken from X; and $a_1, \ldots, a_n$ terms of type X.

seq X $== \{A : \mathbb{N}_1 \nrightarrow X \mid (\exists\, n : \mathbb{N} \bullet \mathrm{dom}\, A = 1..n)\}$

The set of finite sequences whose elements are drawn from X.

$\mathrm{seq}_\infty\, X$ $== \{A : \mathbb{N}_1 \nrightarrow X \mid A \in \mathrm{seq}\, X \vee \mathrm{dom}\, A = \mathbb{N}_1\}$

The set of finite and infinite sequences whose elements are drawn from X.

$\#A$ The length of a finite sequence A. (This is just '#' on the set representing the sequence.)

$\langle\rangle$ $== \{\}$

The empty sequence.

$\mathrm{seq}_1\, X$ $== \{s : \mathrm{seq}\, X \mid s \neq \langle\rangle\}$

The set of non-empty finite sequences.

$\langle a_1, \ldots, a_n \rangle$ $= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$

$\langle a_1, \ldots, a_n \rangle \frown \langle b_1, \ldots, b_m \rangle$

$= \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$

Concatenation.

$\langle\rangle \frown A = A \frown \langle\rangle = A.$

$head\ A$ The first element of a non-empty sequence:

$A \neq \langle\rangle \Rightarrow head\ A = A(1).$

$tail\ A$ All but the head of a non-empty sequence:

$tail\ (\langle x \rangle \frown A) = A.$

$last\ A$ The final element of a non-empty finite sequence:

$A \neq \langle\rangle \Rightarrow last\ A = A(\#A).$

$front\ A$ All but the last of a non-empty finite sequence:

$front\ (A \frown \langle x \rangle) = A.$

$rev\ \langle a_1, a_2, \ldots, a_n \rangle$

$= \langle a_n, \ldots, a_2, a_1 \rangle$

Reverse of a finite sequence; $rev\ \langle\rangle = \langle\rangle$.

$\frown / AA$ $= AA(1) \frown \ldots \frown AA(\#AA)$

Distributed concatenation; where $AA : \mathrm{seq}(\mathrm{seq}(X))$. $\frown / \langle\rangle = \langle\rangle$.

$A \subseteq B$ $\Leftrightarrow \exists\, C : \mathrm{seq}_\infty X \bullet A \frown C = B$

A is a prefix of B. (This is just '$\subseteq$' on the sets representing the sequences.)

$\mathrm{squash}\, f$ Convert a finite function, $f : \mathbb{N} \nrightarrow X$, into a sequence by squashing its domain. That is, $\mathrm{squash}\{\} = \langle\rangle$, and if $f \neq \{\}$ then $\mathrm{squash}\, f = \langle f(i) \rangle \frown \mathrm{squash}(\{i\} \lhd f)$, where $i = min(\mathrm{dom}\, f)$. For example, $\mathrm{squash}\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C \rangle$.

$A \upharpoonright T$ $\quad == \mathrm{squash}(A \rhd T)$

Restrict the range of the sequence A to the set T.

Axiomatic definitions

Let *D* be a list of declarations and *P* a predicate.

The following axiomatic definition introduces the variables in *D* with the types as declared in *D*. These variables must satisfy the predicate *P*. The scope of the variables is the whole specification.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

Generic definitions

Let *D* be a list of declarations, *P* a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$\begin{array}{|l} [X_1, X_2, \ldots X_n] \\ D \\ \hline P \\ \hline \end{array}$$

The declared variables must be uniquely defined by the predicate P.

Free types

$$X ::= ident1 \mid ident2 \langle\langle S \rangle\rangle$$

Free types allow a new free set *X* to be introduced as well as defining constructors to generate elements of the type. The constructors may either be an identifier (*ident1*) which is an element of the new type, or a constructor function (*ident2*) which is a function taking an argument of type *S* and returning an element of the new type. Distinct values of arguments to constructor functions

return distinct elements of the free type, and distinct constructors generate distinct elements. The constructors generate all the elements of the type.

Schema Notation

Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$
\begin{array}{|l}
\hline
S \\
x : \mathbb{N} \\
y : \mathrm{seq}\,\mathbb{N} \\
\hline
x \leq \#y \\
\hline
\end{array}
$$

and horizontally, for the same example,

$$S == [x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{N} \mid x \leq \#y]$$

Schemas can be used in signatures after $\forall$, λ, $\{...\}$, etc.:

$$(\forall\, S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{N} \mid x \leq \#y \bullet y \neq \langle\rangle)$$

$\{S\}$ Stands for the set of objects described by schema S. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$.

Schema operators

Let S be defined as above and $w : S$.

$w.x$ $== (\lambda\, S \bullet x)(w)$

Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \#w.y$' holds.

θS — The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y.

Compatibility — Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion — A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must have the same declared sets) and the predicates of S and T are conjoined. For example,

```
┌─ T ──────────────────────────────
│ S
│ z : ℕ
├──────────
│ z < x
└──────────────────────────────────
```

is equivalent to

```
┌─ T ──────────────────────────────
│ x, z : ℕ
│ y : seq ℕ
├──────────
│ x ≤ #y ∧ z < x
└──────────────────────────────────
```

The included schema (*S*) may not refer to global variables that have the same name as one of the declared variables of the including schema (*T*).

Decoration — Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example, S' is $[x' : \mathbb{N};\ y' : \mathrm{seq}\,\mathbb{N} \mid x' \leq \#y']$.

$\neg\, S$ — The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{N} \mid \neg\,(x \leq \#y)]$.

$S \wedge T$ The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above).

Given $T == [x : \mathbb{N};\ z : \mathbb{P}\,\mathbb{N} \mid x \in z]$, $S \wedge T$ is

$$\begin{array}{|l}
\hline
S \wedge T \\
x : \mathbb{N} \\
y : \mathsf{seq}\,\mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \wedge x \in z \\
\hline
\end{array}$$

$S \vee T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \vee T$ is

$$\begin{array}{|l}
\hline
S \vee T \\
x : \mathbb{N} \\
y : \mathsf{seq}\,\mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \vee x \in z \\
\hline
\end{array}$$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Rightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$$\begin{array}{|l}
\hline
S \Rightarrow T \\
x : \mathbb{N} \\
y : \mathsf{seq}\,\mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \Rightarrow x \in z \\
\hline
\end{array}$$

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Leftrightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$$
\begin{array}{|l}
\hline
S \Leftrightarrow T \\
x : \mathbb{N} \\
y : \mathsf{seq}\,\mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \Leftrightarrow x \in z \\
\hline
\end{array}
$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The parantheses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

$$
\begin{array}{|l}
\hline
(S \wedge T) \upharpoonright (x, y) \\
x : \mathbb{N} \\
y : \mathsf{seq}\,\mathbb{N} \\
\hline
(\exists\, z : \mathbb{P}\,\mathbb{N} \bullet \\
\qquad x \leq \#y \wedge x \in z) \\
\hline
\end{array}
$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists\, D \bullet S$

Existential quantification of a schema.

The variables declared in the schema *S* that also appear in the declarations *D* are removed from the declarations of *S*. The predicate of *S* is existentially quantified over *D*. For example, $\exists\, x : \mathbb{N} \bullet S$ is the following schema.

$$\exists\, x : \mathbb{N} \bullet S$$

$$y : \mathrm{seq}\, \mathbb{N}$$

$$\exists\, x : \mathbb{N} \bullet$$
$$\quad x \leq \# y$$

The declarations may include schemas. For example,

$$\exists\, S \bullet T$$

$$z : \mathbb{N}$$

$$\exists\, S \bullet$$
$$\quad x \leq \# y \land z < x$$

$\forall\, D \bullet S$ Universal quantification of a schema.

The variables declared in the schema *S* that also appear in the declarations *D* are removed from the declarations of *S*. The predicate of *S* is universally quantified over *D*. For example, $\forall\, x : \mathbb{N} \bullet S$ is the following schema.

$$\forall\, x : \mathbb{N} \bullet S$$

$$y : \mathrm{seq}\, \mathbb{N}$$

$$\forall\, x : \mathbb{N} \bullet$$
$$\quad x \leq \# y$$

The declarations may include schemas. For example,

$$\forall\, S \bullet T$$

$$z : \mathbb{N}$$

$$\forall\, S \bullet$$
$$\quad x \leq \# y \land z < x$$

Operation schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

undashed state before the operation,

dashed state after the operation,

ending in "?" inputs to (arguments for) the operation, and

ending in "!" outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS $\quad \widehat{=}\ S \wedge S'$

Change of state schema: this is a default definition for ΔS. In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\quad \widehat{=}\ [\Delta S \mid \theta S' = \theta S]$

No change of state schema.

Operation schema operators

pre S Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

$$\begin{array}{|l}
\hline
S \\
x?, s, s', y! : \mathbb{N} \\
\hline
s' = s - x? \wedge y! = s' \\
\hline
\end{array}$$

pre S is,

$$\begin{array}{|l}
\hline
\text{pre } S \\
x?, s : \mathbb{N} \\
\hline
\exists\, s', y! : \mathbb{N} \bullet \\
\qquad s' = s - x? \wedge y! = s' \\
\hline
\end{array}$$

$S;\ T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then

the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S;\ T$ is,

$$
\begin{array}{|l}
\hline
S;\ T \\
x?, s, s', y! : \mathbb{N} \\
\hline
(\exists\, ss : \mathbb{N} \bullet \\
\qquad ss = s - x? \wedge y! = ss \\
\qquad \wedge\ ss \leq x? \wedge s' = ss + x?) \\
\hline
\end{array}
$$

Appendix B

Syntax of Live Sequence Chart

$< LSCSpec >$ $::= lscspec < ChartDefList >< InstVariList > endlscspec$

An LSC specification contains a set of charts and a list of variables.

$< ChartDefList >$

$::=< ChartDef >; < ChartDefList >|$

$< ChartDef >$ $::=< ExtChartDef >|< UnvChartDef >$

A chart is a universal one or an existential one.

$< ExtChartDef >$

$::= extchart < LSCName >< InstDefList > endextchart$

An existential chart is identified with its name and made up of a set of instances.

$< UnvChartDef >$

$::= unvchart$

$< LSCName >< PrechartDef >< InstDefList >$

$endunvchart$

A universal chart is preceded with a pre-chart.

$< PrechartDef >$

$::= prechart < InstDefList > endprechart$

A pre-chart contains a set of instances.

$< InstDefList >$ $::=< InstDef >; < InstDefList >|$

$< InstDef >$ $::= instance < InstName >< LocationDefList > endinstance$

An instance has a name and is made of a sequence of locations.

$< LocationDefList >$

$::=< LocationDef >; < LocationDefList >|$

$< LocationDef >$

$::= hotlocation < HotLocationDef > endhotlocation \mid$

$coldlocation < ColdLocationDef > endcoldlocation \mid$

$subchart < Subchart > endsubchart$

A location is either a hot one or a cold one or a compositional one.

$< HotLocationDef >$

$::=< EventDef >|< CoregionDef >|< ConditionDef >$

A hot location may be labeled with an event, a condition or a coregion.

$< ColdLocationDef >$

$::=< EventDef >|< CoregionDef >|< ConditionDef >$

$< SubchartDef >$

$::=< LocationDefList >$

A sub-chart contains a sequence of locations.

$< CoregionDef >$

$::= coregion < EventDefList > endcoregion$

A coregion may contain multiple events.

$< EventDefList >$

$::=< EventDef >; < EventDefList >|$

$< ConditionDef >$

$::= hotcondition < Condition > endhotcondition \mid$
$coldcondition < Condition > endcoldcondition$

$< EventDef >$ $::=< ActionDef >|< MessageDef >|< TimerEventDef >$

An event labeled with a location is either an local action or a message or a timer event.

$< ActionDef >$ $::= action < Action > endaction$

$< MessageDef >$ $::= hotmessage < HotMessageDef > endhotmessage \mid$
$coldmessage < ColdMessageDef > endcoldmessage$

A message can be either hot or cold.

$< TimerEventDef >$

$::=< SetTimerDef >|< TimeOutDef >|< EndTimerDef >$

A timer event is either a set timer event or a time out or an end timer event.

$< HotMessageDef >$

$::=< InputDef >|< OutputDef >$

A message event is either an input or output.

$< ColdMessageDef >$

$::=< InputDef >|< OutputDef >$

$< SetTimerDef >$

$::= settimer < Clock >< Duration > endsettimer$

$< TimeOutDef >$

$::= timeout < Clock > endtimeout$

$< EndTimerDef >$

$::= endtimer < Clock > endendtimer$

$< InputDef >$ $::= input < Message > from < InstID > endinput$

$< OutputDef >$ $::= output < Message > to < InstID > endoutput$

$< InstID >$ $::=< InstName >| env$

$< InstVarList >$ $::= instvari < InstName >< VarList > endinstvari$

$< VarList >$ $::=< VarDef >; < VarList >|$

$< VarDef >$ $::= vari < Variable >< TypeDef > endvari$

$< Action >$ $::= setstate < Variable >< Value > endsetstate$