UNIVERSITY OF SOUTHAMPTON

SPECIFYING, REFINING AND VERIFYING REACTIVE SYSTEM DESIGN WITH UML AND CSP

By
Muan Yong Ng
M.Sc.,B.Sc.(Hons)

A thesis submitted for the degree of
Doctor of Philosophy

Faculty of Engineering,
Department of Electronics and Computer Science,
University of Southampton,
United Kingdom.

March 1, 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING

ELECTRONICS AND COMPUTER SCIENCE DEPARTMENT

Doctor of Philosophy

SPECIFYING,VERIFYING AND REFINING REACTIVE SYSTEM DESIGN WITH UML AND CSP

by Muan Yong Ng

The strength of Formal Methods (FMs) lies in having a mathematical framework which supports a formal and logical approach towards specifying and verifying a system. However, the formal mathematical framework which serves as the selling point for FMs is at the same time an offset because it requires certain degrees of expertise and familiarity in order to use FMs. For many years, many practitioners have been reluctant to adopt FMs in their software development process simply because they are being put off by the steep learning curve and the complicated mathematical theories involved. With this reason in mind, we set off to find ways to improve the use of FMs and in this, we concentrate our effort in seeking ways to combine FMs with the intuitive graphical modelling language in order to reap the potentials offered by both.

In this thesis, we have developed a lightweight approach which uses UML to visualize the syntactical behaviour of CSP. We have devised a way of mapping from UML to CSP and used UML as an entry point for system designers who wish to utilize CSP in their design. The results is encouraging in that we allow practitioners to use CSP without having to write the CSP themselves. We feel that this is a great step forward for system designers who are generally not familiar with Formal Methods but would wish to exploit the full advantage of using Formal Methods. Furthermore, we have also developed a formal semantics model which defines the behaviour of UML state diagrams in CSP. The model is crucial for it provides us with a set of *unified* semantics to work on when we design a system using the UML state diagrams. Our work enables practitioners to design in UML based on a set of unified semantics and later use CSP to formally check the correctness of their design. Lastly, we have developed a prototype tool which automatically takes UML diagrams as input and generate CSP that can be fed directly into FDR for model-checking.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I would like to thank my supervisor Michael Butler who has introduced me to the subject and guided me throughout until the completion of this thesis. You have been a superb supervisor in providing great suggestions and guidance, and at the same time giving me the freedom and supports to express my own ideas. I would also like to thank my external examiner, Helen Treharne who has done a superb job in examining my work and providing valuable feedbacks in improving this thesis.

A big thank you to my parents and family, who are always behind me during all these years when I am far from home. I will not have gone this far without your supports and loves. Thanks to my twin sister too, Muan Hong, for putting me up in every situation, you have been great! A big thank you also to all of you in the Southampton Chinese Christian Church, I truly value your prayers, friendships and loves. Thank you to Yue Teng, who has been my main confidant and support especially when things are not going well. Also, a big thank you to all my collegues in DSSE, who have made my stay in ECS a pleasant and memorable one.

Last but most importantly, I would like to thank my God Jesus Christ, who has blessed me graciously since I embarked on this work, and given me the right focus to complete this thesis.

List Of Symbols

S_M	The set of state identifiers found in state machine M.
$S_{M(ss)}$	The set of simple state identifiers found in state machine M.
$S_{M(cs)}$	The set of composite state identifiers found in state machine M.
$S_{M(is)}$	The set of initial state identifiers found in state machine M.
$S_{M(fs)}$	The set of final state identifiers found in state machine M.
$S_{M(choice)}$	The set of choice state identifiers found in state machine M.
$S_{M(region)}$	The set of subregions found in state machine M.
$S_{M(o)}$	The top state of state machine M.
$S_{M(cos)}$	The set of composite-OR-state identifiers found in state machine M.
$S_{M(cas)}$	The set of composite-AND-state identifiers found in state machine M.
T_M	The set of outgoing transition identifiers found in state machine/state M.
$T_{M(exp)}$	The set of explicitly triggered outgoing transitions for state machine/state M.
$T_{M(imp)}$	The set of implicitly triggered outgoing transitions for state machine/state M.
E_M	The set of event identifiers found in state machine M.
$E_{M(exp)}$	The set of explicit event identifiers found in state machine M.
$E_{M(imp)}$	The set of implicit event identifiers found in state machine M.
$\mathcal{A}_M$	The set of action identifiers found in state machine M.
$IMM(S)$	The immediate enclosing state for state S.
$ENCL(S)$	The set of enclosing states for state S.
$label(S)$	The label for state S.
$entry(S)$	The state entry action for state S.
$exit(S)$	The state exit action for state S.
$doActivity(S)$	The do-activity for state S.
$source(T)$	The source state for transition T.
$target(T)$	The target state for transition T.
$event(T)$	The trigger event for transition T.
$guard(T)$	The boolean guard expression for transition T.
$action(T)$	A sequence of actions for transition T.

Chapter 1

Introduction

1.1 Background

A software process is a set of activities which leads to the production of a software product [68]. There are different software process models to cater for different system engineering needs. Among the more common models are: the Waterfall Model [68], the Evolutionary Development [68], the Reused-Based Development [68] and the Formal System Development [68]. Our work seeks to concentrate on the last model. The Formal System Development is based on the formal mathematical transformation process of a system specification to an executable program. The system requirement specification is **specified** in a formal specification which is expressed in a mathematical notation. The formal specification is then **refined**, through a series of transformation into a program. In the transformation process, the formal mathematical representation is systematically converted into a more detailed system representation, and each refinement is **verified** to ensure the newly refined representation still satisfies the requirements stated in the former representation. The transformation process will continue until the formal specification is converted into an equivalent program. The main advantage of this approach is that it renders an incremental step from specification to implementation, therefore increases the accuracy of the final product in satisfying the requirements stated in the specification.

In general, there are four fundamental activities which are defined in the Formal System Development. There are

1. Software Specification - which defines the requirement, functionality and constraint of the software.
2. Software Design and Implementation - which produces the software that meets the specification using a chosen implementation language.
3. Software Validation - which validates the software to ensure that it meets the customer requirement.
4. Software Evolution - which enhances the software to accommodate for the changing customer requirements

Our work deals mainly with the software specification and design stage of the Formal System Development cycle.

1.2 Motivations

One of the key issues we need to address in software engineering is the correctness of use for software systems in the safety critical situation. One only has to consider the risks inherent in the use of software to control nuclear power stations, chemical plants, aircraft and so on to recognize the need to be able to check and certify that the software is reliable. Perhaps there is nothing formal that can be done to prove that the specification for a program is correct. However, in theory, it is now possible to provide formal proof that an implementation of a specification in the form of software meets the specification, if the specification is drawn up in a fully logical and formal way. This is where Formal Methods [11, 79] comes into picture.

Based on the definition given by Formal Method Europe [1], Formal Methods (FMs) are *mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.* FMs have precise notations and semantics which can be used to express system requirements and specification (what a system should do) in an exact and unambiguous manner. Specifying the system properties using FMs very often helps designers to uncover many implicit aspects in the stated requirements at an early stage in the design cycle. This greatly enhances the understanding of a system and even contributes towards significant time and cost saving in producing more accurate software.

Each FM is supported by a specification language with which a system specification can be described formally. Examples of the specification languages include the B method [2, 34, 61], Z-notation [40, 69] and **CSP** [31, 59, 60]. Each specification language has a unique set of notation and mathematical paradigm to work with and they may be supported by tools such as animator, model checker and theorem prover. Instances of these are Atelier-B [16] and the B-Toolkit [36] for B , ZANS [58] for Z and FDR [75] for CSP.

Adopting FMs in the system development requires knowledge of mathematical model notations, understanding of the underlying principles and having good experience will be an added advantage. However, all these demand proper training and significant investment of time to get familiar with a method. Due to lack of resources and support, the industry is often discouraged from adopting the use of FMs in their system development. More often, the use of FMs is restricted within the context of academic and research purposes.

With these reasons in mind, we set out to look at ways to improve the use of FMs in the software system development, especially in the industry. Along this, we observe that

[1]Formal Method Europe is an independent body made up of different individuals, academic and government bodies which aim to promote and support the industrial use of FMs for computer system development.

there is a need to make FMs more accessible to their users, especially to those who are already working in the development cycles but are often FMs novices.

1.3 Outline of Our Work

In order to make FMs more accessible to their potential users, we began our work by proposing the idea of using graphical notations as the front end for system engineers to utilize FMs in designing and verifying the correctness of their work.

We have restricted our work by just looking at one of the FMs. For this, we choose to work with CSP (Communicating Sequential Processes). CSP is a specification language that is used to describe concurrent systems whose components interact with one another and also with other components from other systems. CSP provides a useful way to reason about and design parallel systems which are traditionally seen as complex and trouble-prone. Also, the fact that CSP is well-supported by model checkers such as FDR and animators such as ProBE [44] have further elevated the status of CSP from being a white board language to a concrete language and this enhances its potential in dealing with real problems.

In order to make CSP more appealing to the system designers, we propose to use **UML**(Unified Modeling Language) [3, 4, 5, 17, 24, 57] as a graphical front end that allows system designers to use the graphical notations for design, and subsequently, use CSP to verify the correctness of the design. In achieving this, we have formalized a mapping strategy that allows CSP to be represented in UML. We choose to use UML because it has been proposed by OMG [2] as the standard modelling language for the industry and UML has received increasing attention from the industry in recent years.

In addition, we have also developed an automated tool that will take in a design in UML and generate the corresponding CSP specification. The automated tool will not only provide a transparent platform for system designers to work indirectly with CSP, more importantly, the tool serves as a ground for us to experiment with the proposed mapping strategy in a consistent manner.

The work we have achieved in visualizing CSP in UML has opened the possibility for us to match UML constructs with those of CSP. The discrepancy of UML is that it is a semi-formal language which has an extensive set of constructs with good structural semantics but lacks of a formal behavioural semantics. For this reason, the second part of this thesis is devoted to developing a formal semantics for UML State Diagrams by

[2]The Object Management Group, Inc (OMG) is an international organization comprises system vendors, software developers and users that set out to establish the industry guidelines and object management specifications to provide a common framework for the application of object-oriented technology in software development. In doing so, OMG has adopted the UML specification as the standard modelling language for the industry in order to reduce any confusion over many modelling notation. OMG has also resumed the responsibility of pursuing further the development of UML standard and produced [54].

inferring from the OMG informal semantics and Harel's semantics [22, 27, 28, 29, 30] on statecharts. We then express the derived semantics in CSP.

To restrict the scope of our work, we only limit ourselves to considering **reactive systems**. According to Harel [30], the typical characteristics of a reactive system are:

- It usually has more than one process interacting with one another in parallel.
- Its operations and responses to inputs are often time-critical (The issue of time-critical will not be addressed in this phd work).
- It can interact with the environment via its inputs and outputs. These inputs and outputs could occur at any point of time and they are often asynchronous with the running process.
- It will respond to any interrupt which is regarded as a high-priority event at any point of time even when the system is busy.
- There could be different behavioural outcomes from a system and these depend on the system past history, the current input values and also the current operation mode.

Examples of reactive systems include a telephone, a lift system, a barrier control system, an avionics systems, a VLSI circuits and the machine interface to typical desktop software.

1.4 CSP

1.4.1 Syntax and Notations

CSP is a notation for describing concurrent systems whose components, which are called **processes**, interact with each other and with the environment by communication.

A process is defined in terms of **events**, which are the basic elements of CSP. An event may be initiated either by the process itself or by some agents external to the process. These external agents include other processes in the system with which the system interacts. In other words, events are interfaces through which a process interacts with its environment. The occurrence of an event is assumed to be instantaneous. More than one process may be involved in the performance of an event. When this happens, the event will only take place when all its participants are ready to execute. Processes may be indexed to allow parameterized definitions while identifiers may be introduced into the system via input attached to an event, in a manner to be described next.

A single event may contain more than one piece of information. The information can be the kind of event they are, the entity the events are concerned with, the communication channel they are on, or the message they carry. In this case, we call the event together with all its information a **compound event**. A dot operator "." separates each piece of information in a compound event. For example, *gate.open* is a compound event with *gate* being the entity and *open* being the kind of action associated with the entity. A family of compound events makes up a **channel**, for instance *gate.open* and *gate.close* can be grouped under a channel such as *channel gate: open | close*. On the other hand, we could also use compound events to model the input(?) and output(!) mode of a communication. For instance, $a?x : T$ means channel a is inputting an element x of type T while $b!x : T$ means channel b is outputting an element x of type T.

If a is an event and P is a process, $a \rightarrow P$ is a process that is initially ready to engage in a and when a occurs, the process will subsequently behave as P. ***STOP*** is the simplest process in CSP which does nothing. It is frequently used to represent a deadlock in a system. ***SKIP***, on the other hand, denotes successful termination and it is identified as $\surd \rightarrow STOP$. $\surd$ is a special event in CSP that represents the act of terminating successfully.

More than one process may be synchronized to execute the same event, and we use a parallel operator to represent this. The **parallel** composition, e.g. $P \underset{\{a\}}{\parallel} Q$ shows that processes P and Q are executed in parallel, that is they synchronize over event a and interleave in executing all other events. **Interleaving**(|||) is a special case of parallel composition where $P \mid\mid\mid Q$ means P interleaves with Q and there is no synchronization between their events.

Sequential composition is a different process combination, whereby $P; Q$ means once P has completed, the system control is passed on to Q. Hence, the execution of P and Q is in sequence, starting with P and followed by Q. $B \& P$ is a **guarded** expression where B is a Boolean guard such that process P will only be executed provided B is true.

There are two types of choice in CSP: deterministic choice($\Box$) which is resolved by the environment and nondeterministic choice($\sqcap$) which is decided by the process itself. If P and Q are processes and a and b are events, for a process $a \rightarrow P \Box b \rightarrow Q$, the environment may choose to engage in either a or b, which then causes the subsequent process to behave either as P or Q respectively. On the other hand, for a process $P \sqcap Q$, the process will choose internally whether to behave as P or as Q. In this case, the choice belongs to the process itself and the environment has no control over it.

Process hiding e.g. $P \backslash S$, allows a process to behave like P except all the events in set S are removed from its interface and become internal to the process. Consequently, the process will have no synchronization with other components over all the events in S.

CSP has a time-out construct, denoted as $\triangleright$. In the context of untimed CSP where time is not explicitly modelled, given a process $(x \rightarrow A) \triangleright (y \rightarrow B)$ with x and y being the CSP events and A and B being the CSP processes, this means that if x is not offered at all, y will be offered eventually [3]. The interrupt operator, denoted as $\triangle$, on the other hand, may be used as the name suggests to interrupt a process that is going on.

Finally, CSP allows generalisation of the binary operators over indexed sets of processes, e.g.

- $||\ x : X@P(x)$ is an **indexed parallel** composition of all processes $P(x)$
- $|||\ y : Y@P(y)$ is an **indexed interleaving** composition of all processes $P(y)$
- $\Box\ z : Z@P(z)$ is an **indexed external choice** which means one process P(z) may be chosen out of the range Z of process P by the environment.
- $\sqcap\ z : Z@P(z)$ is an **indexed internal choice** which means one process P(z) may be chosen out of the range Z of process P by the system internally.

1.4.2 Behavioural Semantics

CSP is a notation and calculus that assists us in understanding the interaction between components of a concurrent system. The behaviour of a CSP process is usually defined in terms of its traces, failures and divergences [59].

A trace of the behaviour of a process is a sequence of events which the process performs and they are recorded in the order of their occurrence. There are different traces representing the different behaviour of a system, where the behaviour differs according to its interaction with its external environment. A function *traces(P)* is used to define a set of all possible traces for process P. For example, if $P = a \rightarrow b \rightarrow STOP \ \Box \ c \rightarrow d \rightarrow STOP$, $traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c\rangle, \langle c, d\rangle\}$. The traces in the set tells us the progress of P and also the different possibilities of the behaviour of P, i.e. P might start by doing an "a" followed by "b", or it might choose to do "c" then followed by "d". The set of traces is prefix closed.

A failure for a process is a pair of (t, R) whereby t is a trace being observed after which all the events in the set R may be refused by the process, even if offered by the environment. The set R is called a *refusal.* Suppose we have a deterministic process $P = a \rightarrow STOP \ \Box \ b \rightarrow STOP$ and a nondeterministic process $Q = a \rightarrow STOP \ \sqcap \ b \rightarrow STOP$. P and Q share the same set of traces, i.e. $\{\ \langle\rangle,\ \langle a\rangle,\ \langle b\rangle\}$. However their failure behaviour are different in that at the initial stage of the process P, P is always willing to engage in either a or b (depending which is being offered by the environment) ; whereas Q may initially refuse to do either a or b (as a result of some internal nondeterministic choice). Hence, their failure sets differ from each other where

$$failures(P) = \{(\langle\rangle, \{\}), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$$

$$failures(Q) = \{(\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$$

[3]In the timed CSP model whereby time is modelled explicitly, if x does not occur after a defined unit of time, y will be executed.

From this, we can see that failures model is useful in identifying if a process is deterministic. As demonstrated by the example, a process is said to be deterministic if it can *never* refuse any event which is being offered at each point of the process.

For the divergence behaviour of a process, a divergence component is a set of traces after which the process becomes livelocked. Divergence behaviour is normally encountered when hiding is used where a process may perform infinitely many hidden events and the environment has no way of interrupting the process. It is assumed that once a process can diverge, it can then perform any trace or refuse anything and can always diverge on any later trace. Therefore, the function *divergences(P)* contains not only the traces s on which P can diverge, but also all extensions $s^\wedge t$ of such traces. To observe accurately what a process can do after it has already been able to diverge is difficult and it is not worth the effort. So, in general, divergence is undesirable and hence need to be identified and removed from a design. This is where the divergence model is useful.

1.4.3 Refinement Notions

In general, refinement is a process whereby several levels of specifications are produced, with each specification being derived from the specification before and each specification fulfills the properties of its predecessor. The goal of refinement is that the lowest level specification will possess the structures that closely reflect the implementation.

In CSP, refinement is a relationship between two processes such that if the behaviour of B is a subset of the behaviour of A (i.e. B satisfies the behaviour properties of A), then we say B is a refinement of A and this is expressed as $A \sqsubseteq B$ (pronounced A refined by B).

There are three levels of refinement in CSP: traces, failures and failures/divergences refinement. Given processes A and B, if $traces(B) \subseteq traces(A)$, then we say B is a traces refinement of A ($A \sqsubseteq_T B$). If $failures(B) \subseteq failures(A)$ then we can say B is a failures refinement of A ($A \sqsubseteq_F B$). Similarly, if $divergences(B) \subseteq divergences(A)$ and $failures(B) \subseteq failures(A)$, then $A \sqsubseteq_{FD} B$ such that B is a failures-divergences refinement of A.

Therefore, if $A \sqsubseteq_{FD} B$ is true, this implies that $A \sqsubseteq_F B$ is also true. Since $A \sqsubseteq_F B$ suggests that $failures(B) \subseteq failures(A)$ which in turns implies that $traces(B) \subseteq traces(A)$, from here, we may also deduce that $A \sqsubseteq_T B$ is true.

1.4.4 Tool Support

The fact that CSP is supported by tools has greatly enhanced its potential in solving industrial problems. FDR (stands for *Failure Divergence Refinement*) is the first commercially available tool for model-checking CSP. It carries out two types of check, one on refinement assertions and another on individual processes.

We start by looking at the refinement assertion check. When FDR checks for the refinement assertion between two processes, it bases its check on one of the three behaviour models: traces model, failures model or failures-divergences model. For the traces model, FDR will check for the refinement based on the trace property such that if process B refines A, it ensures that all the possible sequences of communication which B can do are also possible for A. Hence, if we consider A as a specification that determines all the possible safe traces of a system, when $A \sqsubseteq_T B$ is satisfied, this proves that B is a safe implementation, with no wrong events possible. Therefore, traces refinement may be used when we need to determine the safety property of a system. The failures model, on the other hand, allows us to make better distinction between processes based on their permitted executed traces and the corresponding refusals. Under the failures refinement check, if B refuses some events, A should also be able to refuse them (after performing some events). FDR will identify an error as a deadlock if B refuses some events which are not possibly refused by A after performing a similar trace. The failures-divergences model provides further strength as compared to the failures model. Besides checking for deadlock, the failures-divergences model can also be used to analyze a system which has the potential of never executing a visible event. A failures-divergences refinement check fails if the design model contains a livelock which is not possible in the specification model.

In essence, FDR allows a concrete design description to be compared with an abstract specification in order to check if the refinement properties (mentioned above) are satisfied. If the properties are not satisfied (thus the refinement check fails), FDR will generate counter examples that could be used to pin point the failure.

Apart from checking refinement assertions, FDR may also be used to perform checks on individual processes. Three types of behavioural properties could be checked for a single process : deadlock, livelock and determinism. Determinism check is meant for processes which contain internal choices. The check detects if the processes behave in an nondeterminism manner since the external environment has no control over the internal choices.

ProBE is an animator for CSP which has received considerable attention. In contrast to FDR's automatic checking of properties, ProBE is an interactive animator which allows users to control the resolution of non-determinism and the choice of actions, hence enables users to watch a process evolves in response. Both FDR and ProBE are products of Formal Systems [75]. Our work adopts the use of FDR, both for the availability and also for the ease of use.

1.5 UML

1.5.1 Overview

UML (stands for Unified Modeling Language) is a graphical modelling language comprises a collection of graphical notations(diagrams) illustrating different aspect of a software system. It has an extensively structured set of constructs with a structural semantics but lacks a comprehensive behavioural semantics, that is to say there is no formal definitions for how a diagram which is made up of several constructs may behave. Since it has a formally defined syntax and structural semantics but an informally specified dynamic semantics, it is also termed as a semi-formal language.

The methodology of UML is that it is object-oriented and it promotes an iterative and incremental design process. By using an iterative and incremental approach, we can better manage the complexity of a system as well as incorporate requirements and technologies changes as the design evolves over time. Beyond this, UML offers different diagrams to model different aspects of a system. These diagrams are categorized into a few groups according to their functionality, as follows:

- *Use Case View.*
  The use case view describes the different functionalities of a program. It is generally used to capture the basic requirements of a system and to provide the basis for the construction of other views. A **use case diagram** may be used in this view to depict the functionality of a system.
- *Static Model View.*
  It is also called a structural or design view. This view describes the logical structures which support the functional requirements expressed in the use case view. It contains the program components which are principally classes and describes the functionality for each components. This view is particular useful when we need to gain an overall picture of how a system is made up of. A **class diagram** depicts the static structure of a system using relationships between classes and general concepts such as class attributes and operations.
- *Dynamic View.*
  This view illustrates the behavioural aspect of a system in terms of its executable threads and processes. There are a few diagrams available, as follows:

- A **sequence diagram** consists of actors, messages and a timeline. It is used to show interactions between actors/objects through sending and receiving of messages arranged in a time sequence.
- A **collaboration diagram** is similar to a sequence diagram except it has sequence numbers to replace the timeline. A collaboration diagram is useful to show the actual objects involved and the structural relationships between them but it is weaker when it comes to showing the interactions between those objects as a time-ordered sequence of events.

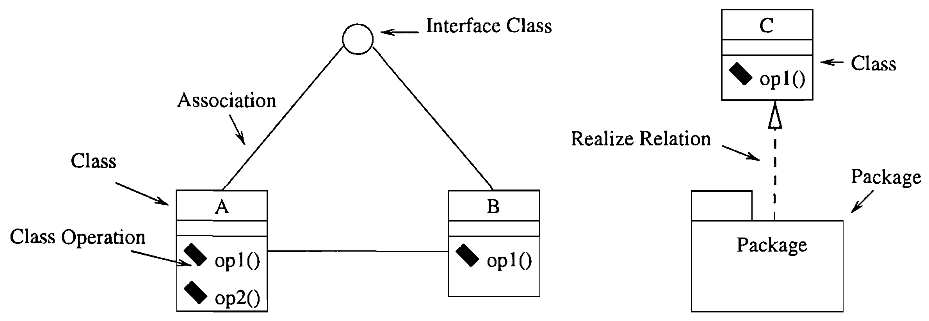


Figure 1.1: Elements in a Class Diagram

- A **state diagram** comprises states and events that show how a system changes from one state to another via different response to the current status condition and also events being offered by the environment.
- An **activity diagram** is similar to a state diagram except it is activity-oriented rather than state and event-oriented.

- *Component View.*

  A **component diagram** illustrates how the different components in a system are connected. A component is a physical and replaceble part of a system that could be representing a source file, an activeX control, a Java servlets and so on. A component comprises many classes and interfaces which show how each component is related to one another in the system.

- *Deployment View.*

  A **deployment diagram** shows the physical hardware (such as a PC) on which the software system will execute, and how the software is deployed on the hardware. It consists of nodes which each of them represents a physical hardware. Each node contains components representing the software system residing in the physical node. The relationship between two nodes shows that there is a connection between the two nodes.

Among all the UML diagrams being offered, we only consider state diagrams and class diagrams in this work. We will further elaborate on these two diagrams in the following sections.

1.5.2 Class Diagram

In the UML context, class diagrams are used to describe the types of objects in the system and the various kinds of static relationships that exist among them. Figure 1.1 shows the the various graphical representations for different elements in a class diagram which are used in this work. The main entity in a class diagram is the **class**. Each class has its own attributes and operations. An **interface class** is a variation of the normal class. It is a class that acts as a template for other classes and no instances of it can be created. **Package**, on the other hand, is a general purpose model element that organizes other elements such as classes into a group.

The principal kind of static relationship between classes is the **association**. Each association has two ends with each end being attached to one of the classes in the association. Another type of relationship that exists between two entities is the **realize relation**. It is a dotted arrow line showing a class realizing the operations offered by the other entity (pointed by the arrow head).

1.5.3 State Diagram

Basic Feature

A UML state diagram is used to describe a system behaviour in terms of its events and state changes. Its notation and semantics are substantially those of Harel's Statechart [22, 27, 28, 29, 30] except it is an object-based variant of Harel's. A UML state diagram specifies the states a system may reside in and the transitions from one state to another. In addition, it also specifies what causes activities to start and stop, and how the system responds to various trigger events. Based on the informal behavioural semantics defined by OMG [54], an event is an observable occurrence that may be generated by the system itself by doing an action or by the environment surrounding the system. A state diagram contains exactly one state machine that describes an object, which could be a class, a use case, a subsystem or the entire system.

The semantics of state diagrams specifies that an object being modelled is always in one of the finite set of states when it is in sequential operation, or it can occupy simultaneously several states within a composite state when it is in concurrent operation. When the object receives an event, it will response by moving from the current state to another state. We may have actions attached to a transition or nested within a state. The OMG defined informal semantics does not state clearly the differences between events and actions. However, the Harel's semantics provides some insights where it defines an event being the receiving of a signal or the effect of an operation call, and an action being the sending of a signal or the call of an operation.

The basic syntax of a state diagram consists of rounded rectangles that represent states, filled circles for the initial states, bull's eyes for the final states, diamonds for choice states and finally, the arrow showing the path between states for the transitions. The syntax for a transition label has three parts, all of which are optional: *Event[Guard]/Action*. A *guard* is a logical condition and a guarded transition may only occur if the guard is true.

For example, suppose we have a simple cassette player mechanism that may reside in one of the two states: *PAUSE* and *PLAYING*, as shown in Figure 1.2. The initial start state(filled circle) and the stop state(bull's eye) indicate the start and the end of the system respectively. When state *PAUSE* is active, if event *play* occurs, it triggers the action *turn_on_player* and also a transition that brings the system to state *PLAYING*.

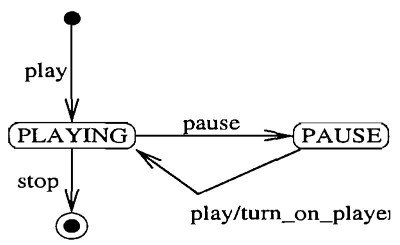


Figure 1.2: A state diagram with two states.

Note that although there are two transitions in the diagram that may be triggered by the event *play*, i.e. one which emanates from the initial state and the other from the *PAUSE* state , only the later transition is triggered because *PAUSE* is active at the time *play* is offered. Only the active state can response to any live event that is offered by the environment. In state *PLAYING*, if event *pause* is offered, it will trigger an opposite transition that brings the system back to the state *PAUSE*, whereas if *stop* is offered, the system will reach the end state and terminate successfully. In this case, we can see that *PLAYING* has two possible outgoing transitions. For a state with more than one outgoing transition, only one transition can be fired at the point of exiting the state.

State

A state is a situation during the lifetime of an object when it waits for some events to take place or it performs some actions/activitities. A state may be passed through instantaneously or not instantaneously. Each state machine has a top state that encloses all the states in the state machine.

A **simple state** is a state which does not have substates whereas a **composite state** is a state that contains other state vertices. These states that are enclosed within a composite state are called the substates of the composite state. There are two types of composite states: OR-state (which is also called a sequential state) and AND-state (which is sometimes called a concurrent state or an orthogonal state). An OR-state is a composite state which contains substates that are OR-ed together such that only one of the substates can be active at one time. On the other hand, an AND-state is a composite state which contains subregions that are AND-ed together, in that when the enclosing AND-state is entered, all the subregions become active at the same time. Figure 1.3 shows an example of an AND-state with two subregions *S1* and *S2*. Each subregion contains states that may not be shared with other subregions. Each subregion must include an initial and an end state. A transition to the AND-state signifies an entry to all the initial states of all the subregions. A transition to the final state of a subregion represents the completion of the activity for the subregion. The activity of an AND-state is assumed to complete only when all the subregions have completed their activities.

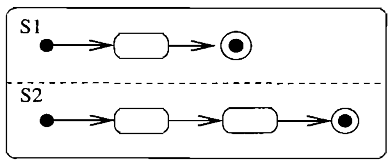


Figure 1.3: An AND-state with subregions S1 and S2.

Pseudostates are transient points in the state machine which are typically used as notational symbol to indicate a special point (such as the initial state) or connect multiple transitions to more complex state transition paths. Because pseudostates are intermediate or transient states between two states, self-transitions (e.g. transitions which originate from and terminate at the same point) are not allowed on pseudostates. Examples of pseudostates are initial states, history states, joins, forks, junctions and choices. Here, we choose to only elaborate on those which are used in our work.

1. **Initial state** represents the source for a single transition to the default/start state of a composite state. The well formedness rules defined by the OMG group [54](p2:157) states that an initial state can have at most one outgoing transition and no incoming transitions.
2. **Final state** cannot have any outgoing transition. A final state within a composite state signifies that the enclosing state is completed. If the enclosing state is a top state, then the final state indicates that the entire state machine has completed.
3. **Choice state** allows a transition path to be split into more than one branch. The choice of which path to take will depend on the trigger event that is offered by the environment and the guard to be satisfied if it is present. Only one path may be activated at one time. In the event where more than one path is enabled, one transition will be chosen, based on a priority rule to be discussed later.

A state can be active or inactive during execution. A state becomes active when it is entered as a result of some transition and it becomes inactive if it is exited as a result of a transition. A state may contain an entry action, an exit action and a do-activity which is made up of a sequence of actions. When a state in entered, the entry action is carried out before any other actions are executed. Conversely, the exit action is carried out prior leaving the state.

The state activity takes place upon the completion of the entry action. The do-activity may be carried out when the state is active. In the situation when the trigger event occurs before the do-activity completes, the activity will be aborted and the exit action takes place prior to the state exit. On the other hand, if the do-activity finishes before any trigger event occurs, the state will raise a *completion event* such that if there is an outgoing transition, the state will be exited.

For a composite state, an incoming transition that terminates on the outside edge of the state indicates the entry to the state. In Figure 1.4, for example, the transition *t1* to

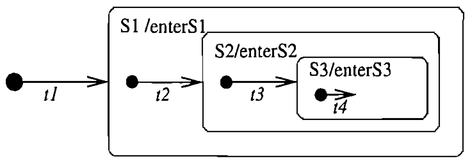


Figure 1.4: Composite state S1 containing substates S2 and S3 and their corresponding entry actions.

the edge of the composite state *S1* indicates an entry into *S1*. The entry action of the composite state i.e. *enterS1* will be carried out before the transition to the default initial substate *S2*. Similarly, after *S2* is entered, *enterS2* is carried out. Each time, the entry action of the substate is executed after the execution of the composite state entry action. This rule is performed recursively until the transition terminates at a direct substate.

Transition

A transition is enabled if and only if

- its source state is the current active state, and
- the event that is being offered matches the trigger for the transition, and
- the conditions for the transition guard (if it exists) is satisfied.

For a simple transition from the source state to the target state, the transition is activated when the trigger event occurs, following which the exit action of the source state will be carried out. In the case where the transition is attached with an action, the transition action is executed before the entry action of the target state is carried out. Looking at Figure 1.5, assuming the system is in *S*, when *ev* occurs, *exitS* is carried out, followed by *a* before *enterT* takes place.

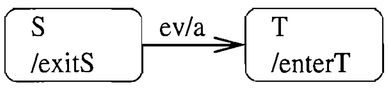


Figure 1.5: State and Transition Actions.

Once the transition originating from the border of a composite state is activated, it will trigger an exit from the composite state and also all the active substates within the composite state. The exit action for all the active substates will be carried out starting with the inner most active substate and finishes with the composite state exit action. In Figure 1.6, for instance, when transition *t1* is triggered, if *S0*, *S1* and *S2* are active, the exit actions are carried out in the following order: *exitS0*, *exitS1*, *exitS2*.

OMG defines some rules to resolve the firing priority of conflicting transitions, i.e. when more than one transition is activated by an event but which only one transition is allowed to occur. An example is shown in Figure 1.7. If the current state is S with C and D being active, if event *a* is offered, a conflict arises of whether transition D-E or S-F

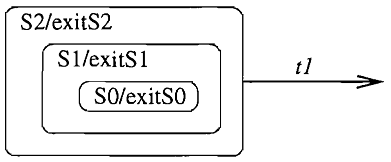


Figure 1.6: The exit action for all the substate will be carried out starting with the inner most substate and finishes with the composite state exit action.

should proceed. OMG defines a transition priority scheme such that the internal transition will always have priority over the higher level transition. In our case, the priority relation resolves to choosing the lower level transition D-E over S-F. However, if event c is offered instead, the conflict between C-H and D-G cannot be resolved in UML since both their source states C and H are in the same level of the state hierarchy.

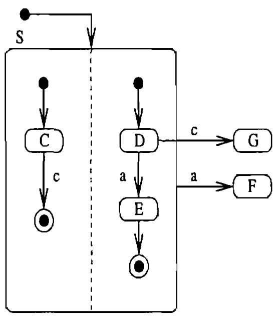


Figure 1.7: An example of conflicting transition.

In a simple transition with a guard, the guard is evaluated before the trigger event occurs. For a multiple transitions such as those originating from a choice point, the order in which the guards are evaluated is undefined.

Event Processing

Before we illustrate how a UML state diagram processes an event, we need to first discuss the Harel's statecharts. Harel first invented statecharts [29, 30] with a vigorious semantics [28] and it is supported by a tool called STATEMATE [22]. As mentioned before, the UML state diagram semantics and notations are substantially those of Harel's statecharts except the former is an object-based variant of the Harel's. In our work, we have chosen to refer to Harel's statecharts semantics in places where the UML informal semantics is found to be lacking.

For event processing, Harel proposes the idea of step execution, where a system executes a *step* when it performs all relevant reactions whose triggers are enabled. The activity being executed within a step is assumed to take zero time. Let us illustrate with some examples. Figure 1.8 shows two cases, each consisting of an execution scenario that is made up of two steps. The first case, Figure 1.8(a), shows that action G is generated as an event when E is offered, and the transition from A to B takes place. The system responds to event G in the next step by making a transition from C to D. Observe that the generation of an event and the response to the event do not happen within the same step. In the second case, Figure 1.8(b), the execution of E triggers a transition from J to K. At this point, state K becomes active. Suppose the condition c is valid at all time, the next step will take the system to L. Again, the generation and consumption of the event E do not happen within the same step.

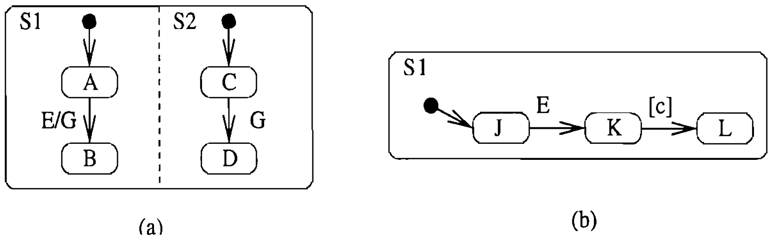


Figure 1.8: Step Execution.

During the execution of a step, the generated event will trigger a valid transition from the current active state and also all the actions associated with the transition. In the case of a concurrent state, it is possible for a single event to fire multiple transitions, but at most one transition is to be fired in each subregion. The step will only complete after all the fired transitions with their respective triggered actions are completed. A new step will commence after the state machine has reached a stable state configuration. A state machine is said to have reached a stable state configuration when it has completed its transition and entered a state which it is residing.

OMG adopts Harel's step execution and complements it by proposing the concept of event queue to fill in the gap left by Harel. In this, they try to explain what happens after an event is generated and before it is being consumed. Under the OMG proposed concept [54](p2:161), a state machine (which represents an object) is assigned with an event queue whereby whenever the environment external to the state machine generates an event, the event will be placed on the queue for further processing. An event instance could be generated by the environment (i.e. an action from users) or by an action executed by another state machine in the system.

At an event queue, the events are taken off the queue in a first in first out (FIFO) manner and processed in steps, as explained by Harel. In this, OMG refers to the Harel's steps as the "run-to-completion (RTC)" steps. A RTC step is initiated when an event is taken from the queue and processed by a state machine one at a time. As such, only one event may be offered to the system at one unit time.

The event which is currently dispatched from the event queue is called the *current event*. The current event will trigger those transitions which source states are the current active states. If no transition is enabled and the event is not in the deferred event list of the current state configuration, the event is discarded and the RTC step is completed. A deferred event list is specified by a state to name a list of events that is to be deferred when the system is at the state. OMG explains that if the current event is found in the list of deferred event of the current state, the event will not be dispatched but instead it will remain in the event queue until the state machine reaches a state where the deferred event triggers a transition, or it is no longer being specified as a deferred event in the current state. However, it is not clear how this is being done, and in what order the deferred event is kept in the queue.

1.6 Related Work

There has been much work going on that involves combining formal and informal methods. We limit ourselves to only look at those which seek to combine graphical notations and formal methods with the aim of reaping the potentials offered by both. In general, we may categorize the existing work into two categories, based on the main objectives of the work.

The first category is made up of those which look at representing FMs using graphical notations. Among those significant are the ongoing work by Snook&Butler [66] in Southampton which uses UML class diagrams to construct B specifications, and Meyer& Souquieres [48] which generate B from OMT diagrams. Wehrheim [78] looks at using UML class digrams to model the system architectural view expressed in CSP-OZ (a combination of CSP and Object-Z), while Fischer et al. [23] proposes using UML-RT (a UML profile for modelling real-time embedded system) to represent CSP-OZ. Work in this category tends to emphasize on providing graphical visualization for the FM notations without adhering strictly to the semantics of the graphical notations being used. In many cases, the graphical notations are found not to be sufficient to express all the information needed in a model. At such, annotations of constraints, variants and operation semantics are added to complete the information in a FM being modelled. The first part of our work presented in Chapter 2 is akin to this nature. Closely related to our work is that of Bolton & Davies [8], Davies & Crichton [15], Brooke&Paige [10], Abeysinghe& Phalp [1] and Engels et al. [19], which seek to represent Hoare's CSP in different graphical notations. A comparison between their work and ours is made in Section 2.8.

The work in the second category is different from the first for it involves more in-depth study which seeks to give a formal meaning to the UML models (we restrict ourself to look at only UML notations). Some of this work makes use of the readily available FMs framework while others define formal semantics that cater for a specific use, all with the common aim of formally reasoning about the behaviour of the UML diagrams. The second part of our work falls into this category and it concentrates on providing a formal semantics in terms of CSP for UML state diagrams (see Chapter 3 & 4).

Among the work featured in this category are those which formalizes the behaviour of UML activity diagrams, such as work by Eshuis&Wieringa [20, 21] which defines a formal execution semantics that allows model-checking and Borger et al. [7] which uses

ASM (Abstract State Machine) [33] semantics. [18] and [25] formalize the UML class diagrams in terms of Z. The work that is of most interest to us is that of UML state diagrams. The related work on this include those of Lilius et al. [41, 42, 43, 56] and Latella et.al [37, 38, 47] which translate UML state diagrams to Promela/SPIN [32] that allow linear temporal logic model-checking. The works in [39, 62, 63] formalizes UML state diagrams in B, [6] in ASM and [73, 74] uses labeled transition systems, but these works do not support formal model-checking. A comparison between our work and those related in this category may be found in Section 4.4.

1.7 Thesis Structure

This thesis is structured as follows:

Chapter 2 presents an approach that visualizes CSP in terms of the graphical notations provided by UML. A tool U2CSP*v1* is developed which inputs a UML model and generates CSP specification that can be fed-directly into FDR for model-checking. A few examples are shown to illustrate the mapping strategy defined for this purpose. A discussion is included which explains why UML class diagrams, state diagrams and certain notations in these diagrams are used. This chapter then concludes with a comparison with other work.

Chapter 3 & 4 focuses on UML state diagrams and presents a formal semantics for the diagrams in the CSP framework. This is done by first defining a structural model for the UML state machine. Using the model, we then define a behavioural semantics for UML state diagrams in terms of CSP. U2CSP*v2* is developed, which is an enhanced version of U2CSPv1 to cover the additional features introduced by this work. A comparison with other related work is presented at the end of the two chapters.

Chapter 5 looks at two case studies with the aim of showing how we may model a system in UML, translate them into CSP and most of all, how FDR may be used to analysis and check for the correctness of the design.

Chapter 6 runs a few analysis on the work produced in this thesis. Firstly, we present the results of comparison between Approach A (as presented in Chapter 2) and Approach B (as described in Chapters 3 & 4). Secondly, an analysis is carried out to compare the CSP generated from our proposed graphical model and the CSP written in a usual way. Lastly, a comparison is made between the UML semantics proposed in this thesis with that of the OMG semantics.

Chapter 7 concludes this thesis and make some suggestions for future work.

Chapter 2

Visualizing CSP in UML

2.1 Introduction

In this chapter, we seek to provide a graphical representation to CSP using UML. We aim to produce a mechanism which allows users to design a system in UML diagrams that are then translated to CSP in an automated mean. This helps users who are not familiar with CSP to be able to make use of formal methods in their design process without having to write the CSP specification themselves. Our work in this chapter emphasizes on providing a graphical visualization for CSP without adhering strictly to the semantics of the graphical notations being used.

We divide the visualization task based on three aspects of CSP: (i) the sequential behaviour, (ii) the parallel composition, and (iii) the refinement assertions. The sequential behaviour considers the events and transitions that are involved in a process. The parallel structure refers to the relationship between different processes, this may include parallel composition, indexed parallel or indexed interleaving. The refinement assertions model the refinement construct in CSP. For each of these aspects, we propose ways to visualize CSP using UML constructs. To this end, we choose to work with a subset of UML constructs from the class diagrams and the state diagrams. In particular, we only consider simple state diagrams with flat hierarchy and simple states without any state actions. A translator **U2CSP*v1*** is developed which inputs a UML model and generates CSP specification that can be fed directly into FDR for model-checking. This work is published in [52].

2.2 Sequential Behavioural View

The sequential behaviour of CSP is modelled using a UML state diagram. In this, we use a UML state machine to represent a CSP root process. A CSP root process has a global data state associated with it and intuitively, we may represent the data states using the UML state identifiers. For this, we use a state in a state diagram to represent a state identifier in a CSP process. In order to avoid the confusion between a CSP state identifier and a UML state, from now on we shall refer a CSP state identifier as a CSP process identifier. For the special CSP process identifiers such as *SKIP* and *STOP*, we may use an end state to represent *SKIP*, and a state with no outgoing transition as *STOP* (see Figure 2.1).

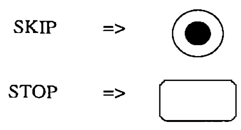


Figure 2.1: Representing *SKIP* and *STOP* in UML.

A state in a CSP process may be changed by atomic events/activities and the effect of the atomic activities is represented by the assignment to the identifiers, i.e. $P = Q$ where P is the identifier and Q being an expression that contains several identifiers which may include P itself since the equation can have recursion. A similar concept can be found in UML, whereby the state change from one to another is by execution of a transition. Therefore we may have the following mapping where we map a CSP process assignment to a UML state transition, as shown in Figure 2.2a.

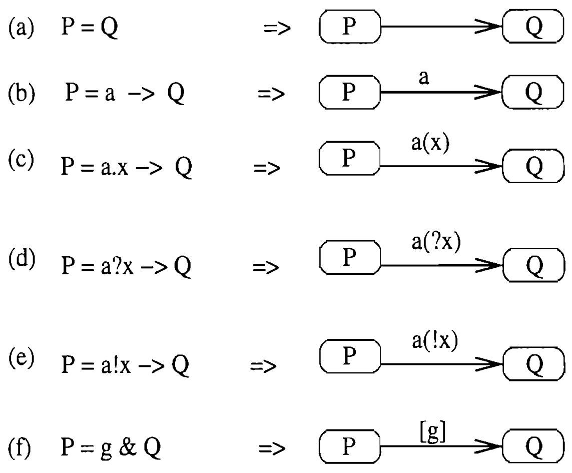


Figure 2.2: Mapping CSP to UML.

Furthermore, we can also map a CSP event prefix to a UML transition with an event (see Figure 2.2b). In this, we have a straight forward mapping from a CSP event to a UML event. In addition to simple events, we might have other information attached to an event to get a compound event such as event with argument $a.x$, event with input $a?x$ or event with output $a!x$. In UML, the syntax for the event label may include a list of parameters separated by commas such that the format will look like this: *event-name(parameter-name,...)*. We can represent the CSP event information using the UML event parameter list in which case $a.x$ will be expressed as *a(x)*, $a?x$ as *a(?x)* and $a!x$ as *a(!x)* (see Figure 2.2c-e). The same mapping rule is applied to multiple-part compound event, e.g. $a.x?y$ can be expressed as *a(x?y)* in UML. In addition, we map a Boolean guard expression in CSP to an UML transition guard (see Figure 2.2f).

To visualize the external and internal choices in CSP, first of all, we take a look at two choice representations that are available in UML: (a) a choice state (represented as a diamond shape), and (b) a normal state with more than one outgoing transition. Every transition out of these states represents a branch for the choice and it may be attached with a guard. The two representations are distinguished in that for the choice state, the decision on which branch to take depends on the prior actions in the same execution step, and the external environment has no control over it. Because of this, it is also called a *dynamic conditional branch.* In contrast, a normal state with more than one outgoing transition denotes a *static conditional branch* - where the choice of branch depends on the trigger event (offered by the environment) that occurs upon exiting from the current state. We can conveniently adopt these concepts and use (a) a choice state with multiple outgoing transitions to represent a CSP internal choice, (b) a normal state with more than one outgoing transitions to represent a CSP external choice. (see Figure 2.3). Although it is not explicitly stated in the diagram, the external and internal choices can both be generalized from 2 branches to n-branches, the tool described in section 2.5.1 supports this.

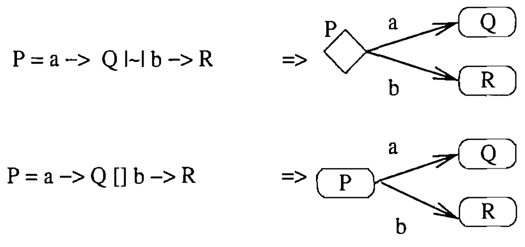


Figure 2.3: An example of representing CSP internal and external choices in UML.

In some cases, we might want to model $Q \square R$ or $Q \sqcap R$. For this, we may represent them in UML as in Figure 2.4.

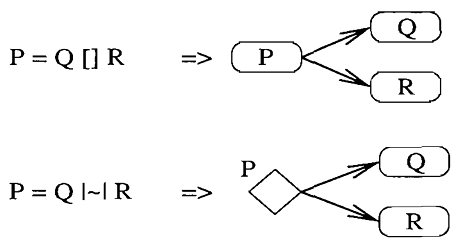


Figure 2.4: Representing "$P = Q \square R$" and "$P = Q \sqcap R$".

Often in CSP we wish to call a process with expressions substituted for its process parameter(s), e.g. we might want to call a parameterised process *Q(i)* with its parameter *i* substituted *i+1*. To represent this in UML, we use the transition action in state diagrams, which according to Harel's Statecharts semantics [30], can be used to represent modification of data values. In our running example, we map the substitution expression *"i:=i+1"* to a UML transition action and the parameterised process *Q(i)* to the transition target state. Figure 2.5 depicts this.

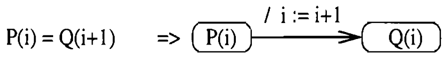


Figure 2.5: Representing "P(i) = $Q(i+1)$" in UML

Here we assume the parameter i is input to the system via the transition event that occurs prior to state P and it is then stored as a class attribute at state P, e.g. *P(i)*. The stored value can then be used in the next transition. The example in Section 2.2.1 shows this.

2.2.1 An example

Given the mapping we have defined earlier which maps from CSP to UML, we will use an example here to show how we can design a system in UML and make use of the mapping rules to convert the UML diagrams to CSP. Figure 2.6 illustrates a simple counter. It is a recursive process that begins by the user inputting a value x. x will be incremented by 1 at each iteration as long as it is less than 10, else the process will terminate.

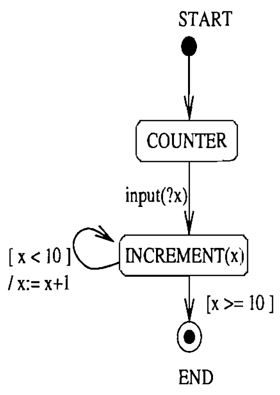


Figure 2.6: A simple counter.

Figure 2.6 can be mapped to CSP as follows.

$$\begin{aligned} START &= COUNTER \\ COUNTER &= input?x \rightarrow INCREMENT(x) \\ INCREMENT(x) &= (x < 10 \; \& \; INCREMENT(x+1)) \;\Box\; (x \geq 10 \; \& \; END) \\ END &= SKIP \end{aligned}$$

2.3 Parallel Composition View

In the previous section, we have illustrated how we may use the UML state diagrams to model the sequential behaviour in CSP processes. From here on, we propose a way to gather these processes and visualize the static relationships, e.g. the parallel composition between them. For this, we only consider parallel composition between sequential processes, that is the parallel structure in a higher level. We justify this based on the reason being most of the case studies we have come across have this structure in general. For the examples we have come across so far, it seems to be sufficient to say that all the parallel composition is used in the outer most level of the process hierarchy. We have not come across and hence do not support any use of parallel composition for substates, in order to keep our representation simple and manageble.

2.3.1 Initial Design

Initially, we attempted to construct a graphical notation which closely resembles the structure of the static architecture for the CSP textual notation. We achieved this by ignoring largely the structural semantics of UML class diagram and placing emphasis on attaining a straight-forward translation rule for the automated tool. However, we soon ran into problems: the class diagrams generated in such a way are not able to render a clear visualization. For example, to visualize the parallel composition between processes P and Q over synchronized events {a,b} such as System = $P \underset{\{a,b\}}{||} Q$, we model it as shown in Figure 2.7.

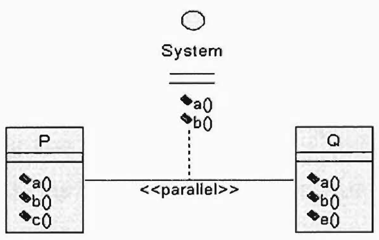


Figure 2.7: An initial approach to visualize the parallel composition of CSP.

We argued that the *association* between P and Q may be stereotyped as ⟨⟨parallel⟩⟩ to represent the parallel relationship. The *association class* (depicted as a dotted line) connects the *association* to the *interface class* labelled "*System*" and it may be used to model the properties of the association, which in this case is the parallel composition. The properties, which contain the synchronized events {a,b} may then be stored in the operation clause of the *interface class.*

This method is obviously not ideal: we have, in some ways, misused the structures in UML class diagram to suit the need of our graphical representation and caused confusion to the UML users. In addition, the unnecessary constructs like the interface class and the association class have cluttered the diagram and greatly distorted the visual quality. This style of representation has proved to be clumsy when more processes are involved. The example in Figure 2.8 illustrates this point, where we are trying to visualize the parallel composition between processes *SndMess*, *RcvAck*, *RcvMess*, *SndAck*, *Tx(i)* and *Rx(i)* over six common events: *snd_mess*, *rcv_ack*, *rcv_mess*, *snd_ack*, *mess* and *ack*. The example has shown that the proposed style is not able to demonstrate clearly the relationship between processes as the line does not represent real communication between them, e.g. it is not clear that *Tx(i)* synchronizes with *SndMess* and *RcvAck*. After the unsatisfactory attempt, we considered the alternative to be discussed in the following sections.

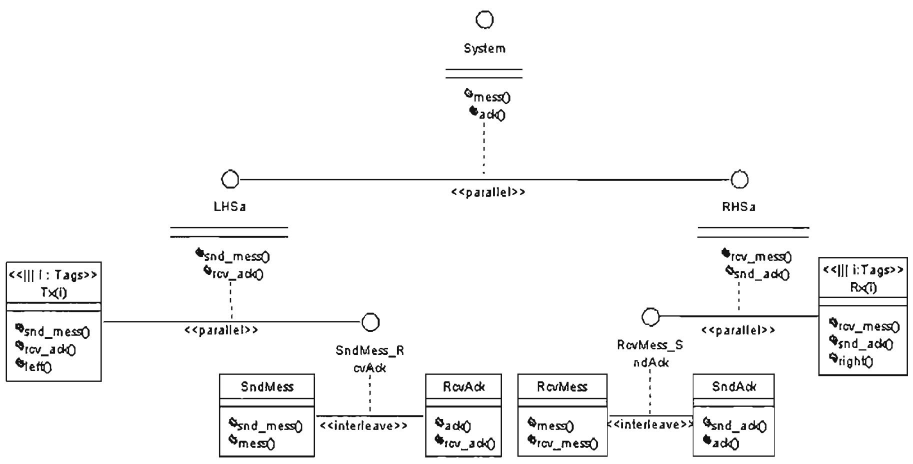


Figure 2.8: Initial attempt to visualize parallel composition involving more than two processes.

2.3.2 Simple Parallel

The conventional role of a UML class diagram is to provide a structural architecture for classes and model the static associations between them. A class acts as a template for all the object instances sharing the same behaviour. Contrary to this concept, our work in this chapter treats each **class** as a CSP *process*. In our representation of CSP, a class and the initial state of its state diagram share the same name.

All the *events* that are involved in a process are listed under the **class operation** clause. We use a UML class association to represent a CSP channel that serves as an interface between two CSP processes. The association label is used to name the common channel (see Figure 2.9(a)). In the case when more than two processes are sharing a common channel, we represent the common channel with a UML interface class, which more than two processes may be connected to (Figure 2.9(b)).

In Figure 2.9(b), three processes are in parallel with one another over some common channels: A shares channel b with B and channel d with C. B shares channel c with C and the three processes in turn share channel a. Applying our mapping strategy,

$$(A \underset{\{a,b\}}{||} B) \underset{\{a,c,d\}}{||} C \qquad (2.1)$$

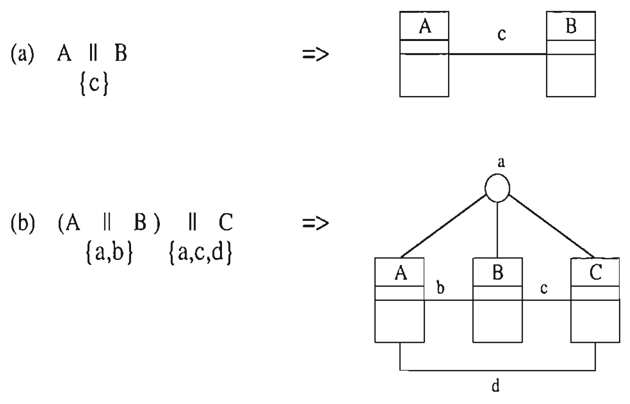


Figure 2.9: Visualizing the parallel composition in CSP.

The pair-wise composition in the CSP expression can be done in any combination, hence, we might write Eq.2.1 as

$$A \underset{\{a,b,d\}}{\|} (B \underset{\{a,c\}}{\|} C) \;,$$

$$(A \underset{\{a,d\}}{\|} C) \underset{\{a,b,c\}}{\|} B \; or$$

$$A \underset{\{a,b,d\}}{\|} (C \underset{\{a,c\}}{\|} B) \;.$$

The above three equations are equivalent to Eq.2.1 based on the associative[1] and symmetric[2] laws [60].

Lastly, we model the interleaving relationship between two processes using two classes with no association connection between them. We show in Figure ?? how we reconstruct Figure 2.8 using the method proposed in this subsection. Under the new method, we can see clearly from the diagram the common channels that are shared among the six processes.

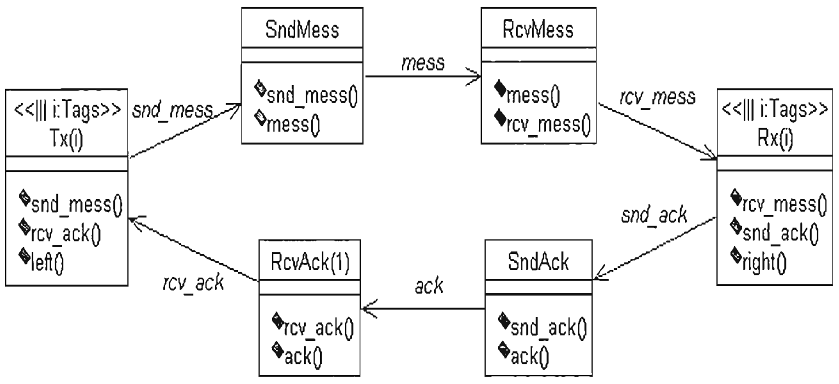


Figure 2.10: Visualizing Figure 2.8 using a better alternative.

[1]$(A \| B) \| C = A \| (B \| C)$
[2]$A \| B = B \| A$

2.3.3 Indexed Parallel/Interleaving

For indexed operations such as indexed parallel, i.e. $\| \ t : T \ A(t)$ or indexed interleaving, i.e. $|||t : T \ A(t)$, we use the stereotype ($\langle\langle \ \rangle\rangle$) of the class icon to represent the indexing. An example is shown in Figure 2.11. For each of the diagrams in the figure, there are multiple copies of *A(t)* or *B(t)* with t from a set T running in parallel/interleaving with one another (denoted by ⟨⟨||t:T⟩⟩ or ⟨⟨|||t:T⟩⟩).

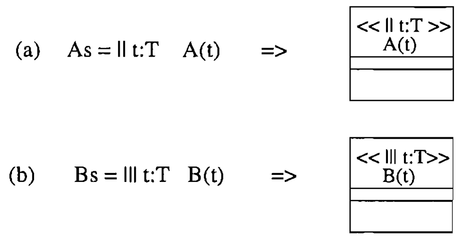


Figure 2.11: Modelling in UML (a) Indexed Parallel, (b) Indexed Interleaving

2.4 Refinement Assertion View

In this section, we discuss the final aspect of CSP visualization in UML by looking at the CSP refinement assertion, and we accomplish the task using the class diagram. There are two participants involved in a refinement assertion: the *abstract specification* and the *concrete implementation*. Assuming we have an abstract process A and a concrete process B such that $A \sqsubseteq B$. We use the **realize relation** to connect B to A with the arrow pointing to A as in Figure 2.12(a). In the case where there is more than one process involved in the implementation, we group the processes (i.e. classes) into a package as the one named B found in Figure 2.12(b). Package is used here as a higher level process that represents all other processes in the lower level. Note the label ⟨⟨*T*⟩⟩ beside the dotted line. It represents a *trace* refinement. In a similar way, we can model the failure-divergence or failure refinement using ⟨⟨*FD*⟩⟩ or ⟨⟨*F*⟩⟩ respectively. The hidden events are generated automatically by the tool by comparing the events in the specification with the events in the implementation. Here, we assume that the set of events found in the specification is a subset of those found in the implementation, e.g. $\alpha A \subseteq \alpha B$[3].

2.5 Tool Support

We built our UML model using the commercial tool Rational Rose©[76] provided by IBM©. We are currently using *Rational Rose 2000e, Rose Enterprise Edition* run on the Windows 2000 platform. Using the mapping strategy we have proposed in this chapter, we develop a translator (U2CSP*v1*) which will take in the UML diagrams and generate automatically CSP that is accepted by FDR. Essentially, what the tool does is it inputs a

[3]αX denotes the set of event alphabets found in process X

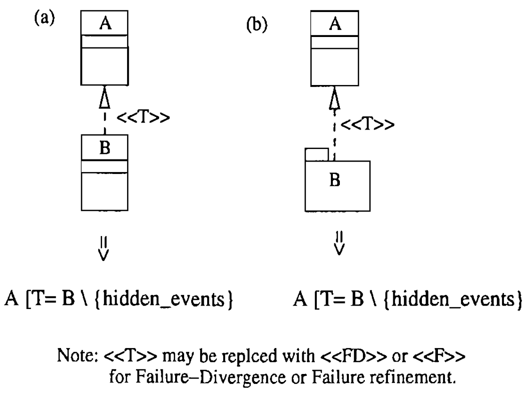

(a)
A
<<T>>
B
A [T= B \ {hidden_events}
(b)
A
<<T>>
B
A [T= B \ {hidden_events}
Note: <<T>> may be replced with <<FD>> or <<F>>
for Failure–Divergence or Failure refinement.

Figure 2.12: Visualizing CSP refinement assertion in UML

UML model consisting of one class diagram and one or more state diagrams and translates them into a CSP specification in the form of a text file. The text file can then be fed into the FDR tool for model-checking (see Figure 2.13).

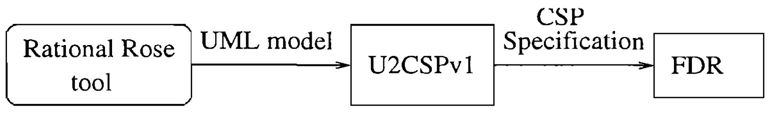


Figure 2.13: The tools involved in generating CSP specification from a UML model for model checking.

In the Rational Rose environment, a UML model is drawn up in a hierarchical manner. Each model contains at least one class diagram featuring different classes and it is situated at the top level of the model. Each of the classes models a CSP process and each class contains exactly one state diagram that is used to model the sequential behaviour of the process. To avoid the class diagram getting cluttered with too many classes, we may hide the classes in a package. In this way, the classes are arranged in a lower level, providing clearer and simpler abstraction at the higher level. Given that at this stage we do not support any nested state, i.e. a substate situated within a composite state, each state diagram has a flat structure.

An example of a model drawn in the Rational Rose environment is shown in Figure 2.14. It contains three windows showing diagrams at three different levels that make up the model. The *Class Diagram: Logical View/Main* is situated at the top level, showing *Design* refining *Specification*. The *Design* is represented by a package which contains three classes, as shown in the *Class Diagram: Design/Main* in the next lower level. At this level, the classes represent *ProcessA*, *ProcessB* and *ProcessC* and they share some common event channels between them. For each of these processes, there is a state diagram attached to it illustrating the sequential behaviour involved in the process. An example of such state diagram is shown in the *Statechart Diagram:ProcessC/NewDiagram* (at the bottom right corner) which corresponds to *ProcessC*. The other two state diagrams corresponding to *ProcessA* and *ProcessB* are not shown here.

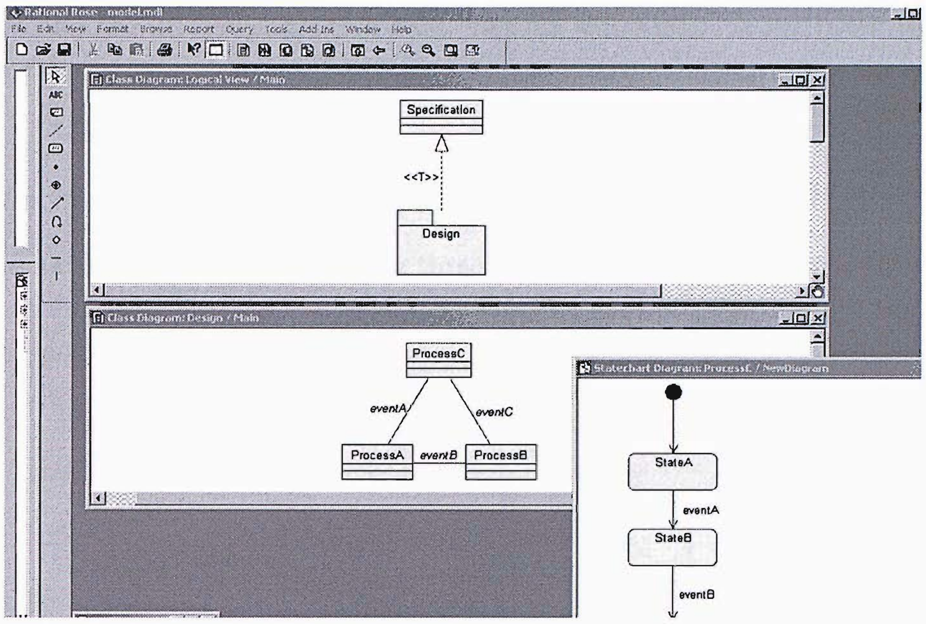

Rational Rose - model.mdl
File Edit View Format Browse Report Query Tools Add Ins Window Help
Class Diagram: Logical View / Main
Specification
<<T>>
Design
Class Diagram: Design / Main
ProcessC
eventA
eventC
ProcessA
eventB
ProcessB
Statechart Diagram: ProcessC / NewDiagram
StateA
eventA
StateB
eventB

Figure 2.14: An example of a UML model in the Rational Rose Environment.

2.5.1 U2CSP*v1*

U2CSP*v1* is essentially a script file that is built into the Rational Rose environment. It is written in the Rational Rose Scripting language, which is an extended version of the Summit BasicScriptlanguage [12, 13, 14]. There is a script editor that runs in the Rational Rose environment that provides access to the scripting environment. The tool is configured as an option on the menu. When U2CSP*v1* is invoked, the translator takes in the current UML model and retrieves the necessary information from the model. It then generates a CSP specification with the file extension *.csp*. The current version of U2CSP*v1* is a prototype used to explore the mapping strategy and the efficiency of the concept.

2.6 Example

In this section, we are going to demonstrate how we can use UML as a graphical front-end to design a system, and then use the mapping strategy we have devised to translate the diagrams into CSP that can be fed into FDR for further model-checking. Here, we would like to show how individuals with no experience of CSP can use CSP in their process of designing a system without having actually write the CSP code themselves.

2.6.1 Lift System

In this example, we would like to design a lift system. In the system, there is a lift door and one door at each of the floor. When the lift arrives at a floor, both the lift and the floor doors will open. A passenger then enters the lift and presses a button corresponding to the floor they wish to go. The lift and the floor door close before the lift moves to the next destination floor. During the course of the lift moving, an emergency button may be pressed and the lift will come to a halt. The lift will remain at halt until the release button is pressed. For simplicity, we have not considered the mechanism for requesting the lift.

Sequential Behavioural View

We begin our design by first identifying three main entities in the system: the lift itself, the door at each floor and the emergency button. For each of these entities, we define its sequential behaviour using a state diagram.

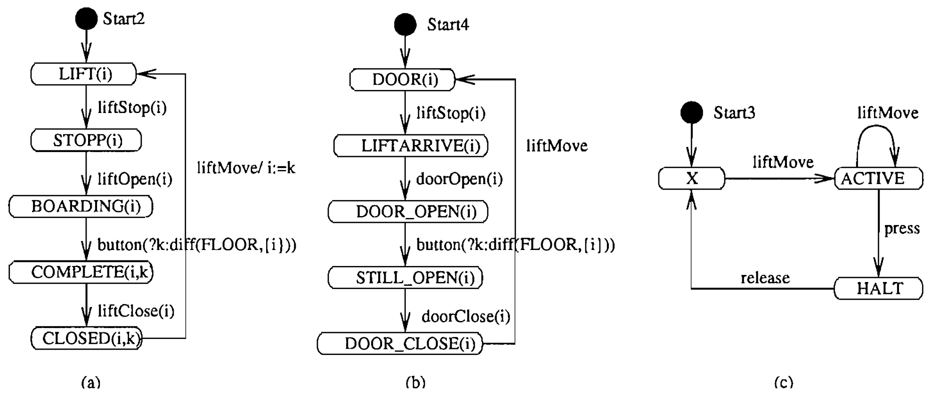


Figure 2.15: State diagram for (a) lift (b) floor door and (c) emergency button

We start by looking at the design of the lift itself. In Figure 2.15(a), the system starts at state LIFT(i). When the lift stops at the i^{th} floor, e.g. *liftStop(i)*, the lift opens, modelled by *liftOpen(i)*. Notice so far that when the states receive their incoming transition event parameters, we append the parameter i in the state names, e.g. *LIFT(i)* and *STOPP(i)*. Someone who enters the lift then presses any button k, except the current floor. Hence, we have k : *diff*$(FLOOR, i)$ where FLOOR is the set of all the floors which the lift serves, and *diff*$(FLOOR, i)$ is the set difference between FLOOR and $\{$i$\}$. Observe that both the arguments i and k are appended to the next state name, e.g. *COMPLETE(i,k)*. The lift then closes at i^{th} floor, e.g. *liftClose(i)*. Next, when the event *liftMove* takes place, it not only triggers a transition from *CLOSED(i,k)* to the next state, at the same time, it will also trigger an action $i := k$ that will substitute i with the next target floor k, and the next state *LIFT(i)* will have the value of k appended to its state name. The whole process is repeated with the target floor k.

Using the proposed mapping strategy, we may now translate Figure 2.15(a) into CSP as follows:

$$\begin{aligned}
\text{Start2} &= \text{LIFT(i)} \\
\text{LIFT(i)} &= \text{liftStop.i} \rightarrow \text{STOPP(i)} \\
\text{STOPP(i)} &= \text{liftOpen.i} \rightarrow \text{BOARDING(i)} \\
\text{BOARDING(i)} &= \text{button?k:diff(FLOOR,}\{i\}) \rightarrow \text{COMPLETE(i,k)} \\
\text{COMPLETE(i,k)} &= \text{liftClose.i} \rightarrow \text{CLOSED(i,k)} \\
\text{CLOSED(i,k)} &= \text{liftMove} \rightarrow \text{LIFT(k)}
\end{aligned}$$

We do the same for the floor door (see Figure 2.15(b)). Bear in mind that there is more than one floor door involved in the system, hence i in *DOOR(i)* refers to the door at the specific floor i. The CSP translation for this diagram may be found in Appendix A.1.

Figure 2.15(c) shows the design for the emergency button. The process begins at X. When the lift moves, the system will transit to state *ACTIVE*. The system will remain at this state as long as the lift is moving, e.g. event *liftMove* is offered by the environment continuously. If the emergency button is ever pressed, e.g. event *press* takes place, and the system comes to a *HALT*. The only possible way for the system to get out of the state *HALT* is when *release* is pressed. The system then goes back to state X, and the whole process is repeated. The corresponding CSP for the state diagram based on our mapping strategy is as follows:

$$\begin{aligned}
\text{Start3} &= X \\
X &= \text{liftMove} \rightarrow \text{ACTIVE} \\
\text{ACTIVE} &= \text{press} \rightarrow \text{HALT} \;\Box\; \text{liftMove} \rightarrow \text{ACTIVE} \\
\text{HALT} &= \text{release} \rightarrow X
\end{aligned}$$

Parallel Composition View

In the previous section, we have drawn three state diagrams with each illustrates the individual process for the lift, the floor door and the emergency button. Now, we need to combine these individual processes and put them in parallel in order to produce a complete design. We achieve this using the method developed in Section 2.3. As shown in Figure 2.16, we have three classes representing the three main processes: *LIFT(1)*, *DOOR(i)* and X . Notice that for process *DOOR(i)*, it is stereotyped as $\langle\langle\ |||i{:}FLOOR\ \rangle\rangle$. Since $FLOOR = \{1..N\}$, the stereotype indicates that there are N copies of processes DOOR(i) interleave with one another. Translating the class into CSP using the mapping strategy, we get

$$DOORs = |||i : FLOOR \quad DOOR(i)$$

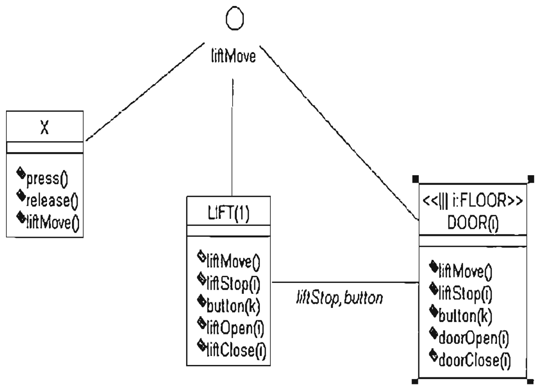


Figure 2.16: The parallel composition between the CSP Processes

Figure 2.16 is interpreted as follows : *DOORs* is in parallel with *LIFT(1)* and they synchronize via the common channels *liftMove, liftStop* and *button.* They in turn are in parallel with X (since X is in parallel with both *LIFT(i)* and *DOORs*) via the common channel liftMove. The parallel composition is expressed in CSP as

$$System1 = DOORs \underset{\{|liftMove, liftStop, button|\}}{||} LIFT(1)$$

$$System = System1 \underset{\{|liftMove|\}}{||} X$$

Refinement View

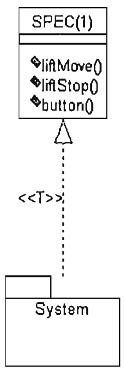


Figure 2.17: The refinement assertion

All the three classes defined earlier are grouped together using a package named *System* as shown in Figure 2.17. The package forms the concrete design for the system, and this is used to refine the abstract specification *SPEC(1).* A refinement assertion (shown below)

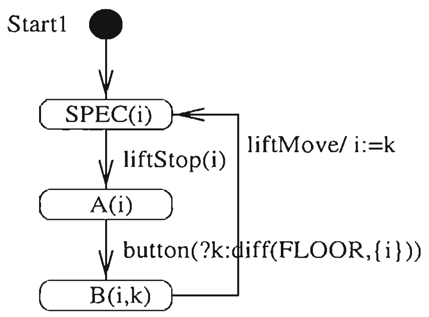


Figure 2.18: Modeling the abstract behaviour of the System.

is generated as follows:

assert SPEC(1) $\sqsubseteq_T$ *System*$\backslash\{|$*press,release,doorOpen,doorClose,liftOpen,liftClose*$|\}$

To define the abstract specification, we specify the basic requirements such that when the lift is in floor *i*, the next destination floor will depend on the button being pressed. The requirement is modelled via the state diagram found in Figure 2.18. We obtain the following CSP representation that corresponds to Figure 2.18.

$$\begin{aligned} START1 &= SPEC(i) \\ SPEC(i) &= liftStop.i \rightarrow A(i) \\ A(i) &= button?k{:}diff(FLOOR,\{i\}) \rightarrow B(i,k) \\ B(i,k) &= liftMove \rightarrow SPEC(k) \end{aligned}$$

The Overview

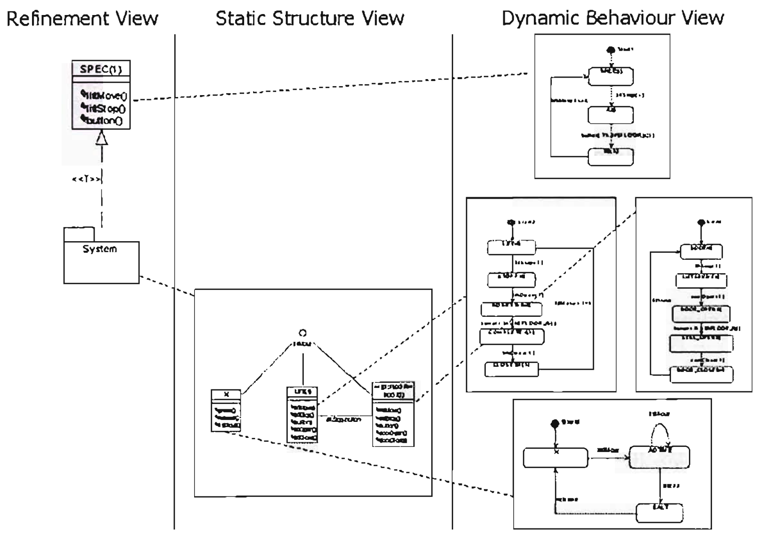


Figure 2.19: An overview of the lift system in UML.

Figure 2.19 put together all the diagrams we have developed so far and present an architectural structure showing how the lift system is modelled in UML. U2CSP*v1* is used to translate these diagrams to CSP. We initiated the variable $FLOOR$ with $FLOOR = \{1..4\}$ in the UML diagrams. (Refer Appendix A.1 for the full CSP textual representation generated for this case study.)

2.6.2 Multiplexed Buffer

This example is taken from [59]. It models a multiplexed buffer system which comprises a number of buffers placed at both transmitting and receiving sides of a communication channel (see Figure 2.20). The channel may be one/both ways. There are four main processes involved in the system: SndMess(send message), RcvMess(receive message), SndAck(send acknowledge) and RcvAck(receive acknowledge). On top of these, we also have local processes for the Tx(transmitter) and Rx(receiver).

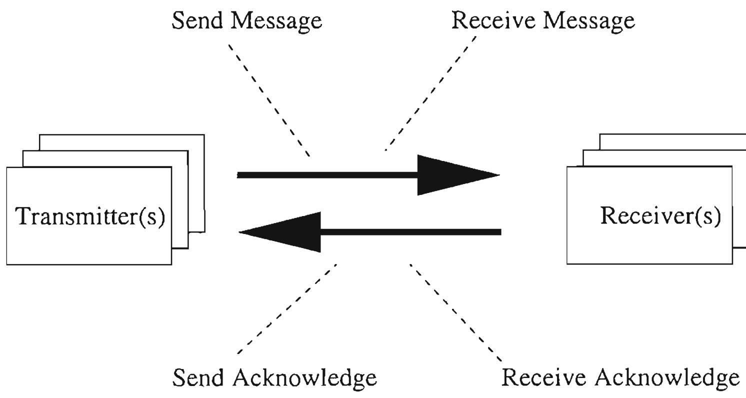


Figure 2.20:

These processes interact with one another by synchronizing over some common events. The interaction is shown using a class diagram in Figure 2.21. The association between two processes shows the common channel which is shared between the processes. We may model the parallel relationship in the following way using CSP: the translator will randomly pick a process to start with. In this example, the translation begins at *Txs* and it synchronizes with *SndMess* over channel *send_mess* (Eq. 2.2). The combination in turn synchronizes with *RcvAck(1)* over *rcv_ack* (Eq.2.3). They in turn synchronizes with *RcvMess* over *mess* (Eq.2.4). The combination of four then synchronizes with *SndAck* over *ack* (Eq.2.5), and lastly, *Rxs* synchronizes with the rest of the processes over two channels *rcv_mess* and *snd_ack*. All these processes are grouped under a main process named *System* (see Eq. 2.7).

$$\begin{aligned}
\text{System1} &= \text{Txs} \underset{\{|snd\_mess|\}}{||} \text{SndMess} && (2.2)\\
\text{System2} &= \text{System1} \underset{\{|rcv\_ack|\}}{||} \text{RcvAck(1)} && (2.3)\\
\text{System3} &= \text{System2} \underset{\{|mess|\}}{||} \text{RcvMess} && (2.4)\\
\text{System4} &= \text{System3} \underset{\{|ack|\}}{||} \text{SndAck} && (2.5)\\
\text{System5} &= \text{System4} \underset{\{|rcv\_mess,snd\_ack|\}}{||} \text{Rxs} && (2.6)\\
\text{System} &= \text{System5} && (2.7)
\end{aligned}$$

Observe that *Txs* and *Rxs* are indexed interleaving processes, as denoted by $\langle\langle i : Tags\rangle\rangle$ on the diagram. They are expressed in CSP as

$$\begin{aligned}
\text{Txs} &= |||\ \text{i:Tags}\quad \text{Tx(i)} && (2.8)\\
\text{Rxs} &= |||\ \text{i:Tags}\quad \text{Rx(i)} && (2.9)
\end{aligned}$$

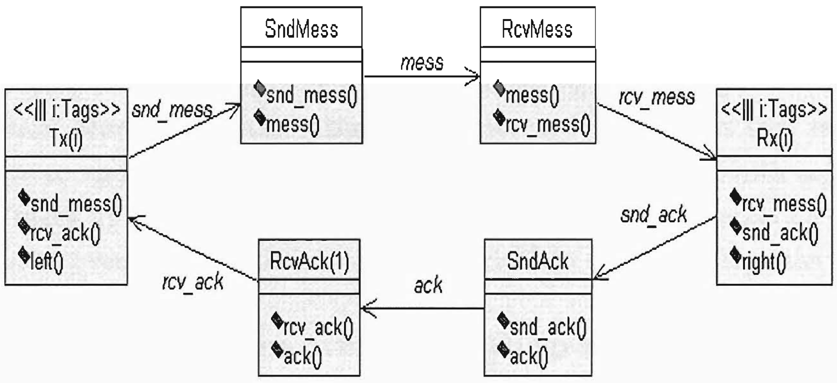


Figure 2.21: The static relationships for processes in the multiplexed buffers system.

Figure 2.22 shows how *System* is being used to refine the abstract specification *Buffer(i)*. The corresponding CSP representation for the diagram is

$$\begin{aligned}
\text{Buffers} &= |||\ \text{i:Tags}\quad \text{Buffer(i)}\\
\text{assert Buffers [FD} &= \text{System} \setminus\\
&\quad \{|\text{snd\_mess,rcv\_ack,mess,ack,rcv\_mess, snd\_ack}|\}
\end{aligned}$$

For each of the processes shown in Figure 2.21 and 2.22, a corresponding state diagram is drawn to model the event transition for the process. These state diagrams together with the the full list of the CSP representation for the system can be found in Appendix A.2.

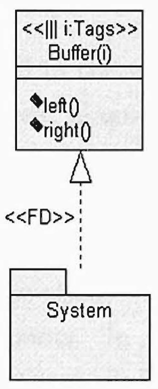


Figure 2.22: The refinement relationship in the multiplexed buffers system.

2.7 Discussion

2.7.1 Why State Diagrams over Activity Diagrams?

In general, both state diagrams and activity diagrams are used to show the states in which an object resides. As pointed out in [17], the primary difference between the two is that: the transitions between states in the state diagrams are particularly triggered by the events produced in the environment. Conversely, the transitions between states for activity diagrams occur not because of event triggers, rather, the transitions are due to the completion of the activities performed within an activity state. From this, we may say that the state diagrams are more adept in modelling reactive systems that react to event occurence. Moreover, a state diagram is concerned with events that take a system from one state to another whereas an activity diagram is concerned with activity within a state that takes up time. Since CSP is a notation concerning interaction of processes with reactions to events, we feel it is more appropriate to use state diagrams in our work.

2.7.2 Why Class Diagrams over Other Diagrams?

To visualize the parallel structure and refinement assertion of CSP, we have adopted class diagrams over all other UML diagrams. The reason class diagrams are chosen is mainly because they are able to provide a clear hierarchical structure for the model of a system. The feature provided by the Rational Rose© modeller allows a state diagram to be nested within a class. With this, we can model the parallel structural behaviour of the system using the class diagram and the dynamic sequential behaviour using the state diagram independent from one another, but at the same time maintain the link between the two. Also, in a class diagram one is allowed to group more than one class into a Package, and this method proved useful in our approach to visualizing the refinement assertion. Furthermore, the operations shown on each class entity can be used to display distinctively the events that are involved in a process.

Having said this, UML has offered two types of physical diagrams at our disposal: Component Diagram[24] and Deployment Diagram[24]. As mentioned before, a component diagram shows the relationship between different components, whereas a deployment diagram shows the physical deployment of a system into the real environment and indicates where the components are situated in the real world. Based on the definitions, one may suggest that we should use component diagrams when trying to visualize the parallel structure between different CSP processes. The main entity in a component diagram are **components**(refer Figure 2.23), with each depicted as a square box with two rectangles attached to the top left corner of the box. A component may have **interfaces** (represented as lollipops sticking out from the square box) which are the visible channels the component is offering to other components. In UML context, a component is used to group classes together. Although component diagrams may have the potential of replacing the class diagrams in our work, to a certain extent, we find working with component diagram in the Rational Rose environment to be tedious. First of all, interfaces need to be defined in a class diagram before they could be assigned to components. In other words, component diagrams cannot be used alone in a design but it must be incorporated with class diagrams. On the other hand, Rational Rose© does not provide any way to link a state diagram to a component in the Component Diagram. Without the link, it could make traversing the model to be confusing, and we will also loose the hierarchy structure that class diagram could offer. Hence, we have left out component diagram for the time being. We may consider using it in the future if a suitable tool is found to support its use in a better way, or if there are additional features for the component diagrams offered in the UML new version 2.0 to be released soon at the time of writing.

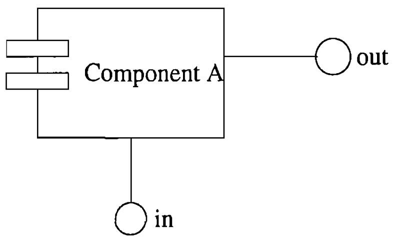


Figure 2.23: An example of a component

2.7.3 Fork and Join

Naturally, one might think that we should use join and fork in UML to represent the parallel composition in CSP. However, our observation suggests otherwise.

Before we explain further, we first introduce the notion of joins and forks. Typically, a *join* construct is used to merge several transitions from the source states to a single outgoing transition, while a *fork* construct is used to split an incoming transition into more than one outgoing transition. An example of how the joins and forks are used is shown in Figure 2.24.

For a fork, the events on its outgoing transitions can only take place after the event on the incoming transition has occurred. For this, we can see that *Search_Flight* and

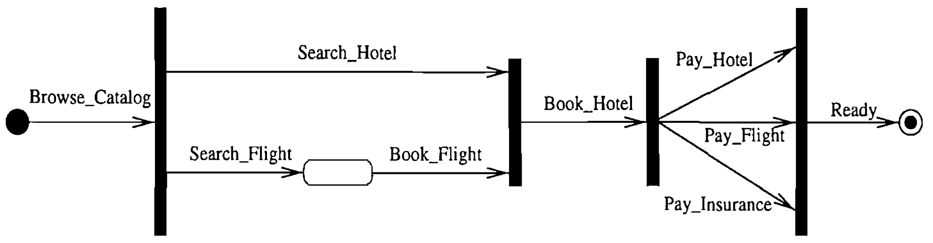


Figure 2.24: Using forks and Joins

Search_Hotel only take place (one after another, regardless of the order) after *Browse_Catalog*. Similarly, *Book_Hotel* can only occur after *Search_Hotel* and *Book_Flight* are completed. Comparing the notion of synchronization between UML and CSP, we observe that for UML, an event is synchronized in such a way that it is to occur before a second event takes place. Conversely, the synchronization in CSP deals mainly with executing a common event shared among different processes. Bearing these differences in mind, we may attempt to translate the diagram to CSP as below:

$$\begin{aligned} \text{P1} &= \text{Browse\_Catalog} \rightarrow \text{Search\_Hotel} \rightarrow \text{Book\_Hotel} \\ &\quad \rightarrow \text{Pay\_Hotel} \rightarrow \text{Ready} \end{aligned}$$

$$\begin{aligned} \text{P2} &= \text{Browse\_Catalog} \rightarrow \text{Search\_Flight} \rightarrow \text{Book\_Flight} \\ &\quad \rightarrow \text{Book\_Hotel} \rightarrow \text{Pay\_Flight} \rightarrow \text{Ready} \end{aligned}$$

To include the event *Pay_Insurance*, we would need to add another process such that

$$\begin{aligned} \text{P3} &= \text{Browse\_Catalog} \rightarrow \text{Search\_Flight} \rightarrow \text{Book\_Flight} \\ &\quad \rightarrow \text{Book\_Hotel} \rightarrow \text{Pay\_Insurance} \rightarrow \text{Ready} \end{aligned}$$

Each of these processes are obtained by tracing the diagram from the initial state onwards until the final state is reached. As a results, three different traces or processes are formed that cover all the possible routes through different branches. Lastly, we may combine these processes to get

$$\begin{aligned} \text{BOOKING\_HOLIDAY} &= (\text{P1} \underset{Browse\_Catalog, Book\_Hotel, Ready}{\|} \text{P2}) \\ &\qquad \underset{Browse\_Catalog, Book\_Hotel, Ready, Search\_Flight, Book\_Flight, Book\_Hotel}{\|} \text{P3} \end{aligned}$$

The translation may seem to work well but we foresee some complications in it. First of all, all the processes are combined under the same diagram, hence, there is no clear and direct visualization of individual processes that are involved in the system. Secondly, it is less obvious which event the processes are synchronized on, as compared to the association we have used to model a common channel between two processes. Thirdly, the diagram might become more cluttered, when many processes are involved. In our two-tier hierarchical representations, we are able to reduce the complexity of the diagram by visualing separately the sequential behaviour and the parallel composition of a system using a class diagram and one/more state diagrams (with one state diagram corresponding to a process in the system). Based on all these reasons, coupled with the consideration of simplicity for our automated translation tool, we have decided not to use forks and joins in our work.

2.8 Comparison with Related Work

We gained our initial inspiration from the work carried by Bolton & Davies [8] which involves Activity Graphs and CSP. The work takes a different approach in which it defines a formal semantics of Activity Graph and then compares it with CSP, whereas we concentrate on giving a full representation of CSP in UML, and emphasizes providing a graphical support towards CSP.

The work by Davies & Crichton [15] provides a formal behavioural semantics to combinations of class, object and state diagrams using CSP. They use a class diagram to describe how objects from different classes can communicate by calling operations on one another and they use a state diagram to show how an object will react to the arrival of an event. In their work, a state diagram for a class is used to describe a parameterised communicating process which is based on the run-to-completion assumption. Their work interpretes a class as a template behaviour for all the objects sharing the same behaviour, whereas our work assumes a class as a CSP process, and hence our class and the initial state of its state diagram share the same name. In comparison, their work resembles more closely to the informal semantics of UML (as interpreted by OMG).

The work by Brooke et al. in [10] is closely related to ours. They are providing a graphical notation for timed CSP (TCSP). Their work is different from ours in a few aspects. First of all, [10] does not provide a complete graphical representation of TCSP in that it does not support representation of refinement notion. Our work is able to do so by using the realize relation under the class diagram to visualize the design process refining the specification. Secondly, we have used the association relationship between classes to model the parallel composition between two processes, as opposed to theirs which does so by placing all the the processes that are in parallel within a square box annotated with "*Synch*{}" (see Figure 2.25). Thirdly, their work opts for Harel's state-

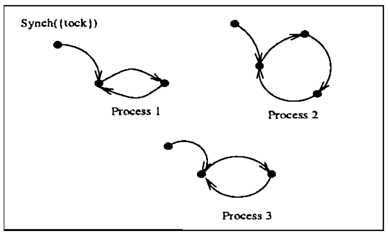


Figure 2.25: A representation of parallel processes in [10].

charts rather than the UML diagrams because they want to avoid the imprecise semantics problem associated with UML. In contrast, we have chosen UML mainly for its wealth of notation offered under different diagrams, in which case we have been utilizing the notations offered by the class diagram and the state diagram for our work. Beyond that, the readily available commercial tools (such as the Rational Rose CASE tool) not only enables us to draw UML diagrams with ease but they also make integration with other tools simple, which in our case, we are able to write a simple script file that works in the Rational Rose environment to generate CSP from the UML diagrams. To compensate for the problem of imprecise semantics of UML, we are providing a formal semantics to UML state diagrams using the formal semantics of CSP. This work is presented in Chapter 3 & 4. Fourthly, the approach they proposed in visualization is complicated, in that they first define a text-based machine readable language (MRL) that describes the TCSP graphical notation. Then, they develop a drawing tool that will draw the program expressed in the MRL. Next, they develop a converter which will take the drawing and transform them into some sort of notation to be fed into FDR or PVS [35, 55]. In comparison, we have taken much simpler approach, in that we integrate the commercial UML drawing tool Rational Rose with U2CSP*v1*. The only step required from the designers is to draw UML diagrams (in Rational Rose CASE tool) and U2CSP*v1* will translate the diagrams automatically to CSP that can be directly fed into FDR for model-checking.

The work by Abeysinghe et al. [1] examines two modelling paradigms: CSP and a subset of Role Activity Diagram (RADs) which is centred around the concept of roles and activities as opposed to processes and events in CSP. A role in RAD describes a sequence of steps/activities which is carried out by an actor. There are two types of activities involved in a role: actions and interations, which cause a step change in the role. Actions are different from interactions in that the former is carried out by the actor of the role alone whereas the later involves other roles as well. RADs have their strength over UML state diagrams in that they are capable of modelling actions that are synchronized between two roles in a much simpler way as opposed to the synchronization using fork and join under UML state diagrams. Also, RADs are able to support refinement notion in the same diagram, and they have a notation called *"part refinement"* which refines the state of a role into a number of separate parts. We feel that this way of representation is useful and straightforward when dealing with small case studies, as both the basic requirements and the detailed refinement are presented in the same diagram. However the diagram may become too cluttered if bigger case studies are involved. For this, our work is able to provide a two-level hierarchical representation whereby a class diagram at the the top level provides an overall view to a system. The user can then choose to zoom into particular parts of the system by looking at the state diagrams in the lower lever for more detailed description of the system.

2.9 Conclusion

In this chapter, we have demonstrated how we can visualize CSP in UML using the mapping strategy we have devised. We use UML state diagrams to model the sequential behaviour of a CSP process, and UML class diagrams to visualize the parallel composition between the CSP processes and also the refinement assertion. The diagrams are then put together to give a complete representation of CSP in UML. Meanwhile, a prototype translator U2CSP*v1* has also been developed based on the mapping strategy that will automatically translate the UML diagrams to CSP that is accepted by FDR.

The main contribution for our work is we are able to introduce a graphical front-end as an entry-point for users who would like to use CSP in the design of a system. In this, our proposed graphical method presents the different components of a system in design in a hierarchical structure: each state diagram is embedded in a class, and classes can be clustered into packages. In addition, the proposed approach allows us to treat each process in a separate state diagram and hence enables us to deal with a system with many processes in a more organised manner. On top of this, designing in the graphical paradigm also provides an easy accessibility to relative novices. This is important when the designers need to deal with clients who have little knowledge of the specific designing language being used, and yet need to get involved to understand what is going on. However, the relative ease of using UML means that we lack formality in our descriptions. CSP supplements this, by having a model checker that can verify formally the correctness of behaviour for a system. Therefore, this suggests that there are benefits in attempting to use both notations in a complimentary ways, with UML notations as a tool in the design and client interaction stage, and CSP to verify the correctness and provide formality to the design. Lastly, we believe that being able to map from one paradigm to another gives a significant advantage to the system designers in reaping the potentials offered by both UML and CSP.

We have not covered all the constructs in CSP, and among these are sequence, event hiding, interrupt and renaming.

Chapter 3

Formalizing UML State Diagrams in CSP (Part 1)

3.1 Introduction

In the previous chapter, we address the issue of system design by looking at ways to improve the use of CSP. In this, we have provided a graphical representation for CSP in terms of UML. While doing so, we have uncovered the possibility of reasoning about UML state diagram constructs in terms of CSP. As mentioned earlier, UML is rich in its syntax constructs but still lacking in terms of having a formal behavioural semantics. In this regard, we wish to exploit the use of CSP to define properly the behaviour of a UML state diagram, especially those UML constructs which have not been covered so far. The main motivation for us to pursue this is because UML CASE tools such as Rational Rose© and Together/J [77] can actually be used to generate Java or C++ code from the UML models. Hence, using CSP to reason about the state diagrams will help to validate the design before implementation in terms of the actual program is produced.

In this chapter and the next one, we present a formal semantics for the UML state diagrams expressed in the CSP framework. We do this by first defining a structural model for the UML state machine. Using this model, we define our mapping from the UML structural model to CSP. U2CSP*v2* is developed, which is an enhanced version of U2CSPv1 to cover the additional features introduced by this work. Part of this work has been published in [53].

This chapter and the next one are essentially one long chapter divided into two. They are structured as follows. Section 3.2 explains the execution mode adopted by our formalization model. Section 3.4 defines a structural model for the UML state machine. Section 3.5 to 3.6 deal with some fundamental concepts involved in developing our formalization. In Section 3.7, 3.9 & 4.1, we present the formal definitions for the behaviour of different states in terms of CSP. For each definition, we include some informal explanation and examples where necessary to motivate the formal definition. Section 4.2 contains the work involved in the development of U2CSP*v2* tool. Section 4.3 discusses some miscellaneous issues involved in our formalization. Section 4.4 is devoted to comparing our work with others. Finally, Section 4.5 concludes the two chapters.

3.2 Formalization Execution Mode

Before we explain the execution mode adopted by our formalization model, let us first take a look at the execution modes that underpin UML and CSP.

In UML, as mentioned earlier in Section 1.5.3 under "Event Processing", the event generation by the environment is assumed to happen one at a time, and the events are collected and stored in an event queue which belongs to a state machine, which could be an OR-state or a subregion of an AND-state. An event is taken off from the queue in a FIFO manner and it is processed by its state machine as the current event. We may summarize the main features of the UML state diagram as follows:

1. The generation and consumption of an event occur asynchronously.
2. Only one event is offered to the state machine at one time as a result of the event queue.

In CSP, however, the execution mode takes on a different view:

1. The environment external to a process is allowed to offer more than one event and this is modelled using the external choice construct ($\Box$).
2. The generation and consumption of an event is assumed to occur synchronously. To achieve this, the environment external to a system is assumed to be running in parallel with the system.

To illustrate further, we will use the CSP classical example of a vending machine. In the example, a person (who acts as the external environment) may choose to have tea or coffee from a vending machine. The external environment which is the person in this case can be modelled as a CSP process as

$$PERSON = (tea \rightarrow PERSON) \Box (coffee \rightarrow PERSON)$$

The vending machine (which acts as the main process) will react to the person (environment) according to what the machine has to offer. If the machine only has coffee left, it might be modelled in CSP as $MACHINE = coffee \rightarrow MACHINE$. The $PERSON$ and $MACHINE$ are then composed in parallel as

$$PERSON \underset{\{coffee\}}{\|} MACHINE$$

Regardless of what is available in the vending machine, e.g. what the process chooses to do, deadlock will not occur because the person/external environment is ready to provide all the options.

However, things are different when we try to model the example in UML. Since the event queue offers only one event at a time, if the machine only has tea left and if the person (external environment) chooses to have coffee instead, a deadlock with occur. From here, we can see that the UML event queue model, which does not allow modelling of choice at the environment side, can pose serious deadlock problems. This is inherently a deficiency found in the event queue concept proposed by OMG.

With regard to this, we decided to adopt the CSP mechanism for our formalization model. In our formalization, we ignore the concept of UML event queue and replace it with the assumption that the environment is always ready to offer an event required by the process, and the generation and consumption of an event is assumed to occur synchronously. By adopting a synchronous execution mechanism, we view the environment external to a state machine as another CSP process running in parallel with the state machine. The event generation by the environment is synchronized with the consumption of event by the state machine. In this way, we will be able to avoid the deadlock problem mentioned above. Moreover, a model constructed in a synchronized mode as opposed to asynchronous mode will also make model-checking easier. This is important in our work since our ultimate goal is to model-check UML using CSP/FDR. We would like to stress that although we are using synchronous mode in our formalization model, we are still able to provide great insight into the interaction complexity concerning the UML state diagram. As we will see in the subsequent formalization, our model has uncovered many intrinsic details on the sequential execution involving different actions, state activity and event. This insight is especially valuable when state hierarchy is involved.

3.3 Well Formedness Rules

In this section, we list out the well-formedness rules that apply to our formalization model.

1. The hierarchy of the state must define a tree, e.g. no cycle is allowed.
2. Only one outgoing transition is allowed from an initial state.
3. At least one initial state must present within each level of a composite-OR-state hierarchy to indicate the start point upon entry into the composite state.
4. A transition originating from an initial state must always terminate at a state within the same hierarchical level where the initial state resides.
5. An incoming transition is not allowed to cross any state boundary (see Figure 3.1). We will discuss later in Section 4.3.2 the complications involved if this type of transitions are included in our model.
6. Only simple actions (i.e. transition action, entry action or exit action) are considered, no assignment statement is allowed in a state or transition action.
7. An entry state action consists of at most one simple action.

8. An exit state action consists of at most one simple action.
9. A state do-activity consists of at most one activity.

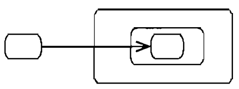


Figure 3.1: An example of a cross-boundary incoming transition which is not allowed in our model.

3.4 Structure of the State Diagrams

A UML state diagram represents a hierarchical state machine that includes the initial states, final states, choice states, simple states, composite states and transitions between states. Each of these constructs is distinguised and referred to by a unique identifier. We will begin by defining a structural model for the state diagram.

Assuming we have a state machine M such that the number of states in M is finite. The set of state identifiers found in M is denoted by $\mathcal{S}_M$. The set of transition identifiers found in M is denoted by T_M and T_M is finite. E_M and $\mathcal{A}_M$ represent respectively the identifiers of the set of events and actions the state machine M is involved in. Some actions involved in a state machine could also be events of the machines, that is $E_M \cap \mathcal{A}_M \neq \emptyset$.

3.4.1 State Configuration

$\mathcal{S}_M$ is partitioned into six disjoint sets as follows: simple states $S_{M(ss)}$, composite states $S_{M(cs)}$, initial states $S_{M(is)}$, final states $S_{M(fs)}$, choice states $S_{M(choice)}$ and subregions $S_{M(region)}$.

Every simple state or composite state contains a label, and may have an entry action, an exit action and do-activity. The label is compulsory for a state whereas the entry and exit actions and do-activity are optional. For state $K \in \mathcal{S}_M$, it is represented by the following functions:

- the state label is represented as *label* : $\mathcal{S}_M \rightarrowtail$ *LABEL*, where the total injective function specifies clearly that no two states within a state machine are allowed to have the same name.
- the entry action is represented as *entry* : $\mathcal{S}_M \nrightarrow \mathcal{A}_M$
- the do-activity is represented as *doActivity* : $\mathcal{S}_M \nrightarrow \mathcal{A}_M$
- the exit action is represented as *exit* : $\mathcal{S}_M \nrightarrow \mathcal{A}_M$

The partial functions used for *entry*, *doActivity* and *exit* model the fact that they are optional attributes for a state in a state machine.

For a state machine, we assume that there is a composite state that contains all other states in the state machine and we refer to it as the top state, S_{Mo}, with $S_{Mo} \in \mathcal{S}_M$.

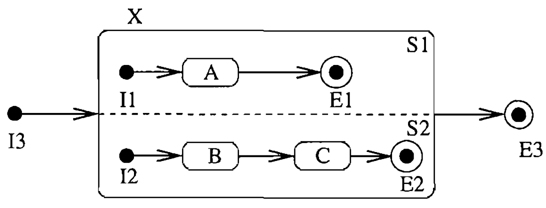


Figure 3.2: An Example of a composite-AND-state.

We define a binary function $IMM : S_M \nrightarrow S_M$ that maps each state to its immediate enclosing state. For example, given a composite state X with Y nested within it, we have $IMM(Y) = X$ (read as "the immediate enclosing state for Y is X"). Because the domain and range of the function are elements from the same set S_M, we run into a possibility where a state may be mapped to itself through IMM, which is obviously not correct. To avoid this from happening, IMM must define a tree. For this, we introduce a constraint where $IMM^+ \cap id(S_M) = \emptyset$ (see footnote [1,2]) . To satisfy the constraint, IMM^+ should not have any reflesive pair, i.e. $x \mapsto x$. Since $IMM \subseteq IMM^+$, the constraint will in turn force IMM not to contain any reflesive pair.

For convenience, we also define a function $ENCL$ where $ENCL : S_M \rightarrow \mathbb{P}S_M$. $ENCL$ may be defined in terms of IMM as $ENCL(x) = IMM^+[\{x\}]$. Due to the constraint introduced earlier, $ENCL$ satisfies $\forall x \cdot x \notin ENCL(x)$.

The composite states are partitioned into two sets: $S_{M(cos)}$ which refers to the composite-OR-states and $S_{M(cas)}$ for the composite-AND-states. An OR state contains substates that are OR-ed together so that only one substate can be active at a time, while an AND-state contains subregions that are AND-ed together, so that when the state is entered, all the subregions become active at the same time. Each composite-AND-state is divided into a finite set of subregions with each separated from the others by a dotted line. A subregion may contain substates that consist of states in S_M. Figure 3.2 shows an example [3] of a composite-AND-state X which consists of two subregions *S1* and *S2*, with *S1* containing substates *I1*, A and *E1*, and *S2* containing *I2*, B, C and *E2*. We define $S_{M(region)}$ as a global set which contains all the subregions of all the composite-AND-states in the state machine M, e.g. $S1, S2 \in S_{M(region)}$. To locate the AND-state in which a subregion resides, we use the function *IMM*. For instance, *IMM*$(S1) = X$. Similarly, we use the same function to determine the subregion to which a substate belongs to, e.g. *IMM*$(A) = S1$.

The hierarchy of various states found in a state diagram may be described using a tree. Figure 3.3(a) shows an example of a state machine which the hierarchy of its states are represented by a tree in Figure 3.3(b). The *root* node a represents the top state a of

[1] IMM^+ is the transitive closure of IMM, e.g. $IMM^+ = \bigcup_{i \geq 1} IMM^i$

[2] $id(S_M)$ is the identity function on S_M, e.g. $id(S_M) = \{(s,t) \mid s \in S_M \bigwedge t \in S_M \bigwedge s = t\}$

[3] When presenting an example, we will use the same identifier to refer to a state and the label attached to the state.

the state machine. The *parent vertices* labeled a, b and k with vertices below them correspond to the composite states a, b and k. Nodes n and o correspond to subregions n and o that reside in the composite-AND-state k. The *leaf* vertices with no children attached to them are states labeled h,e,f,j,l,c,d,m, p,q,r,s,t and u. In this figure, for instance, the set of enclosing state for t, that is $ENCL(t)$ is equal to $\{o, k, a\}$.

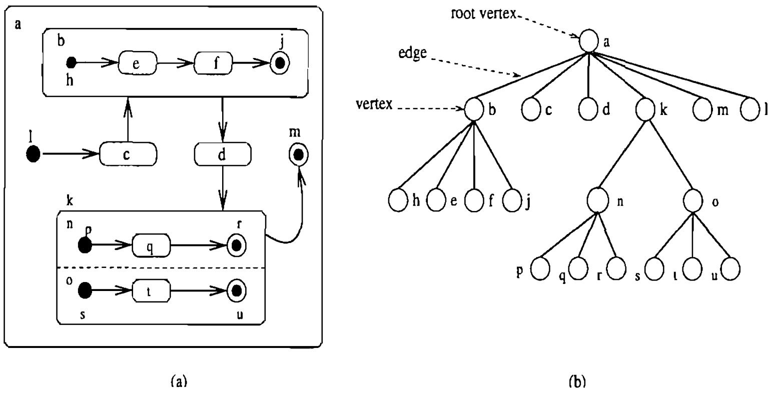


Figure 3.3: State hierarchy corresponds to a tree.

3.4.2 Transition Configuration

A transition identifier t where $t \in T_M$ (where M is the state machine) consists of a source state, a target state, a trigger event, a guard and an action list. All information except the source state and the target state is optional. Each transition t is represented by the following functions:

- the source state is represented by *source* : $T_M \rightarrow \mathcal{S}_M$.
- the target state is represented by *target* : $T_M \rightarrow \mathcal{S}_M$.
- the trigger event is represented by *event* : $T_M \rightarrow E_M$.
- the guard for the transition is a Boolean expression. Assuming we have a language $\mathcal{B}$ that describes the Boolean expressions, we have *guard* : $T_M \rightarrow \mathcal{B}$.
- the transition action is expressed as an ordered sequence of actions, e.g. $a_1; a_2; ...; a_n$. Thus, we have *action* : $T_M \rightarrow$ *seq* $\mathcal{A}_M$, where *seq* $\mathcal{A}_M$ refers to sequences which are made up of elements from the set $\mathcal{A}_M$.

In UML, there are two types of transition in a state diagram: transitions that are triggered by an explicit event and transitions that are triggered by an implicit event (i.e. completion event generated implicitly by the a state upon the completion of the state activity). We denote the set of explicitly triggered transitions as $T_{M(exp)}$ and the set of implicitly triggered transitions as $T_{M(imp)}$ such that $T_M = T_{M(exp)} \cup T_{M(imp)}$. Given the set of explicit events in M as $E_{M(exp)}$ and the set of implicit events as $E_{M(imp)}$ where $E_M = E_{M(exp)} \cup E_{M(imp)}$, we have $(t \in T_{M(exp)}) \Leftrightarrow (event(t) \in E_{M(exp)})$ and $(t \in T_{M(imp)}) \Leftrightarrow (event(t) \in E_{M(imp)})$.

Similarly, for a state K, the set of explicitly triggered outgoing transitions emanating from K is $T_{K(exp)}$, where $T_{K(exp)} = \{t \mid t \in T_{M(exp)} \bigwedge source(t) = K\}$. The set of implicitly triggered outgoing transitions emanating from K is $T_{K(imp)}$, where $T_{K(imp)} = \{t \mid t \in T_{M(imp)} \bigwedge source(t) = K\}$. Hence, $T_K = T_{K(exp)} \cup T_{K(imp)}$.

3.5 Basic Concepts of Formalization

Our formalization is built on the foundation that each UML state is mapped to a CSP process and each UML event to a CSP event. When a state becomes active, it will wait for the next event to occur that will trigger a transition that brings the system to the next state. If the trigger event is an external event, we model it as a CSP event. We will start by first explaining some basic concepts that will be used in our formalization.

3.5.1 Single Transition

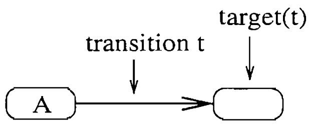


Figure 3.4: An example.

Suppose we have a state A (see Figure 3.4) and it has one outgoing transition t. For simplicity, assume there is no outgoing transition emanating from any enclosing state of A. When A becomes an active state in the system being modelled, it will wait for the next event that will trigger a transition out of the state through t. When the event becomes available, the transition will take place that brings the system to the next state. If the trigger event is an external event, we model it as a CSP event. We may express the behaviour of a state with a single outgoing transition as

$$A = event(t) \rightarrow target(t)$$

Otherwise if event(t) is implicit, we write

$$A = target(t)$$

Here, we make an important assumption that the system is always willing to proceed to the next state. Hence, we do not model A as $A = target(t) \sqcap STOP$. This aligns with our fundamental concept in modelling the multiple choice between implicit and explicit events (as we will see in the next subsection), where we assume that the implicit events will eventually take place if the explicit events do not occur.

3.5.2 Multiple Transitions

A state is allowed to have more than one outgoing transition. The decision on which transition to choose from will depend on what trigger event is being offered.

Let us have a state A which has more than one outgoing transition. For simplicity, assume there is no outgoing transition emanating from any enclosing state of A. If all the transitions are triggered by the explicit events, i.e. $\forall t \cdot t \in T_A \Rightarrow event(t) \in E_{M(exp)}$, the choice of transition is determined by the external environment. Hence, we formalize the choice using the external choice construct ($\Box$) in CSP as follows.

$$A = \Box_{t \in T_A}\ event(t) \rightarrow target(t)$$

If all the transitions are triggered by the state completion event which is implicit, i.e. $\forall t \cdot t \in T_A \Rightarrow event(t) \in E_{M(imp)}$, the choice of the transitions will be resolved by the process internally. As such, we model the choice as non-deterministic using the internal choice construct ($\sqcap$) in CSP as follows.

$$A = \sqcap_{t \in T_A}\ target(t)$$

When both implicitly and explicitly triggered outgoing transitions are present at state A, a problem arises to determine whether the process or the environment has the right to resolve the choice. To explain how we resolve this problem and hence arrive at a solution, we will use a simpler example to illustrate.

Suppose now state A has one explicitly triggered outgoing transition t_e and one implicitly triggered outgoing transition t_i. Upon the completion of the do-Activity within A, we are faced with two possibilities, (a) the environment offers *event(t_e)* which will trigger a transition out of A through t_e, or (b) state A produces a completion event which will trigger a transition out of A through t_i. Here, we are faced with one question: do we model the choice between the two transitions as determined by the external environment (and hence it is a deterministic choice) or the choice will be resolved internally by the process itself (and hence it is a nondeterministic choice)?

UML semantics does not specify the behaviour of this type of process. In view of this, we decided to adopt the interpretation offered by Roscoe's CSP semantics [59](p79-80). Roscoe proposed the following way to reason about this behaviour: when the hidden event becomes available, if the unhidden event does not occur, the hidden one will be carried out eventually. In this, Roscoe introduces the notion of "timeout", denoted as $\rhd$, whereby given $A \rhd B$, if A does not occur, B will be carried out eventually. Applying the interpretation to our problem, we may express the behaviour at state A as

$$A = (event(t_e) \rightarrow target(t_e))\ \rhd\ target(t_i)$$

The above expression states that if *event(t_e)* is not offered by the environment, the system will eventually take the transition to *target(t_i).*

3.5.3 Transition Guard

In UML, we may have guards attached to transitions and they are Boolean conditions that must be satisfied in order to enable a transition to take place. According to the OMG semantics, a guard is evaluated when an event instance is dispatched from the event queue. If the guard is true at that time, its corresponding transition will be fired. Otherwise, the transition is disabled. Based on [3], a guard condition may refer to the parameters from the triggering events or the attributes of the objects that belong to the state machine.

In the context of CSP, we may express a guarded transition using the CSP Boolean guard construct: $g\&P$ (read as *if g then P*). Having identified the construct in CSP, we may now formalize the UML transition guard using the CSP boolean guard, as follows. Using our running example, suppose state A has an explicitly triggered transition t with *guard(t)*, we may model the guarded transition as

$$A = guard(t) \;\; \& \;\; event(t) \rightarrow target(t)$$

If t is implicitly triggered, we have

$$A = guard(t) \;\; \& \;\; target(t)$$

3.5.4 State Actions and Transition Action

In UML, we may have actions attached to a transition or nested within a state such as entry action, exit action or do-activity. For simplicity, we are only going to consider modelling UML actions as CSP events.

Transition Action

More than one action may exist under a transition action component. For all the actions that are attached to a transition, when the transition is triggered, this will automatically execute the actions. The actions may be expressed in CSP as a sequence of events according to their linear order along the segments of the transition. They occur after the trigger event takes place.

We define a CSP process named ACTION(t) which defines the sequence of execution for all the actions belonging to a transition t when the transition is trigged.

Definition 1 *Given a transition* $t \in T_M$ *with* $action(t) \in seq\mathcal{A}_M$.
If $action(t) = \langle a_1, a_2, \cdots, a_n \rangle$, $ACTION(t) = a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow SKIP$
If $action(t) = \langle\rangle$, $ACTION(t) = SKIP$.

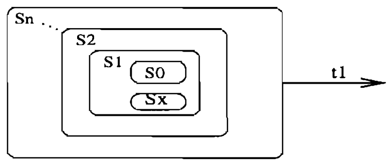


Figure 3.5: Transitions with multiple source.

State Entry and Exit Action

The well-formedness rule for our model allows at most one action to exist for each state entry action and exit action. For a simple or a composite state with entry action, the entry action is executed upon the state being entered. Similarly, the exit action is carried out when the state is exited, after the triggered event but before the transition action takes place. Below is the formalization for the UML actions.

Taking the transition and state actions together, we formalize their sequence of occurence as follows. Again, using our running example state A with outgoing transition t, suppose *action(t)* $= a_1; a_2; ...; a_n$. If t is explicitly triggered, then

$$A = entry(A) \rightarrow event(t) \rightarrow exit(A) \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow target(t)$$

Otherwise if t is implicitly triggered, we write

$$A = entry(A) \rightarrow exit(A) \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow target(t)$$

In cases where we have a transition with multiple nested source states (see Figure 3.5), we consider the following. For a transition with multiple source states e.g. transition *t1*, the order in which the state exit actions are to be executed begins with the exit action of the innermost nested state which is currently active. This is followed by the exit action of the closest composite state that encloses the innermost active substate, and this rule applies recursively until the composite state from which the transition directly emanates from is reached. For our example in Figure 3.5, if the current active state is S_0, the execution sequence of the exit action is $exit(S_0) \rightarrow exit(S_1) \rightarrow exit(S_2) \rightarrow \ldots \rightarrow exit(S_n)$. If the current active state is S_x, the execution sequence becomes $exit(S_x) \rightarrow exit(S_1) \rightarrow exit(S_2) \rightarrow \ldots \rightarrow exit(S_n)$.

States Do-Activity

The do-activity for a state represents the execution of an interruptable sequence of actions that occurs while the state is active. The activity starts executing upon entering the state, following the entry action. If the activity completes while the state is still active, it will raise a completion event that triggers an exit out of the state through its implicitly triggered transition (if it is present). If the state is exited as a result of the firing of one of its outgoing transitions before the activity is complete, the activity is aborted prior to its completion. In our formalization, we model two important execution points of an activity: its beginning and its termination. We represent these two occurence as CSP events labeled *beginActivityName* and *endActivityName.*

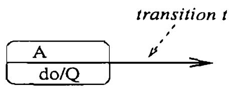


Figure 3.6: State with do-activity.

Assuming A (see Figure 3.6) is a simple state enclosed immediately by the top state S_{Mo} and A contains do-activity named Q. Suppose there is only one outgoing transition t emanating from A. If t is an implicitly triggered transition, the behaviour at state A may be described as

$$\begin{aligned} A = \quad & entry(A) \rightarrow beginQ \rightarrow endQ \\ & \rightarrow exit(A) \rightarrow target(t) \end{aligned} \tag{3.1}$$

If t is an explicitly triggered transition where $event(t) \in E_{M(exp)}$, we model the interruption of *event(t)* on do-activity Q using the CSP deterministic choice construct. For this, process A may be expressed as

$$\begin{aligned} A = entry(A) \rightarrow (\; (\; beginQ \rightarrow (\; & (endQ \rightarrow INT) \\ & \Box \; (INT) \\ &) \\ ) \; \Box \; (INT) & \\ ) & \end{aligned} \tag{3.2}$$

where $INT = event(t) \rightarrow exit(A) \rightarrow target(A)$.

In the above formalization, we can clearly see that event(t) is offered as a choice to interrupt the operation before, during and after the execution of activity Q. This conforms with the informal UML semantics defined by OMG [54].

One might wonder why we do not use the CSP interrupt operator ($\triangle$) to model the external events interrupting the execution of the do-activity. We will explain, using the running example as follows. Suppose we use the CSP interrupt operator to model the interruption of *event(t)* on do-activity Q. For this, we have

$$\begin{aligned} A = \quad & entry(A) \rightarrow (\; (beginQ \rightarrow endQ \rightarrow event(t) \rightarrow target(t)) \\ & \qquad \triangle(event(t) \rightarrow target(t)) \;) \end{aligned} \tag{3.3}$$

A closer inspection on the above equation reveals that if activity Q terminates successfully and proceeds to *target(t)*, the subsequent execution from *target(t)* may also be interrupted by *event(t)*. This is clearly not desired. To overcome this problem, we can replace *target(t)* with $RUN_{\{event(t)\}}$. Eq. 3.3 becomes

$$\begin{aligned} \text{AR} = \; & entry(A) \rightarrow ((beginQ \rightarrow endQ \rightarrow event(t) \rightarrow RUN_{\{event(t)\}}) \\ & \triangle(event(t) \rightarrow RUN_{\{event(t)\}})) \end{aligned} \tag{3.4}$$

where it synchronizes with

$$\text{RA} = event(t) \rightarrow exit(A) \rightarrow target(t) \tag{3.5}$$

And, we have

$$A = \text{AR} \underset{\{event(t)\}}{\|} \text{RA} \tag{3.6}$$

In this way, the interrupt operator will only have effect after *entryA* has occured up to and inclusive of *event(t)* in Eq.3.4. The subsequent execution modelled by Eq.3.5 is free from the interruption. $RUN_{\{event(t)\}}$ in Eq.3.4 is a special CSP process where $RUN_{\{event(t)\}} = event(t) \rightarrow RUN_{\{event(t)\}}$. $RUN_{\{event(t)\}}$ helps to avoid the parallel composition in Eq.3.6 from getting deadlocked when there are occurences of *event(t)* in the subsequent process.

We do not adopt CSP interrupt operator in our formalization because as demonstrated by the above example, the approach is rather cumbersome. It also appears to be misleading to use parallel contruct to model sequential execution. Based on these reasons, we decided on the approach that uses the CSP deterministic choice operator to model UML states with do-activity event.

Although we model both the transition event and action as a CSP event, the resulted CSP expression are different in the way they are formalized. To illustrate, see Figure 3.7. Using the concepts we have defined for multiple transitions earlier on, we may express

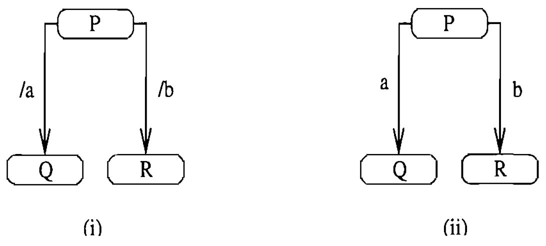


Figure 3.7: Transition Events and Actions

Figure 3.7(i) and (ii) in CSP as

$$(i) \quad P = (a \rightarrow Q) \sqcap (b \rightarrow R) \tag{3.7}$$

$$(ii) \quad P = (a \rightarrow Q) \Box (b \rightarrow R) \tag{3.8}$$

Both diagrams in Figure 3.7 model a different behaviour. In Figure 3.7(i), once state P completes its activity, it produces a completion event which implicitly chooses to trigger one of the two transitions. The action along the chosen transition is carried out when the transition is being executed. Eq.3.7 demonstrates this, whereby the internal choice shows that the decision between a and b lies in the process itself. In Figure 3.7(ii), once state P completes its activity, it waits for the environment to offer either a or b. In our formalization, this is modelled using an external choice (see Eq.3.8) which shows that the choice is upon the environment. From here, we can see how our formalization has faithfully model the behaviour of the state P in both cases.

The basic concepts we have discussed so far will be used as the fundamental blocks on which formal mapping definitions for non-composite states and composite states will be built.

3.5.5 Multiple State Exit Actions

Before we proceed further, we need to consider the issue where multiple exit actions are involved. Transitions $t1$ and $t2$ shown in Figure 3.8 are examples of transitions which may involve multiple state exit. The two transitions are similar in that they may be taken if A is the current state (e.g. the state where the system is residing at the moment) and this will result in a series of state exit actions being triggered from A to D. The difference between the two is that for Figure 3.8(a), the only possible current state where $t1$ can be executed is A whereas for Figure 3.8(b), the possible current state where $t2$ can be taken from could be either A, B or C. This is because in UML, an outgoing transition from an enclosing state is essentially a valid outgoing transition from each of its nested states, and in our case, Figure 3.8(b) is a simplified version of Figure 3.9, which has $t2$ originates from each state nested within C. Therefore, the exit actions involved when $t2$ is taken will depend on the current state when the transition is taken.

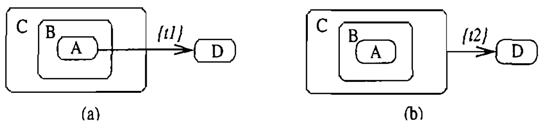


Figure 3.8: Multiple State Exit Transition.

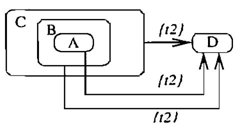


Figure 3.9: This state diagram is equivalent to the state diagram in Figure 3.8(b).

In order to keep a neat representation in the subsequent formalization, we define here a process named $EXIT(A, t)$ which represents a sequence of exit actions being executed in the correct order when transition t is carried out from the current state A. For example, looking at Figure 3.8(a), if the current state is A and if $t1$ is taken,

$EXIT(A, t1) = exit(A) \rightarrow exit(B) \rightarrow exit(C)$. Similarly for transition $t2$ in Figure 3.8(b), if the current state is A and if $t2$ is taken, $EXIT(A, t2) = EXIT(A, t1)$. However, if the current state is now B, $EXIT(B, t2) = exit(B) \rightarrow exit(C)$. From this, we can see that the state exit actions involving a transition depend on the current state from which the transition is taken. We now define formally the expression $EXIT(s, t)$ as follows.

Definition 2 *Given a current state $A \in S_M$ and a transition $t \in T_A$.*

$$\begin{aligned} EXIT(A,t) &= exit(A) \rightarrow SKIP \ ; \ exit(S_1) \rightarrow SKIP \ ; \\ &\quad exit(S_2) \rightarrow SKIP \ ; \ \ldots \ ; \ exit(S_n) \rightarrow SKIP \end{aligned}$$

$$\begin{aligned} \textit{where} \quad & \{S_1, S_2, \cdots, S_n\} = \{S \mid S \in ENCL(A) \bigwedge S \notin ENCL(target(t))\} \ \bigwedge \\ & S_1 = IMM(A) \ \bigwedge \ n \in NAT \ \bigwedge \ n > 1 \ \bigwedge \ S_n = IMM(S_{n-1}) \end{aligned}$$

Note if the current state A is a final state or a top state, it will not have any outgoing transition. Hence the second assumption of the definition, e.g. a transition $t \in T_A$ will not be true and $EXIT(A, t)$ does not exist for A.

3.6 The Mapping Function $\mathcal{H}$

To approach the formalization of the state diagram behaviour, we define a function $\mathcal{H}$ that maps the structure of a state machine to a CSP process. The function takes in two arguments, $\mathcal{H}(M, S)$ where M refers to a state machine and S refers to a state residing in the state machine. Note that a CSP process definition comprises a process name (N) and a process term (P), and it is written as N=P. Essentially, what $\mathcal{H}$ does is it will contruct a CSP term for every state in a state machine. Under our formalization, each state will give rise to a CSP process definition of the form $label(A) = \mathcal{H}(M, A)$ where A is a state identifier. As you can see, the state label will form the name of the CSP process, and $\mathcal{H}(M, A)$ will define the process term, which represent the behaviour of the state. For an example shown in Figure 3.10, state A will give rise to the equation $A = e \rightarrow B$ and state B will give rise to $B = f \rightarrow C$. We will explain further in the current and the next chapter how $\mathcal{H}$ defines the behaviour for different types of UML states.

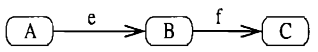


Figure 3.10: An example.

Unless specifically mentioned, we assume that we are dealing with a state machine named M from here onwards.

3.7 Formalization for Non-Composite State

In this section, we will formalize the UML non-composite states, i.e. those without any nested states.

3.7.1 Initial State

The well formedness rules defined by the OMG group [54](p2:157) says that an initial state can have at most one outgoing transition and no incoming transitions. This statement is rather vague as there is no mention if more than one initial state is allowed within the same level of a state. Consequently, we choose to allow more than one initial state to present within a state hierarchy level (which will be reflected in the definition for composite states later). This provides more freedom to the modelling style that can be supported by our formalization.

The outgoing transition that emanates from an initial state may be labeled, in which case the label event refers to the incident that initiates a system routine or creates an object (in an object-oriented context). If the transition is not labeled, the transition out of the initial state points to the first state to be encountered in an enclosing state. The formalization for an initial state is therefore

Definition 3 (Initial State) *Given an initial state A where*
$A \in S_{M(is)} \quad \bigwedge \quad t \in T_A$.
Recall the well formedness rule defined in Section 3.3, $card(T_A) = 1$.

1. *If* $t \in T_{A(imp)}$, $\mathcal{H}(M, A) \;=\; guard(t) \;\;\&\;\; (\; ACTION(t)\; ;\; target(t)\;)$.
2. *If* $t \in T_{A(exp)}$, $\mathcal{H}(M, A) \;=\; guard(t) \;\;\&\;\; (\; event(t) \rightarrow ACTION(t)\; ;\; target(t)\;)$.

3.7.2 Final State

Suppose we have a final state F nested within a set of enclosing states, *ENCL(F)*. Referring to both diagrams in Figure 3.11, $ENCL(F) = \{A1, A2\}$. If there is no transition emanating from any of the enclosing states, a transition to F represents a successful termination for the state machine where F and all the enclosing states reside, e.g. see Figure 3.11(a).

On the other hand, if there is at least one outgoing transition from one of the states in the set *ENCL(F)*, the entry to F denotes a successful termination for all the activity within the immediate enclosing state of F, followed by the occurrence of the event (either internal or external) that triggers the transition out of the enclosing state. In our running example in Figure 3.11(b), when the system reaches F, it denotes a successful termination for activity within *A1*. If neither *event(t1)* nor *event(t2)* are offered by the environment, the implicitly triggered transition to *S2* will be taken. Note that the implicitly triggered

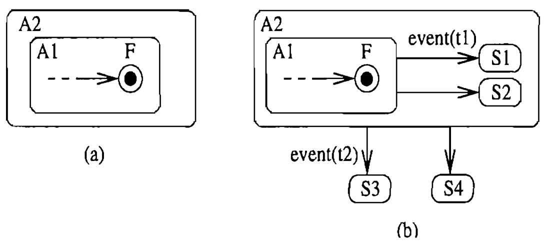


Figure 3.11: (a) A final state F without any outgoing transition where $\mathcal{H}(M,F) = SKIP$, (b) A final state F with outgoing transitions where $\mathcal{H}(M,F) = ((event(t_1) \rightarrow S_1)\Box(event(t_2) \rightarrow S_3)) \rhd S_2$.

transition to *S4* is not available at this point because *A2* has not reached its completion. As mentioned before, the system transition to F only denotes the completion of activity within *A1* and not *A2*.

Definition 4 (Final State) *Given a final state F where*
$F \in S_{M(fs)} \ \bigwedge \ (\forall S \cdot S \in ENCL(F) \Rightarrow S \notin S_{M(cas)})$.

1. *If there is no outgoing transition from any $ENCL(F)$,*

$$\mathcal{H}(M, F) = SKIP$$

2. *If there is at least one implicitly triggered outgoing transition from $IMM(F)$,*

$$\mathcal{H}(M, F) = \mathcal{F}_1 \rhd \mathcal{F}_2$$

3. *If there is no implicitly triggered outgoing transition from $IMM(F)$,*

$$\mathcal{H}(M, F) = \mathcal{F}_1$$

where

$$\mathcal{F}_1 = \Box_{X \in ENCL(F)} \Box_{t \in T_{X(exp)}} guard(t) \ \& \ (\ event(t) \rightarrow EXIT(F, t) \ ; \ ACTION(t) \ ; \ target(t) \ )$$
$$\mathcal{F}_2 = \sqcap_{u \in T_{IMM(F)(imp)}} guard(u) \ \& \ (\ EXIT(F, u) \ ; \ ACTION(u) \ ; \ target(u) \ )$$

The above definition is valid for different scenarios possible for F. We will show using a few examples of how this is true. Before that, we present below a few CSP algebraic laws which might be useful when we apply the definition to different scenarios. Note that these algebraic laws are also applicable to subsequent definitions. Assuming P is a CSP process,

Law 1	$x : \emptyset \rightarrow P(x) = STOP$
Law 2	$P \Box STOP = P$
Law 3	$STOP \rhd P = P$

To illustrate how we can apply Definition 4 to other scenarios, suppose we have a final state F shown in Figure 3.12(a) where $IMM(F) = X$. Since there is an implicitly triggered outgoing transition from $IMM(F)$, we use case 2 from Definition 4 to define the behaviour for F where $\mathcal{H}(M, F) = \mathcal{F}_1 \rhd \mathcal{F}_2$. However, when defining $\mathcal{F}_1$, since $T_{X(exp)} = \emptyset$, applying Law 1 to $\mathcal{F}_1$ will produce $\mathcal{F}_1 = STOP$. At this point, $\mathcal{H}(M, F) = STOP \rhd \mathcal{F}_2$ and using Law 3, $\mathcal{H}(M, F) = \mathcal{F}_2$.

We consider a different scenario in Figure 3.12(b). Since there is no implicitly triggered outgoing transition from $IMM(F) = X1$, we use case 3 to define for the behaviour of F, where $\mathcal{H}(M, F) = \mathcal{F}_1$. From the diagram, $ENCL(F) = \{X1, X2\}$. Because there is only one outgoing transition from $X1$ and no outgoing transition from $X2$, this gives rise to $\mathcal{F}_1 = (a \rightarrow Y) \Box STOP$. Applying Law 2 to the equation, we get $\mathcal{F}_1 = (a \rightarrow Y)$.

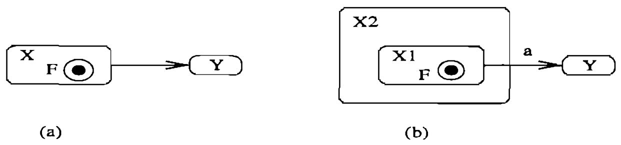


Figure 3.12: Some possible scenarios for a final state.

Note that Definition 4 only models those final states which are not enclosed by any AND-state (the constraint is imposed by the predicate $\forall S \cdot S \in ENCL(F) \Rightarrow S \notin S_{M(cas)}$ in the definitions). The reason for this will be clear when we consider final states which are enclosed by one or more AND-states in the next chapter.

3.7.3 Simple State

A simple state is a state which does not contain any substates. For a simple state A, the state may be exited in a few different ways as follows:

a. State A completes its activity and produces a completion event that triggers an outgoing transition through one of its implicitly triggered transitions (if there exists one) at the state border. Here, we would like to point out that the completion event will not have effect over any implicitly triggered transition at any of the enclosing states that enclose A, e.g. the completion event produced by X in Figure 3.13 can only trigger transition i but not j (note that both i and j with no labelled events are implicitly triggered transitions).

b. The external environment offers an event that triggers an outgoing transition at the border of state A. The activity within the state is then abandoned and the state is exited.

c. An outgoing transition at one of A's enclosing states, say K, is fired and this triggers an exit from K, A and all A's subsequent enclosing states up to but not including the enclosing state for the target state. E.g. when A is active (see Figure 3.14), if transition K-E is fired, it will trigger an exit for A, B, K and C, but not D since D is also the enclosing state for the target state of K-E.

Now we are ready to formalize the behaviour of a simple state, and the formalization only applies to those simple states where their immediate enclosing states are not a composite-AND-state.

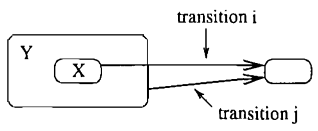


Figure 3.13: The completion event generated by X can only trigger transition i but not j.

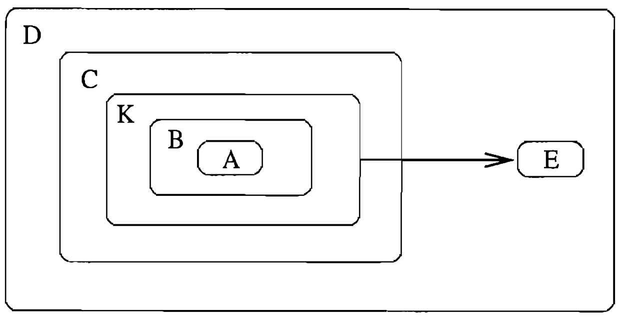


Figure 3.14: Transition K-E.

Definition 5 (Simple States) *Given a simple state A where* $A \in S_{M(ss)} \;\bigwedge\; (\forall S \cdot S \in ENCL(A) \Rightarrow S \notin S_{M(cas)})$.

1. *If A does not contain a do-activity, then*

   (a) *If* $T_{A(imp)} \neq \emptyset$, $\quad \mathcal{H}(M,A) = entry(A) \rightarrow (\; \mathcal{F}_1 \triangleright \mathcal{F}_2 \;)$

   (b) *If* $T_{A(imp)} = \emptyset$, $\quad \mathcal{H}(M,A) = entry(A) \rightarrow \mathcal{F}_1$

2. *If A contains a do-activity named $\mathcal{Y}$, then*

   (a) *If* $T_{A(imp)} \neq \emptyset$,

$$\mathcal{H}(M,A) = entry(A) \rightarrow (\; (begin\mathcal{Y} \rightarrow (\; (end\mathcal{Y} \rightarrow (\; \mathcal{F}_1 \triangleright \mathcal{F}_2 \;) \;) \;\Box\; \mathcal{F}_1 \;) \;) \;\Box\; \mathcal{F}_1 \;)$$

   (b) *If* $T_{A(imp)} = \emptyset$,

$$\mathcal{H}(M,A) = entry(A) \rightarrow (\; (begin\mathcal{Y} \rightarrow (\; (end\mathcal{Y} \rightarrow \mathcal{F}_1 \;) \;\Box\; \mathcal{F}_1 \;) \;) \;\Box\; \mathcal{F}_1 \;)$$

where

$$\mathcal{F}_1 = (\; \Box_{n \in T_{A(exp)}}\; guard(n) \;\&\; (\; event(n) \rightarrow EXIT(A,n) \;;\; ACTION(n) \;;\; target(n) \;) \;) \;\Box\; (\; \Box_{K \in ENCL(A)} \Box_{w \in T_{K(exp)}}\; guard(w) \;\&\; (\; event(w) \rightarrow EXIT(A,w) \;;\; ACTION(w) \;;\; target(w) \;) \;)$$

$$\mathcal{F}_2 = \sqcap_{m \in T_{A(imp)}}\; guard(m) \;\&\; (\; EXIT(A,m) \;;\; ACTION(m) \;;\; target(m) \;)$$

We would like to highlight the point that the behaviour of the simple state is formalized in such a way that after an event occurs, the do-activity be may interrupted and aborted, and the system will carry out and complete all its triggered actions, i.e. the source state exit action, transition action and the target state entry action, before responding to the next event.

For example, when a state X is active and if an event e occurs, the system will exit X, make a transition to the next state (say Y) and enter state Y. It is only after all these actions are completed and when the system has reached a stable state configuration[4] that it is able to respond to another event.

Again, note that Definition 5 only models those simple states which are not enclosed by any AND-state (the constraint is imposed by the predicate $\forall S \cdot S \in ENCL(A) \Rightarrow S \notin S_{M(cas)}$ in the definitions). We will consider simple states which are enclosed by one or more AND-states in the next chapter.

3.7.4 Choice State

The need to model data manipulation in UML arises when we attempt to model the choice state, whose role is slightly different from that of a normal state. To model a choice state, we first need to address the issue of modelling transition actions as value assignment so as to allow data manipulation on the choice conditions. What we mean by this is, suppose we have two cases as shown in Figure 3.15. Using the definitions we have formalized so

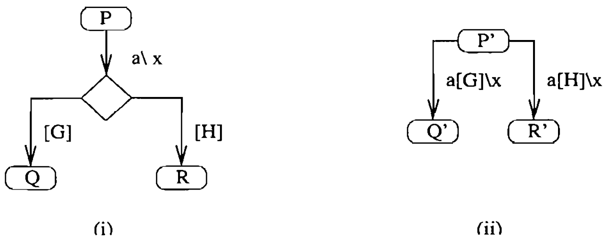


Figure 3.15: Comparing a choice state with a normal state.

far, we may express each of these cases in CSP as follows:

i. $P = a \rightarrow x \rightarrow (\ (G\&Q) \Box (H\&R)\ )$

ii. $P' = (G \ \ \& \ \ a \rightarrow x \rightarrow Q') \Box (H \ \ \& \ \ a \rightarrow x \rightarrow R')$

The CSP expressions for case (i) and (ii) will exhibit the same behaviour if we model the UML transition actions as CSP events, since action x does not change the value of the guards G and H. On the other hand, if the transition actions allow for value assignment, (i) and (ii) will exhibit a different behaviour as action x is now capable of changing the

[4]A system is said to reach a stable state configuration when it has completed its transition and entered a state in which it is residing.

data value in the guards, and hence in process P the choice between Q and R is determined after action x takes place. From this, it shows that there is a need to model UML transition actions as assignment statements in CSP in order to allow the function of the choice states to be distinguished from that of the normal states.

However, since we do not support UML identifier in our formalization model, we will not model the distinction between a simple state and a choice state. Having said so, we do not foresee any difficulty to model such distinction with the possible extension to support UML identifiers (which will be discussed in Section 3.8).

Choice State

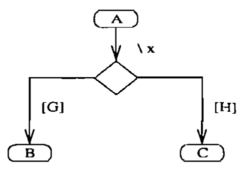


Figure 3.16: A choice state acting as a pseudostate between normal states.

The formalization for a choice state is similar to that of a simple state, except that it does not contain any state action or do-activity (refer Definition 5).

Definition 6 (Choice State) *Given a choice state A where*
$A \in S_{M(choice)} \quad \bigwedge \quad (\forall S \cdot S \in ENCL(A) \Rightarrow S \notin S_{M(cas)})$

i. If $T_{A(imp)} \neq \emptyset, \quad \mathcal{H}(M,A) = \mathcal{F}_1 \triangleright \mathcal{F}_2$
ii. If $T_{A(imp)} = \emptyset, \quad \mathcal{H}(M,A) = \mathcal{F}_1$

where

$$\begin{aligned}
\mathcal{F}_1 &= (\ \Box_{n \in T_{A(exp)}}\ guard(n)\ \ \&\ \ (\ event(n) \rightarrow EXIT(A,n)\ ;\ ACTION(n)\ ;\ target(n)\ )\ ) \\
&\quad \Box\ (\ \Box_{K \in ENCL(A)} \Box_{w \in T_{K(exp)}}\ guard(w)\ \ \& \\
&\qquad\qquad (\ event(w) \rightarrow EXIT(A,w)\ ;\ ACTION(w)\ ;\ target(w)\ )\ \ ) \\
\mathcal{F}_2 &= \sqcap_{m \in T_{A(imp)}}\ guard(m)\ \ \&\ \ (\ EXIT(A,m)\ ;\ ACTION(m)\ ;\ target(m)\ )
\end{aligned}$$

Note that Definition 6 only models those choice states which are not enclosed by any AND-state (the constraint is imposed by the predicate $\forall S \cdot S \in ENCL(A) \Rightarrow S \notin S_{M(cas)}$). We will consider choice states which are enclosed by one or more AND-states in the next chapter.

3.8 Possible Extensions to Support UML Identifiers

The UML identifiers may be categorized into two groups: event parameters and object attributes. Event parameters are a list of parameters passed to the event and they correspond to the parameters of an operation which belongs to a class. Object attributes, on the other hand, are mutable data held for an instance of a class. (Here, we assume there is only one instance associated to each class.)

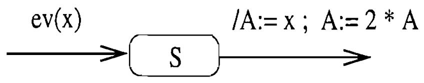


Figure 3.17: The accessibility scope for an event parameter in the State Diagram.

Before we attempt to formalize the UML identifiers in CSP, we first need to determine the scope of accessibility for these identifiers in the state diagram. In this, we propose the following:

i. **Object Attributes** Since a state diagram is a state machine that describes the behaviour of an object of a particular class, the attributes corresponding to the object may be accessed from any point within the diagram.

ii. **Event Parameter** We propose that an event parameter is only available to (a) the target state of the transition where the parameter is introduced via its event, and (b) the outgoing transitions from the target state. For example, the parameter x which is input via *ev* (see Figure 3.17) can only be accessed by S and all its nested states, and also the outgoing transitions from S.

   The restriction is formed on the basis of viewing event parameters as procedure input variables in the programming language. In programming terms, a procedure input variable is only available locally within the procedure. If we wish to make the parameter available to subsequent states, (e.g. other procedures in a program) we can assign the parameter value to the state machine global attribute, e.g. we assign x to the global attribute A in the running example. This mirrors the assignment of local copy of variable to a global variable which is readily accessible by the rest of the program in programming terms.

Also, we require that object attributes may be assigned with different values using UML actions or do-activity, but event parameters are not allowed to do so.

We will illustrate further what we have discussed so far using an example. In Figure 3.18, we have a choice state *C1* with two conditional branches. Notice that the state machine also contains three object attributes x, y and z and an event with parameter i. What the machine attempts to model is: it will take in an input value i from the user and increment it to 1 before using it for further operation. As discuss earlier, we do not

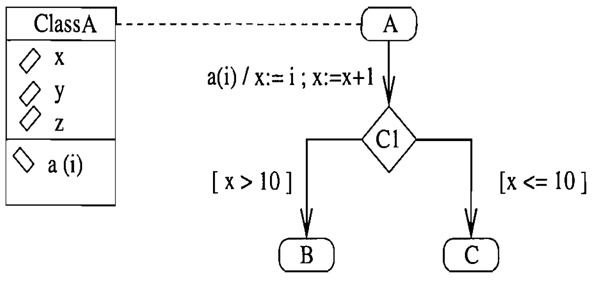


Figure 3.18: An example of a choice state.

allow any value assignment to an event parameter. As such, we need to assign i to the object attribute x before incrementing the value to 1. We may model the steps involved as follows:

i. When i is input, $A(x,y,z) = a?i \rightarrow A'(x,y,z,i)$

ii. To assign i to x, $A'(x,y,z,i) = A''(i,y,z,i)$

iii. To increment x by 1, $A''(x,y,z,i) = C1(x+1,y,z,i)$

Hence, at C1, we still have a modified value of x and an original copy of input i. We may formalize the behaviour of A and C1 in CSP as follows:

$$\begin{aligned} A(x,y,z) &= a?i \rightarrow C1(x+1,y,z,i) & (3.9)\\ C1(x,y,z,i) &= (\, x > 10 \;\; \& \;\; B(x,y,z)\,) \;\square\; (\, x \leq 10 \;\; \& \;\; C(x,y,z)\,) & (3.10)\end{aligned}$$

As you can see, the UML object attributes x, y and z are modelled as process parameters for each CSP process in the model. The event parameter i is represented as a CSP input in Eq.3.9 and it is only passed on to the next process *C1* using the process parameter i. As clearly expressed in the CSP model, we represent the assignment to parameter i as C1(x+1,...) instead of $a?i + 1$, where x+1 is a copy of the updated i with the value i remaining unchanged. Lastly, as mentioned earlier, i is only available to the immediate next state $C1$ and not B or C.

What we have discussed in this section shows the possibility of modelling UML identifiers in CSP. However, to keep our formalization simple, we have excluded the UML identifiers in our model.

3.9 Formalization for Composite OR-State

A composite state (OR-state or AND-state) refers to a state where there are other states nested within it. Hence, a composite state always satisfies

$$S \in S_{M(cs)} \Leftrightarrow \exists K \cdot K \in (S_{M(ss)} \cup S_{M(cs)}) \bigwedge S \in \mathrm{ENCL(K)}$$

In other words, we can say that a state S is a composite state if there exists some states that are enclosed by S. The top state S_{M_0} is an example of a composite state.

3.9.1 OR-State

As stated in the well-formedness rules defined in Section 3.3, no cross-border incoming transition is allowed and at least one initial state must present within each level of a composite-OR-state hierarchy to indicate the start point upon entry into the composite state. As such, a transition to a composite-OR-state represents a transition to an initial state within the first level of the composite state. For example, a transition to the composite state S in Figure 3.19 signifies an entry into S which leads to the commencement of activity within S starting from initial state I. Hence, we have $S = enterS \rightarrow I$. In situations where we have more than one initial state within the first hierarchical level of an OR-State, the choice of selecting an initial state is non-deterministic.

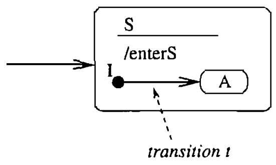


Figure 3.19: A composite OR-state

Definition 7 (Composite-OR-state) *Given an OR-state S where $S \in S_{M(cos)}$.*

$$\mathcal{H}(M,S) = \sqcap_{t \in Q}\ entry(S) \rightarrow source(t)$$

where $Q = \{\ t \mid t \in T_M \bigwedge source(t) \in S_{M(is)} \bigwedge source(t) \in IMM^{-1}[\{S\}]\ \}$.
Recall the well-formedness rule which specifies that an outgoing transition from an initial state must terminate at a state within the same level of the initial state. Therefore, t must also satisfy $target(t) \in IMM^{-1}[\{S\}]$

At this point, one might wonder if Definition 7 is sufficient to model the behaviour of an OR-state. How about the transitions that emanate from an OR-state, i.e. transition labelled with event z from *S1* in Figure 3.20? We have not forgotten about these types of transitions. Rather, they are being considered when we formalize for states (e.g. simple states, choice states and end states) which are nested within the OR-state. Therefore, to model the diagram in Figure 3.20, we have

$$\begin{aligned}
\mathcal{H}(M,S1) &= I1 \quad \text{(using Definition 7)}\\
\mathcal{H}(M,I1) &= S2 \quad \text{(using Definition 3)}\\
\mathcal{H}(M,S2) &= I2 \quad \text{(using Definition 7)}\\
\mathcal{H}(M,I2) &= A \quad \text{(using Definition 3)}\\
\mathcal{H}(M,A) &= z \rightarrow C \quad \text{(using Definition 5)}
\end{aligned}$$

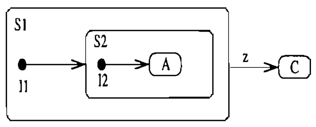


Figure 3.20: Simple state A nested in S1 and S2.

As you can see, the transition that emanates from *S1* is covered under the formalization for simple state A. As pointed out earlier in section 3.3, our model does not allow any *incoming* transition which crosses state boundaries. Therefore we do not allow design such as in Figure 3.21. With this restriction, we are able to avoid the problem of having to consider multiple entry actions with one transition. To keep our formalization simple, we do not support do-activity for OR-states.

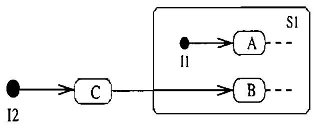


Figure 3.21: S1 has two possible start states: A or B.

Up to this point, we have defined the mapping function $\mathcal{H}$ for composite-OR-states and all their nested states. In the next chapter, we will deal with composite-AND-states.

Chapter 4

Formalizing UML State Diagrams in CSP (Part 2)

4.1 Formalization for Composite AND-state

Based on the OMG informal semantics [54](p2-163), whenever a composite-AND-state is entered, all its subregions are entered. At least one initial state must be present at each of its subregions to indicate the start state for a particular subregion. If the incoming transition terminates at the border of the AND-state, each of its subregions is entered by default, i.e. through to its initial state. If the transition explicitly enters a subregion, this region is entered explicitly while the rest are entered by default. Due of the well-formedness rule defined in Section 3.3, we will not model the type of incoming transitions which cross the boundary of the AND-state and enter explicitly into a subregion. On another hand, whenever there is a transition out of the composite state from any substate of a subregion, it will trigger an exit out of all the subregions in the composite state simultaneously. Note that the cross-border transitions between subregions are not allowed in UML.

In general, we may view the behaviour of a subregion to be similar to that of an OR-state. Subsequently, an AND-state can be thought of as a set of OR-states running in parallel. As such, we may model the behaviour of the AND-state S in Figure 4.1 as

$$S = S1 \parallel S2 \tag{4.1}$$

Recall that the transition to an OR-state represents the transition to an initial state nested within its first hierarchical level (see section 3.9.1). Applying the same concept to the subregions in the AND-state, an entry to the AND-state signifies the simultaneous transition to all the initial states within the first hierarchical level of the subregions. So, we can also write $S = (I1 \sqcap I2) \parallel I3$.

For each of the nested states within a subregion, we may attempt to apply the definitions which we have defined in Section 3.7 to model their behaviour. For example, by

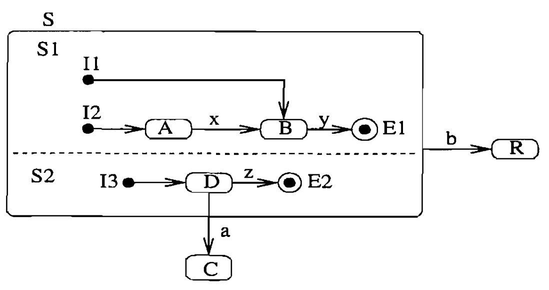


Figure 4.1: A composite-AND-state with transition crossing the AND-state border.

applying Definition 5 to the simple state A nested in $S1$, we get

$$\mathcal{H}(M, A) = (x \rightarrow B) \ \square \ (b \rightarrow R) \tag{4.2}$$

Just as an outgoing transition from an OR-state is a potential transition from any of its nested states (as discussed in Section 3.9.1), similarly, an outgoing transition from an AND-state may be taken by a nested state in any of its subregions. This explains why we have $b \rightarrow R$ as a choice in $\mathcal{H}(M, A)$.

However, because all the subregions in S are running in parallel, any transition that causes an exit from S, be it a transition emanating from the border of S (i.e. transition b) or crossing the border of S (i.e. transition a) will have potential effect on a nested state in every subregion. For this reason, we will also need to include the cross-border transition a as part of the potential behaviour of A, as shown below.

$$A = (x \rightarrow B) \ \square \ (b \rightarrow R) \ \square \ (a \rightarrow C) \tag{4.3}$$

This means that, when A is active, if state D is active at the same time and if transition a is taken, because a is a transition out of the composite AND-state, it will simultaneously trigger an exit out of A and subsequently out of subregion *S1*. This clearly shows that the informal semantics of UML state diagrams is not compositional, e.g. where AND-states are involved, we cannot define a semantics of regions independently of the context in which they appear. At this point, it is clear to us why the function $\mathcal{H}$ which we have developed for some nested states (e.g. simple, final and choice states) enclosed by OR-state(s) are not applicable if the nested states are enclosed by an AND-state. Hence, we will deal with nested states within an AND- state later in Section 4.1.2.

As we continue with our running example, remember that each of the subregions are composed in parallel with each other. Now suppose if D is to take on the transition to C which is situated outside the AND-state, the system will move out of the AND-state and as a result, the parallel composition should cease to exist. For this reason, we will need to terminate any activity that is still running within the parallel composition. To achieve this, we replace $(a \rightarrow C)$ with $(a \rightarrow SKIP)$ in Eq. 4.3. We do the same for transition labelled with event b which causes an exit out of the AND-state by replacing $(b \rightarrow R)$

with $b \rightarrow SKIP$ in Eq. 4.3. Eq. 4.3 then becomes

$$A = (x \rightarrow B) \ \Box \ (b \rightarrow SKIP) \ \Box \ (a \rightarrow SKIP) \tag{4.4}$$

Subsequently, we would also need to modify Eq. 4.1 to include the modelling of transition from S to any state outside S, i.e. state C or R, as

$$S = (S1 \underset{\{a,b\}}{\parallel} S2) \underset{\{a,b\}}{\parallel} ((a \rightarrow C) \ \Box \ (b \rightarrow R)) \tag{4.5}$$

What the above equation expresses is that: subprocess $S1 \underset{\{a,b\}}{\parallel} S2$ (which describes the behaviour of the AND-state S) will synchronize with $(a \rightarrow C) \ \Box \ (b \rightarrow R)$ over the set of events $\{a, b\}$, so that if a or b is to occur, $S1 \underset{\{a,b\}}{\parallel} S2$ will terminate and $(a \rightarrow C) \ \Box \ (b \rightarrow R)$ will carry on. Events a and b are exit events which will select the next state (either C or R) following the exit out of the AND-state.

However, the above model in CSP will deadlock if event a or b were to appear in the subsequent execution in process C or R, as it needs to synchronize with process $(S1 \underset{\{a,b\}}{\parallel} S2)$ which would have terminated by then. To overcome this problem, we use $RUN_{\{a,b\}}$(See footnote [1]) instead of $SKIP$ in Eq. 4.4.

Looking at Eq.4.5, one might raise the question of why a sequential operator is not used in place of the second parallel operator. Perhaps it is useful to first explain that a sequential composition between two processes is a mechanism of transferring control from a process which has *terminated successfully* to another process. However, in our case, we are attempting to model a behaviour whereby the control is passed from S to another state whenever an exit event from S is performed, and the subsequent state depends on which exit event occurs. This happens regardless of whether S has terminated successfully. If a sequential operator is used where process S is followed by R, i.e. $S; R$ (see Figure 4.1), when a occurs , it causes the termination of S. If the subsequent process (from state C onwards) terminates with a *SKIP*, the sequence operator will mean that the next state in sequence will be R. This is clearly not a correct behaviour for the diagram.

[1] RUN_X is a standard CSP process which can perform any event in the set X, i.e. $RUN_X = ?x : X \rightarrow RUN_X$.

Following our discussion so far, we now list out the CSP for our example in Figure 4.1. The nested simple states and final states are modelled as

$$\begin{aligned}
A &= (x \rightarrow B) \;\Box\; (b \rightarrow RUN_{\{a,b\}}) \;\Box\; (a \rightarrow RUN_{\{a,b\}}) \\
B &= (y \rightarrow E1) \;\Box\; (b \rightarrow RUN_{\{a,b\}}) \;\Box\; (a \rightarrow RUN_{\{a,b\}}) \\
E1 &= (b \rightarrow RUN_{\{a,b\}}) \;\Box\; (a \rightarrow RUN_{\{a,b\}}) \\
D &= (z \rightarrow E2) \;\Box\; (a \rightarrow RUN_{\{a,b\}}) \;\Box\; (b \rightarrow RUN_{\{a,b\}}) \\
E2 &= (b \rightarrow RUN_{\{a,b\}})
\end{aligned}$$

For the initial states, we use Definition 3 in the last chapter to produce

$$\begin{aligned}
I1 &= B \\
I2 &= A \\
I3 &= D
\end{aligned}$$

The behaviour for the AND-state is

$$S = (S1 \underset{\{a,b\}}{\parallel} S2) \underset{\{a,b\}}{\parallel} ((a \rightarrow C) \;\Box\; (b \rightarrow R))$$

And the behaviour for the subregions are

$$\begin{aligned}
S1 &= I1 \sqcap I2 \\
S2 &= I3
\end{aligned}$$

The formal definitions to generate CSP for the behaviour of an AND-state and its nested states will be discussed in the next few sections.

4.1.1 AND-state and Subregions

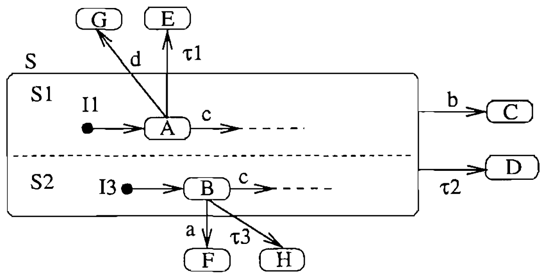


Figure 4.2: Transitions A-E, A-G, B-H, B-F, S-C and S-D may trigger an exit out of S.

Suppose we have an AND-state with six transitions that will cause an exit out of the states (see Figure 4.2). They are A-E, A-G, B-F, B-H, S-C and S-D. Among them, A-E, S-D and B-H are implicitly triggered. Notice that unlike the convention we have kept so far, here we use auxilliary labels to represent the corresponding implicit triggered events on the implicitly triggered transitions. The reason for this is because it is necessary in order to allow hidden events to be represented in the parallel composition that is used to synchronize the exit from all subregions. This will become clear when we look at the CSP representation for the behaviour of *S*, as follows:

$$\begin{aligned} S \;=\; & (\\ & (S1 \underset{\{a,b,d,\tau 1,\tau 2,\tau 3,c\}}{\|} S2) \underset{\{a,b,d,\tau 1,\tau 2,\tau 3\}}{\|} \\ & ((a \rightarrow F) \;\Box\; (b \rightarrow C) \;\Box\; (d \rightarrow G) \;\Box\; (\tau 1 \rightarrow E) \;\Box\; (\tau 2 \rightarrow D) \;\Box\; (\tau 3 \rightarrow H)) \\ &) \setminus \{\tau 1, \tau 2, \tau 3\} \end{aligned}$$

The CSP expression above suggests that the subregions *S1* and *S2* synchronize with each other over all the events (including the hidden ones) that trigger an exit out of *S*. $\{a, b, d, \tau 1, \tau 2, \tau 3\}$ is the set of exit events involved. Moreover, they will also need to synchronize on the common events, i.e. event c which is common to both of them. The parallel composition $S1 \parallel S2$ which describes the AND-state behaviour in turn synchronizes with the next state selector over the exit events set $\{a, b, d, \tau 1, \tau 2, \tau 3\}$. Lastly, we hide the hidden events $\tau 1$, $\tau 2$ and $\tau 3$ from the environment. We generalize the formalization for the composite-AND-state as follows.

Definition 8 (Composite-AND-state) *Given a composite-AND-state S where $S \in S_{M(cas)}$. Assume each subregion i is denoted as S_i, $1 \leq i \leq N$.*

$$\mathcal{H}(M,S) = (\; P_N \underset{\mathcal{W} \cup \{\varepsilon\}}{\|} (\; \Box_{t \in (\mathcal{Q} \cup \mathcal{R})} (event(t) \rightarrow \varepsilon \rightarrow target(t)) \;) \;) \setminus \mathcal{Z}$$

(Note: The parallel composition in the above equation ensures there is synchronization between the termination of S and the starting of another state outside S.)

where

N *is the total number of subregions in S.*

$P_n = S_n \underset{\mathcal{E}_n \cup \mathcal{W} \cup \{\varepsilon\}}{\|} P_{n-1}$ *for* $1 < n \leq N$ *where* $P_1 = S_1$

$\mathcal{E}_n$ *is a set of events found in* S_n *which is common to those in* S_1, S_2, …, S_{n-1}, *i.e.* $\mathcal{E}_n = \alpha S_n \cap (\alpha S_1 \cup \alpha S_2 \cup \ldots \cup \alpha S_{n-1})$.

Note: αS_x *denotes all the event alphabets found in the subregion* S_x.

$\mathcal{W}$ *comprises all the events (incl. hidden events) that trigger an exit out of S.*
$\mathcal{W} = \{event(t) \mid t \in (\mathcal{Q} \cup \mathcal{R})\}$

$\mathcal{Q}$ *is a set of transitions emanating from S.*
$\mathcal{Q} = \{t \mid (t \in T_M \bigwedge source(t) = S)\}$.

$\mathcal{R}$ *is a set of transitions emanating from a state within S and crossing the border of S.*
$\mathcal{R} = \{t \mid (t \in T_M \bigwedge S \in ENCL(source(t)) \bigwedge S \notin ENCL(target(t)))\}$.

ε *is a dummy event that is required to synchronize the exit from all current active substates when a transition out of the AND-state is triggered.*

$\mathcal{Z}$ *is a set of all the implicit events involved in the transitions that cause an exit out of S.*

$$\mathcal{Z} = \{event(z) \mid (z \in T_{M(imp)}) \bigwedge (S \notin ENCL(target(z))) \bigwedge ((source(z) = S) \bigvee (S \in ENCL(source(z))))\}.$$

The dummy event ε which has been added to Definition 8 is particularly important when we deal with nested states with exit actions. The dummy event ensures that the exit actions take place before the transition to a state outside the AND-state. An example presented later in Section 4.1.4 will demonstrate this. In our model, we treat the subregions as an ordered list of OR-states, which we traverse through from top to bottom or left to right, depending on how the subregions are arranged in an AND-state.

Another interesting point to note is the impact of including the set $\mathcal{W}$ (i.e. the set of all events that trigger an exit from S) in the synchronization set of the parallel composition of processes. If any one of the events in 'the set of all events that trigger an exit from S' occurs, it will trigger an exit out of each of the composite state's subregions and consequently a total exit out of the composite state S.

Definition 9 (Subregion) *Given a subregion S where $S \in S_{M(region)}$.*

$$\mathcal{H}(M,S) = \sqcap_{t \in Q}\ entry(S) \rightarrow source(t)$$

where $Q = \{\ t \mid t \in T_M \bigwedge source(t) \in S_{M(is)} \bigwedge source(t) \in IMM^{-1}[\{S\}]\}$.
Recall the well-formedness rule which specifies that an outgoing transition from an initial state must terminate at a state within the same level of the initial state. Therefore, t must also satisfy $target(t) \in IMM^{-1}[\{S\}]$.

4.1.2 Nested States

As pointed out earlier, we need to define the behaviour for simple states, choice states and final states in the case when at least one of their enclosing states is an AND-state. The formalization for an initial state remains the same, as in Definition 3, regardless of whether it is enclosed by an OR-state or an AND-state.

Simple State

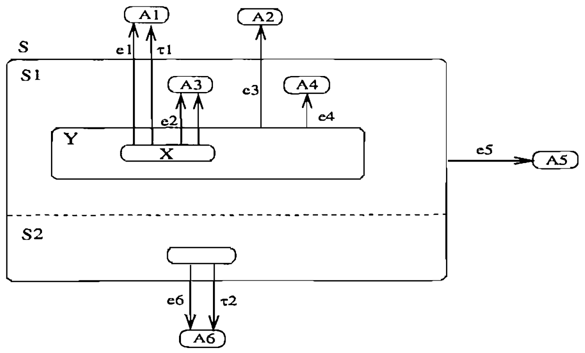


Figure 4.3: A simple state X enclosed by an AND-state S.

Suppose we have a simple state X (see Figure 4.3) where $ENCL(X) = \{Y, S1, S\}$. All the explicitly triggered transitions are those labelled with events ei, with $1 \leq i \leq 6$, and we attach the auxilliary label $\tau 1$ and $\tau 2$ to the implicitly triggered transitions that cause an exit out of the AND-state. As mentioned before, they are necessary so that if the transition labelled $\tau 1$ or $\tau 2$ is to be taken, the label can be used to synchronize the exit from the AND-state by all the active states from different subregions. The diagram is not complete and we only choose to show all the transitions that trigger an exit out of X together with the states necessary for our illustration. The behaviour for X can be expressed in CSP as below:

$$
\begin{array}{lll}
X = (& & \\
 & (\ (e2 \rightarrow A3) \, \Box \, (e4 \rightarrow A4) & \text{(Line 1)} \\
 & \Box \, (e1 \rightarrow RUN_F) \, \Box \, (e3 \rightarrow RUN_F) \, \Box \, (e5 \rightarrow RUN_F) & \text{(Line 2)} \\
 & \Box \, (e6 \rightarrow RUN_F) \, \Box \, (\tau 2 \rightarrow RUN_F) & \text{(Line 3)} \\
 & \Box \, (\tau 1 \rightarrow RUN_F)\ ) & \text{(Line 4)} \\
 &) \ \triangleright \ A3 & \text{(Line 5)}
\end{array}
$$

where $F = \{e1, e3, e5, e6, \tau 1, \tau 2\}$.

Line 1 refers to those explicitly triggered transitions which emanate either from X or from any enclosing states of X and terminate within S. Line 2 refers to those transitions that are similar to those in Line 1 except they terminate outside S. Line 3 models both explicitly and implicitly triggered transitions which emanate from a state nested in any of the subregions except *S1*, and terminate outside S. Line 4 refers to those implicitly triggered transitions emanating from X and terminating outside S. Line 5 refers to those that are similar to Line 4 except they terminate within S. Finally, F is the set of exit events from S.

Before we generalize the formalization, we assume a function $SUBREG : S_M \nrightarrow S_{M(reg)}$ which defines the most *immediate* subregion in which a state within an AND-state is nested. For example, $SUBREG(X) = S1$, and this implies $IMM(SUBREG(X)) = S$.

Definition 10 (Simple State Enclosed by An AND-State)
Given a simple state X where $X \in S_{M(ss)} \Rightarrow ENCL(X) \cup S_{M(cas)} \neq \emptyset$, and there exists an AND-state $S \in S_{M(cas)}$ where $S = IMM(SUBREG(X))$.

Let $\mathcal{R}$ be the set of implicitly triggered transitions emanating from X, and terminating within S (this will be defined later).

i. *If X does not contain any do-activity and $\mathcal{R} \neq \emptyset$ then*

$$\mathcal{H}(M,X) = entry(X) \rightarrow (\ \mathcal{F}_1 \ \triangleright \ \mathcal{F}_2\ )$$

ii. *If X does not contain any do-activity and $\mathcal{R} = \emptyset$ then*

$$\mathcal{H}(M,X) = entry(X) \rightarrow \mathcal{F}_1$$

iii. *If X contains do-activity $\mathcal{Y}$ and $\mathcal{R} \neq \emptyset$ then*

$$\mathcal{H}(M,X) = entry(X) \rightarrow (\quad (\ begin\mathcal{Y} \rightarrow (\ (end\mathcal{Y} \rightarrow \mathcal{F}_3) \ \Box\ \mathcal{F}_1\ )\ )\ \Box\ \mathcal{F}_1 \quad)$$

iv. *If X contains do-activity $\mathcal{Y}$ and $\mathcal{R} = \emptyset$ then*

$$\mathcal{H}(M,X) = entry(X) \rightarrow (\quad (\ begin\mathcal{Y} \rightarrow (\ (end\mathcal{Y} \rightarrow \mathcal{F}_1) \ \Box\ \mathcal{F}_1\ )\ )\ \Box\ \mathcal{F}_1 \quad)$$

$\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ are defined as

$$\begin{aligned}
\mathcal{F}_1 \quad &= \quad (\ \Box_{a\in\mathcal{P}} \quad guard(a) \rightarrow event(a) \rightarrow EXIT(X,a)\ ;\ ACTION(a)\ ;\ target(a)\ ) \\
&\quad \Box\ (\ \Box_{c\in(\mathcal{Q}\cup\mathcal{N}\cup\mathcal{V})} \quad guard(c) \rightarrow event(c) \rightarrow EXIT(X,c)\ ;\ ACTION(c)\ ;\ \varepsilon \rightarrow \\
&\qquad\qquad RUN_{\{\ event(u) | u\in(\mathcal{Q}\cup\mathcal{N}\cup\mathcal{V})\ \}\ \cup\ \{\varepsilon\}}\ ) \\
\mathcal{F}_2 \quad &= \quad \sqcap_{b\in\mathcal{R}} \quad guard(b) \rightarrow EXIT(X,b)\ ;\ ACTION(b)\ ;\ target(b) \\
\mathcal{F}_3 \quad &= \quad \mathcal{F}_1 \ \triangleright\ \mathcal{F}_2
\end{aligned}$$

where

$\mathcal{P}$	*is a set of explicitly triggered transitions, emanating from X or ENCL(X), and terminating within S, i.e.* $\mathcal{P} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) \in \{X\} \cup ENCL(X)) \bigwedge (S \in ENCL(target(t)))\}$
$\mathcal{Q}$	*is a set of explicitly triggered transitions, emanating from X or ENCL(X), and terminating outside S, i.e.* $\mathcal{Q} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) \in \{X\} \cup ENCL(X)) \bigwedge (S \notin ENCL(target(t)))\}$
$\mathcal{R}$	*is a set of implicitly triggered transitions, emanating from X, and terminating within S, i.e.* $\mathcal{R} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = X) \bigwedge (S \in ENCL(target(t)))\}$
$\mathcal{N}$	*is a set of implicitly triggered transitions, emanating from X, and terminating outside S, i.e.* $\mathcal{N} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = X) \bigwedge (S \notin ENCL(target(t)))\}$
$\mathcal{V}$	*is a set of both explicitly and implicitly triggered transitions, emanating from a nested state situated in a subregion other than SUBREG(X), and terminating outside S, i.e.* $\mathcal{V} = \{t \mid (t \in T_M) \bigwedge (IMM(SUBREG(source(t))) = S) \bigwedge (SUBREG(source(t)) \neq SUBREG(X)) \bigwedge (S \notin ENCL(target(t)))\}$
ε	*is a dummy event that is required to synchronize the exit from all current active substates when a transition out of the AND-state is triggered.*

As we have seen before in the previous chapter, $\mathcal{F}_1$ and $\mathcal{F}_2$ are expressions used to define scenario *i* to *iv* in order to make the definitions neater and hopefully more comprehensible. The set of implicitly triggered transitions emanating from a simple state X and terminating within S (denoted as $\mathcal{R}$ above), is dealt with in $\mathcal{F}_1$ and not $\mathcal{F}_2$.

Both scenario *i* and *iii* describe a simple state X nested within a composite state S and X has at least one implicitly triggered outgoing transition which terminates within S (e.g. $\mathcal{R} \neq \emptyset$). That is why we find $\mathcal{F}_2$ (and $\mathcal{F}_3$ which contains $\mathcal{F}_2$) in the definition for these two scenarios. On the other hand, scenario *ii* and *iv* describe a simple state X nested within a composite state S and X does *not* have any implicitly triggered outgoing transitions which terminate within S (e.g. $\mathcal{R} = \emptyset$). That is why we do not find $\mathcal{F}_2$ in the definition for these two scenarios.

Observe that *ENCL*(X) is not being considered in $\mathcal{R}$ and $\mathcal{N}$. This is because when X is active, and when X completes its activity, it may produce a completion event that triggers an exit out of X through one of its implicit transitions. However, this completion event will not have effect over any implicit transitions at any of X's enclosing states (simple because these states might not have terminated yet). That is why the implicit transitions for any states in $ENCL(X)$ are not being considered in $\mathcal{R}$ and $\mathcal{N}$ when defining the behaviour of X.

Choice State

The behaviour of a choice state nested within an AND-state is similar to that of a simple state explained in the previous subsection, except a choice state does not contain any state action.

Definition 11 (Choice State Enclosed by An AND-State)
Given a choice state X where $X \in S_{M(cs)} \Rightarrow ENCL(X) \cup S_{M(cas)} \neq \emptyset$, and an AND-state S where $S \in S_{M(cas)} \Rightarrow S = IMM(SUBREG(X))$.

i. If $\mathcal{R} \neq \emptyset$, $\mathcal{H}(M,X) = \mathcal{F}_1 \ \triangleright\ \mathcal{F}_2$
ii. If $\mathcal{R} = \emptyset$, $\mathcal{H}(M,X) = \mathcal{F}_1$ *(See below for definition of* $\mathcal{R}$*.)*

$\mathcal{F}_1$ *and* $\mathcal{F}_2$ *are defined as*

$$\begin{aligned}
\mathcal{F}_1 \quad = \quad & (\ \Box_{a\in\mathcal{P}} \quad guard(a) \rightarrow event(a) \rightarrow EXIT(X,a) \ ; \ ACTION(a) \ ; \ target(a) \ ) \\
& \Box \ (\ \Box_{c\in(\mathcal{Q}\cup\mathcal{N}\cup\mathcal{V})} \quad guard(c) \rightarrow event(c) \rightarrow EXIT(X,c) \ ; \ ACTION(c) \ ; \\
& \qquad\qquad \varepsilon \rightarrow RUN_{\{ \ event(u) | u\in(\mathcal{Q}\cup\mathcal{N}\cup\mathcal{V}) \ \} \cup \{\varepsilon\}} \ ) \\
\mathcal{F}_2 \quad = \quad & \sqcap_{b\in\mathcal{R}} \quad guard(b) \rightarrow EXIT(X,b) \ ; \ ACTION(b) \ ; \ target(b)
\end{aligned}$$

where

$\mathcal{P}$ *is a set of explicitly triggered transitions, emanating from X or ENCL(X), and terminating within S, i.e.*
$\mathcal{P} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) \in \{X\} \cup ENCL(X)) \bigwedge (S \in ENCL(target(t)))\}$

$\mathcal{Q}$ *is a set of explicitly triggered transitions, emanating from X or ENCL(X), and terminating outside S, i.e.*
$\mathcal{Q} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) \in \{X\} \cup ENCL(X)) \bigwedge (S \notin ENCL(target(t)))\}$

$\mathcal{R}$ *is a set of implicitly triggered transitions, emanating from X, and terminating within S, i.e.*
$\mathcal{R} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = X) \bigwedge (S \in ENCL(target(t)))\}$

$\mathcal{N}$ *is a set of implicitly triggered transitions, emanating from X, and terminating outside S, i.e.*
$\mathcal{N} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = X) \bigwedge (S \notin ENCL(target(t)))\}$

$\mathcal{V}$ *is a set of both explicitly and implicitly triggered transitions, emanating from a nested state situated in a subregion other than SUBREG(X), and terminating outside S, i.e.*
$\mathcal{V} = \{t \mid (t \in T_M) \bigwedge (\ IMM(SUBREG(source(t))) = S \ )$
$\bigwedge (SUBREG(source(t)) \neq SUBREG(X)) \bigwedge (S \notin ENCL(target(t)))\}$

ε *is a dummy event that is required to synchronize the exit from all current active substates when a transition out of the AND-state is triggered.*

Final State

Figure 4.4 shows two final states *X1* and *X2* nested within an AND-state. As before, all those transitions labelled with τ_i are implicitly triggered. Again, we only choose to show those constructs that are needed in our discussion. The main difference between *X2*

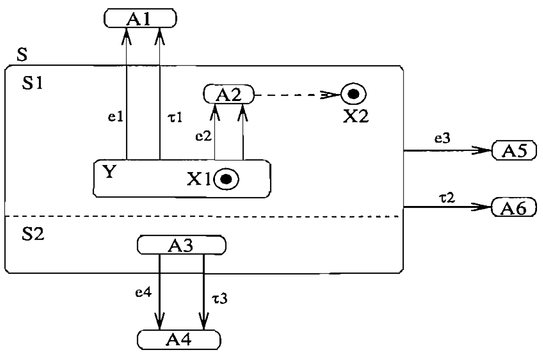


Figure 4.4: Final states *X1* and *X2* enclosed by an AND-state *S*.

and *X1* is that $IMM(X2)$ is a subregion whereas $IMM(X1)$ is not. This is significant because what it means is that when the system reaches *X2*, it shows that the subregion *S1* has terminated and the system can take on transition $\tau 2$ *if all other subregions have terminated as well.* Unlike *X2*, when the system reaches *X1*, *S1* has not terminated, hence the system cannot take on transition $\tau 2$ even though other subregions might have terminated.

Using the approach similar to that for the simple states and choice states, we now try to express the behaviour of these two final states in CSP. We will start with $X1$.

For $X1$, $ENCL(X1) = \{Y, S1, S\}$ and $IMM(X1) = Y$. The potential transitions out of *X1* include (a) all the explicitly triggered transitions emanating from ENCL(X1), (b) the implicitly triggered transitions emanating from IMM(X1), and (c) all the transitions emanating from a nested state in all subregions other than $S1$ and they terminate outside S.

$$\begin{aligned} X1 \;=\; (\;\; & (e2 \rightarrow A2) \\ & \Box\ (e1 \rightarrow RUN_F)\ \Box\ (\tau 1 \rightarrow RUN_F)\ \Box\ (e3 \rightarrow RUN_F) \\ & \Box\ (e4 \rightarrow RUN_F)\ \Box\ (\tau 3 \rightarrow RUN_F) \\ ) \;\; & \triangleright \;\; A2 \end{aligned}$$

where $F = \{e1, e3, e4, \tau 1, \tau 2, \tau 3\}$

For *X2*, $ENCL(X2) = \{S1, S\}$ and $IMM(X2) = \{S1\}$. The potential transitions out of *X2* include, (a) all the transitions emanating from ENCL(X2), and (b) all the transitions emanating from a nested state in all subregions other than $S1$ that terminate outside S. For (a), really we are only dealing with transitions emanating from S since there are no outgoing transitions from a subregion.

$$\begin{aligned} X2 \;=\; & (e3 \rightarrow RUN_E)\ \Box\ (\tau 2 \rightarrow RUN_E) \\ & \Box\ (e4 \rightarrow RUN_E)\ \Box\ (\tau 3 \rightarrow RUN_E) \end{aligned}$$

where $E = \{e1, e3, e4, \tau 1, \tau 2, \tau 3\}$

Also, observe that *X2* is able to respond to $\tau 2$ but not *X1*, for the reason as discussed earlier. We may now generalize the formalization as follows.

Definition 12 (Final State Enclosed by An AND-State)
Given a final state X where $X \in S_{M(fs)} \Rightarrow ENCL(X) \cup S_{M(cas)} \neq \emptyset$, there exists an AND-state $S \in S_{M(cas)}$ where $S = IMM(SUBREG(X))$.

i. If $\mathcal{P} \cup \mathcal{Q} \cup \mathcal{R} \cup \mathcal{M} \cup \mathcal{N} \cup \mathcal{V} = \emptyset$ *then* $\mathcal{H}(M, X) = SKIP$

ii. Else

(a) If $IMM(X) \notin S_{M(region)}$ *and* $\mathcal{R} \neq \emptyset$*, then*

$$\mathcal{H}(M,X) \; = \; \mathcal{F}_1 \rhd (\sqcap_{c \in \mathcal{R}} \quad guard(c) \rightarrow EXIT(X,c) \; ; \; ACTION(c) \; ; \; target(c) \;)$$

(b) If $IMM(X) \notin S_{M(region)}$ *and* $\mathcal{R} = \emptyset$*, then* $\mathcal{H}(M, X) = \mathcal{F}_1$

(c) If $IMM(X) \in S_{M(region)}$*, then*

$$\mathcal{H}(M,X) \; = \; (\; \Box_{a \in (\mathcal{M} \cup \mathcal{N} \cup \mathcal{V})} \quad guard(a) \rightarrow event(a) \rightarrow EXIT(X,a) \; ; \; ACTION(a) \; ; \; \varepsilon \rightarrow RUN_{\{event(u) | u \in (\mathcal{P} \cup \mathcal{M} \cup \mathcal{N} \cup \mathcal{V})\} \cup \{\varepsilon\}} \;)$$

$\mathcal{F}_1$ is defined as

$$\begin{aligned} \mathcal{F}_1 \; = \; & (\; \Box_{a \in \mathcal{Q}} \quad guard(a) \rightarrow event(a) \rightarrow EXIT(X,a) \; ; \; ACTION(a) \; ; \; target(a) \;) \\ & \Box \; (\; \Box_{b \in (\mathcal{P} \cup \mathcal{M} \cup \mathcal{V})} \quad guard(b) \rightarrow event(b) \rightarrow EXIT(X,b) \; ; \; ACTION(b) \; ; \\ & \qquad \varepsilon \rightarrow RUN_{\{event(u) | u \in (\mathcal{P} \cup \mathcal{M} \cup \mathcal{N} \cup \mathcal{V})\} \cup \{\varepsilon\}} \;) \end{aligned}$$

where

$\mathcal{P}$ *is a set of transitions emanating from ENCL(X) except S, and terminating* ***outside*** *S.*

$\mathcal{P} = \{t \mid (t \in T_M) \bigwedge (source(t) \in ENCL(X) - \{S\}) \bigwedge (S \notin ENCL(target(t)))\}$

$\mathcal{Q}$ *is a set of* ***explicit*** *transitions emanating from ENCL(X) except S, and terminating* ***within*** *S.*

$\mathcal{Q} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) \in ENCL(X) - \{S\}) \bigwedge (S \in ENCL(target(t)))\}$

$\mathcal{R}$ *is a set of* ***implicit*** *transitions emanating from IMM(X) (where IMM(X) is not a subregion) and terminating* ***within*** *S.*

$\mathcal{R} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = IMM(X))$

$\bigwedge (IMM(X) \notin S_{M(region)}) \bigwedge (S \in ENCL(target(t)))\}$

$\mathcal{M}$ *is a set of* ***explicit*** *transitions emanating from S.*

$\mathcal{M} = \{t \mid (t \in T_{M(exp)}) \bigwedge (source(t) = S)\}$

$\mathcal{N}$ *is a set of* ***implicit*** *transitions emanating from S.*

$\mathcal{N} = \{t \mid (t \in T_{M(imp)}) \bigwedge (source(t) = S)\}$

$\mathcal{V}$ *is a set of transitions emanating from a state situated in a subregion other than SUBREG(X), and terminating* ***outside*** *S.*

$\mathcal{V} = \{t \mid (t \in T_M) \bigwedge (IMM(SUBREG(source(t))) = S)$

$\bigwedge (SUBREG(source(t)) \neq SUBREG(X)) \bigwedge (S \notin ENCL(target(t)))\}$

ε *is a dummy event that is required to synchronize the exit from all current active substates when a transition out of the AND-state is triggered.*

4.1.3 Restriction on AND-states

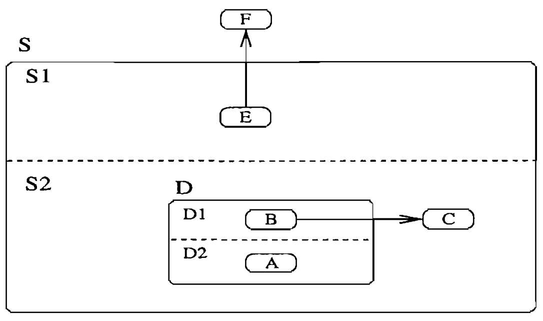


Figure 4.5: A nested AND-state D within an AND-state S.

Our formalization is able to model the behaviour of an AND-state with all kinds of nested states except a nested AND-state, i.e. we do not support AND-states such as S (see Figure 4.5) which has another AND-state D nested within it.

The reason we do not allow nested AND-state is to keep our formalization simple. To illustrate the level of complexity involves, suppose we are formalizing the behaviour for the current active state A in Figure 4.5. Based on Definition 10, we would need to include transition B-C in the behaviour of A since it also has the effect of triggering an exit of out A and $D2$. Similarly, we would have to include transition E-F as E-F is not only able to trigger an exit out of E, $S1$ and S but at the same time an exit from $S2$ and A which may be the current active state within $S2$. What this suggests is that the complexity of our formulation for active state A will increase with the increasing number of enclosing AND-states. As we have seen before, the lack of compositionality in a UML state diagram (i.e. the behaviour of a subregion depends on the behaviour of other subregions) has already made our existing formalization rather complicated. The introduction of enclosing AND-state will further increase the complexity involved. With this reason in mind, we have decided not to support nested AND-states in an AND-state.

Furthermore, we do not model do-activity for an AND-state. The exclusion of this feature has further helped to simplify our semantics definition for the AND-state.

4.1.4 An Example

In this example, we show how the definitions we have formalized in this chapter may be used to formalize the behaviour of a composite-AND-state S in Figure 4.6. Note that we use the auxilliary labels τ_1, τ_2 and τ_3 to represent the implicit events that may trigger an exit out of the composite state S. The following are a list of CSP expressions describing the behaviour of S.

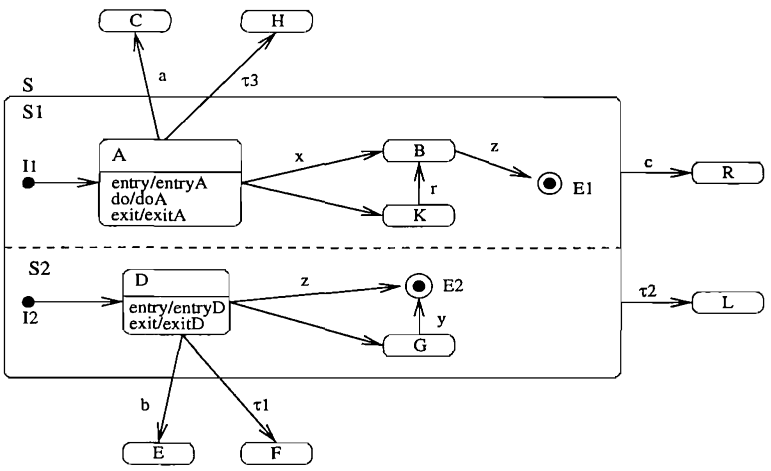


Figure 4.6: Formalizing the behaviour of a composite-AND-state.

$$
\begin{aligned}
\mathcal{H}(M,S) &= (\ (\ S_1 \underset{\{z,a,b,c,\tau_1,\tau2,\tau3,\varepsilon\}}{\|} S_2\ ) \\
&\qquad \underset{\{a,b,c,\tau_1,\tau2,\tau3,\varepsilon\}}{\|} \\
&\quad (\ (a \rightarrow \varepsilon \rightarrow C) \ \Box\ (b \rightarrow \varepsilon \rightarrow E) \ \Box\ (c \rightarrow \varepsilon \rightarrow R) \ \Box \\
&\quad (\tau_1 \rightarrow \varepsilon \rightarrow F) \ \Box\ (\tau_2 \rightarrow \varepsilon \rightarrow L) \ \Box\ (\tau_3 \rightarrow \varepsilon \rightarrow H)\ ) \\
&\quad)\setminus \{\tau_1, \tau2, \tau3\} \qquad [Definition 8] \\
\mathcal{H}(M,S1) &= I1 \qquad [Definition 9] \\
\mathcal{H}(M,S2) &= I2 \qquad [Definition 9]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{H}(M,I2) &= D \qquad [Definition 3] \\
\mathcal{H}(M,D) &= entryD \rightarrow (\quad (\ (z \rightarrow exitD \rightarrow E2) \ \Box\ (b \rightarrow exitD \rightarrow \varepsilon \rightarrow RUN_J) \\
&\qquad \Box\ (\tau_1 \rightarrow exitD \rightarrow \varepsilon \rightarrow RUN_J) \ \Box\ (c \rightarrow exitD \rightarrow \varepsilon \rightarrow RUN_J) \\
&\qquad \Box\ (a \rightarrow exitD \rightarrow \varepsilon \rightarrow RUN_J) \Box\ (\tau_3 \rightarrow exitD \rightarrow \varepsilon \rightarrow RUN_J) \\
&\qquad)\ \triangleright\ (exitD \rightarrow G) \\
&\quad) \qquad [Definition 10] \\
\mathcal{H}(M,G) &= (y \rightarrow E2) \ \Box\ (c \rightarrow \varepsilon \rightarrow RUN_J) \ \Box\ (a \rightarrow \varepsilon \rightarrow RUN_J) \\
&\quad \Box\ (\tau_3 \rightarrow \varepsilon \rightarrow RUN_J) \qquad [Definition 10] \\
\mathcal{H}(M,E2) &= (c \rightarrow \varepsilon \rightarrow RUN_J) \ \Box\ (\tau_2 \rightarrow \varepsilon \rightarrow RUN_J) \ \Box\ (a \rightarrow \varepsilon \rightarrow RUN_J) \\
&\quad \Box\ (\tau_3 \rightarrow \varepsilon \rightarrow RUN_J) \qquad [Definition 12]
\end{aligned}
$$

$$\begin{array}{rcl}
\mathcal{H}(M,I1) & = & A \qquad\qquad [Definition 3] \\
\mathcal{H}(M,A) & = & entryA \rightarrow (\ (begindoA \rightarrow ((enddoA \rightarrow S_1)\Box S_2))\ \Box\ S_2\ ) \quad [Definition 10] \\
\mathcal{H}(M,B) & = & (z \rightarrow E_1)\ \Box\ (c \rightarrow \varepsilon \rightarrow RUN_J) \\
 & & \Box\ (b \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (\tau_1 \rightarrow \varepsilon \rightarrow RUN_J) \qquad [Definition 10] \\
\mathcal{H}(M,K) & = & (r \rightarrow B)\ \Box\ (c \rightarrow \varepsilon \rightarrow RUN_J) \\
 & & \Box\ (b \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (\tau_1 \rightarrow \varepsilon \rightarrow RUN_J) \qquad [Definition 10] \\
\mathcal{H}(M,E_1) & = & (c \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (\tau_2 \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (b \rightarrow \varepsilon \rightarrow RUN_J) \\
 & & \Box\ (\tau_1 \rightarrow \varepsilon \rightarrow RUN_J) \qquad [Definition 12]
\end{array}$$

where

$$\begin{array}{rcl}
S_1 & = & (\quad S_2 \quad) \rhd (exitA \rightarrow K) \\
S_2 & = & \quad (x \rightarrow exitA \rightarrow B)\ \Box\ (a \rightarrow exitA \rightarrow \varepsilon \rightarrow RUN_J) \\
 & & \quad \Box\ (c \rightarrow exitA \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (b \rightarrow exitA \rightarrow \varepsilon \rightarrow RUN_J) \\
 & & \quad \Box\ (\tau_1 \rightarrow exitA \rightarrow \varepsilon \rightarrow RUN_J)\ \Box\ (\tau_3 \rightarrow exitA \rightarrow \varepsilon \rightarrow RUN_J) \\
J & = & \{a, b, c, \tau 1, \tau 2, \tau 3, \varepsilon\}
\end{array}$$

Using Definition 5, we get

$$\begin{array}{rcl}
\mathcal{H}(M,C) & = & STOP \\
\mathcal{H}(M,H) & = & STOP \\
\mathcal{H}(M,E) & = & STOP \\
\mathcal{H}(M,F) & = & STOP \\
\mathcal{H}(M,R) & = & STOP \\
\mathcal{H}(M,L) & = & STOP
\end{array}$$

One might ask how z triggers an exit out of the composite state S? First of all, z is an event common to both subregions *S1* and *S2*. This means that when z occurs, it will simultaneously trigger two transitions: *B-E1* and *D-E2*. When this happens, both subregions reach their respective final states and are *ready* to exit the composite state. At this point, they will wait for either event c or $\tau 2$ to occur which will subsequently activate a transition to exit the composite state S. If we take a look at the formalization for $E1$ and $E2$ above, e.g. $\mathcal{H}(M, E1)$ and $\mathcal{H}(M, E2)$, events c and $\tau 2$ are included in the possible behaviour for the final states.

4.2 Tool Support

4.2.1 U2CSP*v2*

We have developed a prototype translator U2CSP*v2*, which is essentially an enhanced version of U2CSP*v1* with emphasize on generating CSP for the UML state diagram. Compared to its previous version, U2CSP*v2* covers more state diagram constructs and these include nested states, OR-state and AND-state. Also, more in-depth treatment is given for a state to consider its entry action, exit action and do-activity. Beyond this, U2CSP*v2* is able to support different combinations of explicitly and implicitly triggered transitions emanating from a state. In essence, U2CSP*v2* is developed with the aim of seeking to understand the behaviour of the UML state diagram in a formal and consistent manner, as compared to U2CSP*v1* which emphasizes on representing CSP using UML constructs.

One shortcoming of the Rational Rose CASE tool is that it is not able to support the composite-AND-state and its subregions. In view of this, we seek an alternative representation using the fork and join constructs. We use a state to represent a subregion and place all the states/subregions between a pair of fork and join. Figure 4.7 shows how this is being done.

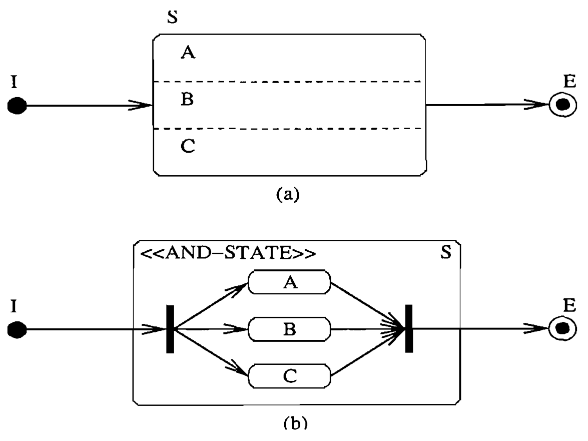


Figure 4.7: (a) A standard representation of the UML composite-AND-state S. (b) An alternative representation of state S in Rational Rose.

U2CSP*v2* supports this alternative representation but with a few limitations asserted as follows:

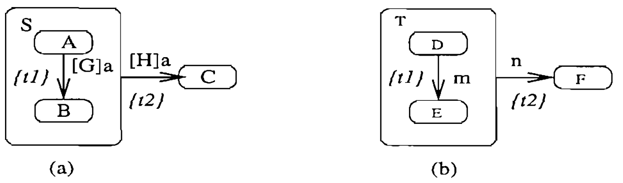
- A state diagram is only allowed to have a pair of fork and join, hence only one AND-state is supported.
- No event labels are allowed on the transition out of/into the fork/join.
- The trigger event which triggers an exit out of an AND-state is modelled using the event label on the outgoing transition from the join. Therefore, only one exit event may be modelled.

- The tool does not support any transition from a substate of a subregion to a state outside the AND-state.

4.3 Miscellaneous

4.3.1 Priority of Transition

The problem of transition conflict arises when more than one transition originating from the same source are enabled. The problem is categorised into two cases: those which are caused by a single event, and those which are caused by more than one event.



Note: {t1} and {t2} are labels given to the transitions for illustration purposes.

Figure 4.8: Transition conflict due to (a) a single event, (b) multiple events.

In the first case, a conflict arises when an event triggers more than one transition originating from the same source. In Figure 4.8(a), for example, if the currently active state is A, when event a is offered, two transitions: *t1* and *t2* are triggered, because both of them originate from the same source A. Since only one transition is allowed to take place, according to the OMG defined priority rule, the lower level transition *t1* is selected over the higher level transition *t2*.

We propose a possible extension to model this priority in CSP using transition guards. In the example, we introduce a guard $\overline{G} \land H$ to the lower priority transition *t2*. The formalization for A becomes

$$A = (\ G\ \&\ a \rightarrow B\ )\ \Box\ (\ (\ \overline{G} \land H\ )\ \&\ a \rightarrow C\ )$$

What the above expression says is this: When a is offered, whenever G is evaluated true (regardless if H is true), $a \rightarrow B$ is selected. Otherwise, transition $a \rightarrow C$ will be selected, provided its guard, H is true. A transition with no guard present is assumed to have a guard which is always evaluated to true.

To include this type of transition priority in our model, we would need to devise a way to model the different hierarchical level of a state. Lilius et. al. has addressed the issue by introducing the notion of covering in [41] which defines the hierarchy of the states in the formalization that can be used to resolve the transition conflict. We may adopt a similar concept, where we define a binary operator called *higher_than*, denoted as $\succ$ such that given two states S_1 and S_0, $S_1 \succ S_0$ iff $S_1 \in$ *ENCL(S_0)*. In our running example

(Figure 4.8(a)), since $source(t2) \succ source(t1)$, therefore, the firing priority will be given to the lower transition *t1*.

In the second case, as shown in Figure 4.8(b), if events m and n are offered when D is active, both transitions $t1$ and $t2$ are enabled and hence they will be in conflict. This problem only arises when more than one event is offered. As before, one may use a priority scheme to resolve the conflicts. However, we will not be able to formalize this in CSP since we do not know prior to the execution if the environment is going to offer only one or both events. For this, we cannot express in CSP the choice of events that is definitely going to be offered by the environment. (See footnote[2] on the notion of priority in CSP.) The probe primitive proposed by [46] might be able to deal with this. The probe primitive enables a communication channel between two processes to be probed to determine if the occurrence of an event is possible. With this, we may use the boolean guarded command so that if an event that triggers a higher priority transition is known to occur, we will select the transition over other enabled transitions. Having said so, however, we perceive that things will become rather complicated if we are to define the "probe" notion in our formalization. Moreover, "probe" is not part of the standard CSP and it is not supported by FDR.

4.3.2 Interlevel Transition

Interlevel transitions refer to those transitions which cross state boundaries. There are two types of interlevel transitions, as discussed below:

Cross Boundary Incoming Transition

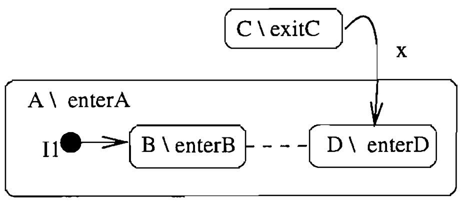


Figure 4.9: Cross Border Incoming Transition.

The well-formedness rules defined for our formalization model does not allow any incoming transition which crosses a state boundary (e.g. transition C-D in Figure 4.9). In this section, we will attempt to show you the intricacy involved if we allow such transitions to be supported by our model.

Looking at Figure 4.9, we may describe the behaviour of the OR-state using Definition 7 as $A = enterA \rightarrow I1$ where $I1 = B$, $B = enterB \rightarrow \ldots$, $D = enterD \rightarrow \ldots$ and so on

[2]The notion of *priority* mentioned by Roscoe in [59](p409-413) concerns with the internal events having higher priority over the external events in the context of timed CSP. This has no relation with the issue we are addressing here.

as defined by Definition 3 and 5. In this, we attach the entry action for each state at the beginning of the process term generated by the $\mathcal{H}$ function of the corresponding state. However, the structure of our formalization will be destroyed if we allow cross-border transition such as C-D to present in a state machine. To show you how, we may express the behaviour of C as $C = x \rightarrow exitC \rightarrow D$. However, because the transition is entering A before it reaches D, we will need to modify the expression for D defined earlier to include the entry action to A, where $D = enterA \rightarrow enterD \rightarrow \ldots$. To cater for this sort of transition, a tracing mechanism will be required in our model to trace the state entry of an incoming transition wherever more than one state entry is involved.

In order to avoid the complexity involved in developing such mechanism, we decided not to allow any incoming transition which crosses state border. As a result, we are able to develop a clean and tidy mapping process which is very handy when it comes to automating the process.

Cross Boundary Outgoing Transition

We deal with the problem involving cross-boundary *outgoing* transition using the process *ACTION(t)* which has been discussed in Section 3.5.4 under "Transition Action".

4.3.3 Multiple State Machines

So far, we have developed our formalization based on the operation of a single state machine. In this section, we will show how our model can be extended to model operation involving multiple state machines. The behaviour of a system consisting of multiple state machines is similar to that of a composite-AND-state comprising a few subregions, except the latter may have outgoing transitions that will trigger an exit out of all subregions in the composite state. For a process involving multiple state machines, they synchronize on events which are common to each other.

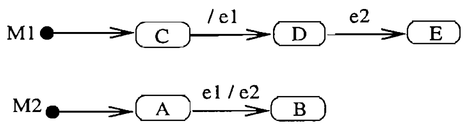


Figure 4.10: Multiple State Machines, M1 and M2.

Suppose we have a system with two state machines, M1 and M2 as in Figure 4.10. Using our formalization model we may describe Figure 4.10 in CSP as follows.

For M1:

$$\begin{aligned}\mathcal{H}(M1, M1) &= C \text{ where}\\ \mathcal{H}(M1, C) &= e1 \rightarrow D\\ \mathcal{H}(M1, D) &= e2 \rightarrow E\\ \mathcal{H}(M1, E) &= STOP\end{aligned}$$

For M2:

$$\begin{aligned}\mathcal{H}(M2, M2) &= A \text{ where}\\ \mathcal{H}(M2, A) &= e1 \rightarrow e2 \rightarrow B\\ \mathcal{H}(M2, B) &= STOP\end{aligned}$$

Since the two machines needs to synchronize on the common events *e1* and *e2*, we have

$$SYSTEM = M1 \underset{\{e1,e2\}}{\parallel} M2$$

4.4 Comparison with Other Work

The task of formalizing UML has been addressed using various available formal techniques, as we will discuss below. Most of these works are complementary, and they differ in approaching the task from different viewpoints and aims.

Much work has been carried out to give a formal semantics to the state diagrams in particular. The work by Engels et al. [19] involves translation from UML state diagrams to CSP but their aim is not to provide a formal semantics to the state diagrams. Rather, the reason they use CSP is to check the consistency between different UML diagrams that represent a model. They are interested to find out if the relationship of "*classDiagram B inherits classDigram A*" holds, will "*stateDiagram B inherit stateDiagram A*" holds as well. For this purpose, they derive the "*rule-based-notation*" which is used to map state diagrams into CSP, and use the CSP refinement assertion to check for the consistency.

Closely related to our work is that of Bolton & Davies [8] which presents a formal behavioural semantics for the UML activity diagrams. They use CSP to provide a syntactical interpretation of the activity graph but they have adopted an approach rather different from ours. For example, given a diagram in Figure 4.11, they translate the activity states as

$$\begin{aligned}P(playing) &= line1 \rightarrow (\ playing \rightarrow stop \rightarrow line2 \rightarrow P(playing) \quad ||| \quad P(playing)\ )\\ P(resting) &= line2 \rightarrow (\ resting \rightarrow line3 \rightarrow P(resting) \quad ||| \quad P(resting)\ )\\ \text{where} &\quad CombinedProcess = (\ P(playing) \underset{\{line2\}}{\parallel} P(resting)\ ) \setminus \{line1, line2, line3\}\end{aligned}$$

In their approach, each transition is given a name which is then used to synchronize the common transitions between two states. Because interleaving is used to model the process corresponding to each state, in order to eliminate divergence, they need to put an upper limit on the number of time a CSP event is allowed in any place. Readers who are interested to know how this is done may refer to [8]. Although the limit helps to ensure there is no divergence during model-checking, it has inevitably introduce limitations to their formalization model.

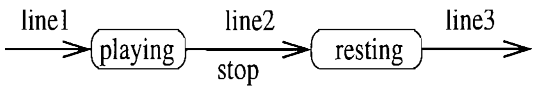


Figure 4.11: An example.

Latella et al. [37, 38, 47] develop a formal semantics for a subset of UML state diagrams which includes sequentialisation, parallelism, non-determinism and the transition priorities schemas. What they have done is to map the UML state diagrams to an intermediate format of the extended hierarchical automata, and then define an operational semantics for these automata based on Kripke structures[3]. They adopt the same approach as in our work which uses a hierarchical representation of the state diagrams. As opposed to our work which translates the state diagrams to CSP which is verified with FDR, their work in [37] translates the hierarchical automata to PROMELA that is verified using SPIN [32]. Also, their work in [26] attempts to verify the state diagrams in the JACK [9] environment.

The work by Mikk et al. [49, 50, 51] is loosely related to ours in that they formalize the semantics for the Harel's Statecharts. Similar to the work by Latella et al., their work translates Harel's Statecharts into Promela using extended hierarchical automata (EHA) [50] as an intermediate format. The EHA allows them to 'flatten' an interlevel transition by imposing restricted source and target to the transition. In their work, they propose two sets of mapping strategies, the first deals with the mapping from Statecharts to the EHA, and the second strategy map the EHA to Promela.

Lilius & Porres [41, 42, 43, 56] have also worked on formalizing the behaviour of the UML state diagrams. Their work claims to present a complete formalization of UML state machine semantics and provides an operational semantics for verifying the state diagrams. Their work takes on the same approach as ours which first formalizes the structure of the UML State machine, before defining the operational semantics of UML states based on the defined structure. In addition, they have also developed a tool called vUML [42] that translates the UML models into Promela language that may be fed into SPIN for model-checking. In order to cater for the model-checking using SPIN, one needs to include invalid states in the state diagram which serves as error claims [32] for the

[3]A nondeterministic finite state machine whose states are labeled with boolean variables, which are the evaluations of expressions in that state.

verification process in SPIN. In this, our approach is different. To verify a state diagram against certain properties, we model the properties in a separate UML diagram called the specification model. We then verfiy the correctness of our main model by carrying out refinement checking on the model against the specification model. (This will be demonstrated in Chapter 5)

Bӧrger et al. [6] describes the dynamics of UML state machines using the ASM (Abstract State Machine) rules [33]. Their work covers the event driven run-to-completion scheme, the sequential execution of entry/exit actions for nested states and also the simultanoues execution of do-activities for nested states that are currently active.

Sekerinski & Zurob [62, 63] and Ledang & Souquières [39] present algorithms that translate state diagrams to the AMN (Abstract Machine Notation) of the B method [2, 34]. In their B models, a state is represented as a value of an enumerated set type and a transition is represented as a change of value for a typed variable. The events and actions are modelled as B operations which modify the typed variables. Ledang & Souquières's algorithm involves two stages: the first stage uses an abstract machine to model the events, and the second uses the refinement/implementation which models the relationship between the events with their triggered transitions and actions. The refinement/implementation model is done by importing/extending the abstract machine. The presentation in B is rather intricate especially in terms of tracing the sequentiality and parallelism in a state diagram. Moreover, the complexity of the model increases when the notion of "include", "extend" or "import" of other B machines are involved. In comparison, CSP is much more straightforward and intuitive when it comes to model the dynamic behaviour of a state diagram. The refinement model in CSP is also more comprehensible as oppose to that of the B-model.

4.5 Conclusion

First of all, our formalization model adopts a synchronous interaction mode (we view the generation and consumption of an event to occur synchronously), as oppose to the asynchronous mode of the UML model. The discussion found in Section 3.2 justifies for this.

We started our work by first defining a structural model for a UML state machine. We then proceed by using this structural model to describe formally the behaviour of the UML structures in terms of CSP. In this, we have developed a set of formal definitions which define formally the behaviour of all the different states found in a UML state machine. Currently, our model is able to support initial state, final state, simple state, choice state, composite-OR-state and AND-state. Our model also includes constructs such as entry action, exit action, do-activity and transition action. Furthermore, we have proposed possible extensions to support UML identifiers and the transition priority scheme suggested by OMG.

CSP was chosen to model the UML state diagrams mainly because it is well-supported by model checking tools such as FDR. In addition, we find the CSP features to be appropriate in modelling various behaviours of a state machine. For example in our work, it is useful to have the CSP nondeterministic choice to model the choice between multiple implicitly triggered transitions, the deterministic choice for the multiple explicitly triggered transitions, the time-out feature for states with both explicitly and implicitly triggered outgoing transitions and the choice operator for the interrupt occurrence in UML. The CSP interrupt operator is not used in our work to model the interrupt behaviour in UML (i.e. the occurrence of an external event which interrupts a state do-activity) because we found using them to be cumbersome, as shown in Section 3.5.4 under "*State Do-Activity*".

To model the implicit events in UML, e.g. the state completion events that trigger the implicitly triggered transitions out of a state, our model has adopted two approaches. They are listed as follow:

i. We *do not* model explicitly a completion event by representing it using a "*tau_event*" in CSP. Instead, we express, for example the implicitly triggered transition from P in Figure 4.12 as $P = Q$. If there are more than one implicitly triggered transition available, we use the CSP nondeteministic choice to model the choice between them. If a state has a choice between an explicitly and an implicitly triggered transitions, i.e. state Q, we use a CSP timeout operator ($\rhd$) to model the choice, where $Q = (a \rightarrow R) \rhd S$. This is equivalent to writing $Q = (\ (a \rightarrow R) \ \Box \ (\tau \rightarrow S)\ ) \ \backslash \ \{\tau\}$. This method is used in most of the definitions we have seen in the model.

ii. We model explicitly a completion event by representing it using a "*tau_event*" in CSP. In this, a CSP hiding operator is used ($\backslash$) to hide the "*tau_event*". The need for this method arises when we model the behaviour of the AND-states and their nested states. In an AND-state, a τ event is used to represent an implicitly triggered exit event from the state so that it can be used later to synchronize the exit from the AND-state to the subsequent next state. (See Section 4.1.1)

We have preferred method (i) over (ii) for it makes our definitions simpler and less clutter by not having to model explicitly the implicit events. However, when we start to formalize for AND-state, we realize that method (i) is not sufficient and we are forced to switch to method (ii) instead. In doing so, we have decided to keep method (i) for all other definitions, bearing in mind that the inconsistency of methods used to represent UML in CSP will not actually affect the performance of our ultimate goal, that is to model-check UML state diagrams using FDR.

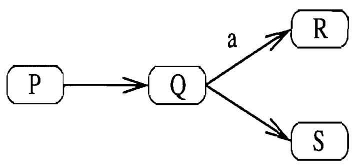


Figure 4.12: Transitions triggered by implicit events.

Our formalization is designed to be general and hence it does not restrict to any specific domain application. At the same time, we have developed a prototype translator, U2CSP*v2*, which is an extension to U2CSP*v1* to generate CSP automatically from the state diagrams which is then used to model-check the diagrams in FDR.

In order to keep our formalization simple and straighforward, we have defined a set of well-formedness rules (in Section 3.3) to constrain our work to a subset of UML state diagrams. Moreover, our formalization does not support

i. do-activity for OR-states and AND-states, and
ii. AND-state with nested AND-states.

Chapter 5

Formal Reasoning About The UML State Diagrams with FDR

In this chapter, we demonstrate how state diagrams are used for the design of a system, and how CSP/FDR can be used to verify the correctness for each design phase and assist in the design process. For each case study, the UML design is first translated into CSP using U2CSP*v2* based on the formalization defined in Chapter 3 & 4. The CSP generated by U2CSP*v2* is then analysed using FDR.

5.1 Compact Disc Player

5.1.1 Specification

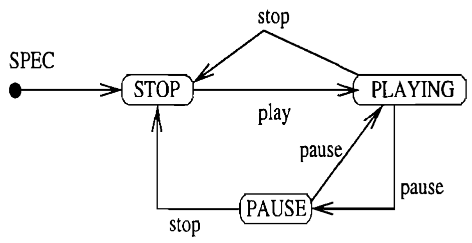
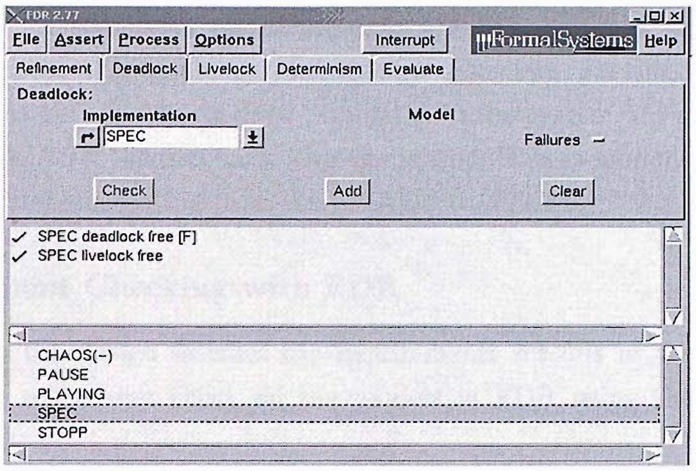


Figure 5.1: The specification model.

Figure 5.1 shows a simple CD player with three states: *STOP*, *PLAYING* and *PAUSE*. The events *play*, *stop* and *pause* model the actions performed by the user by pressing the various buttons on the player. This diagram captures the basic requirements of how a CD player is expected to work. Since this is a model of a single component that does not involve any event synchronization, it is obvious from the diagram that the design is deadlock free. Also, the absence of any impicit event should also imply that the model has no livelock problem. The checks using FDR (see Figure 5.2) confirms these observations.



✓ SPEC deadlock free [F]
✓ SPEC livelock free

CHAOS(-)
PAUSE
PLAYING
SPEC
STOPP

Figure 5.2: FDR confirms that the specification model is free of deadlock and livelock.

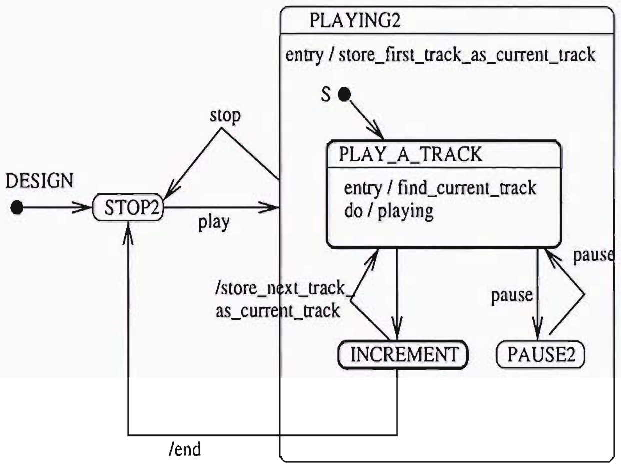


Figure 5.3: The design model.

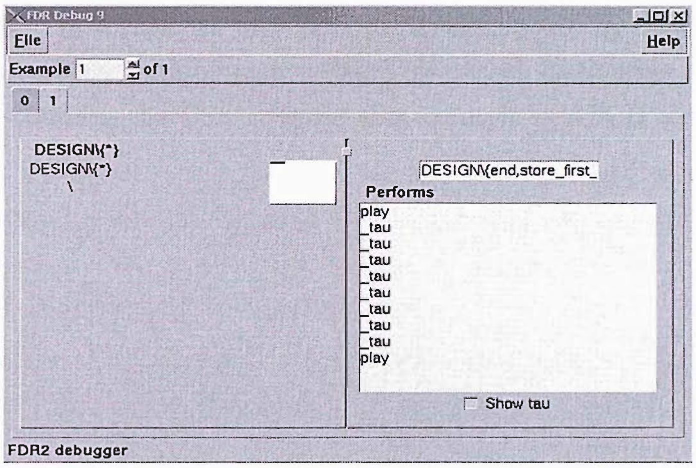
5.1.2 Design

Now suppose we want to include more details to the initial specification. For the purpose of translating into CSP later, we rename the states in the initial specification to *STOP2*, *PLAYING2* and *PAUSE2* in order to avoid repetition of process names. Referring to Figure 5.3, we add two extra states *PLAY_A_TRACK* and *INCREMENT* and we place these states together with *PAUSE2* within *PLAYING2*. At this point, we would like to present a design rule as suggested by Harel (previously mentioned in Section 1.5.3 under "*Basic Feature*"), where we will use transition events to model received signals (e.g. input from the environment) and transition actions to model generated signals (e.g. signals that are generated by the system internally). At *STOP2*, when the button *play* is pressed by the external user, the system enters *PLAYING2* in which *store_first_track_as_current_track* will be carried out by the player. At *PLAYING2*, the default first state is *PLAY_A_TRACK* which upon being entered into, the system will *find_current_track* and begin *playing*. During this time, *pause* may be pressed to temporarily stop the playing. A second *pause* will start the current track from the beginning again. After the current track finishes, an implicit completion event will trigger the transition from *PLAY_A_TRACK* to *INCREMENT*. At this state, two situations may be possible: if the current track is the final track on the CD, the process will end; else the system will store the next track as the current track and the cycle of playing the current track is repeated. At this level, we leave the two possiblitities as two choices resolved implicitly by the system. At any state when the system is in *PLAYING2*, the user may *stop* the player. This is modelled by the outgoing trasition labelled *stop* emanating from *PLAYING2* to *STOP2*.

5.1.3 Refinement Checking with FDR

To check whether the design satisfies the requirements set out in the specification, we carry out a trace refinement check on the models in FDR using the trace refinement assertion below .

$$\begin{aligned} SPEC \sqsubseteq_T DESIGN \backslash \{ & end, store\_first\_track\_as\_current\_track, find\_current\_track, \\ & beginPlaying, endPlaying, \\ & store\_next\_track\_as\_current\_track\} \end{aligned} \tag{5.1}$$


FDR Debug 9

File | Help

Example 1 of 1

0 | 1

```
DESIGN\{"}
DESIGN\{"}
     \
```

DESIGN\{end,store_first_

Performs

```
play
_tau
_tau
_tau
_tau
_tau
_tau
_tau
_tau
play
```

Show tau

FDR2 debugger

Figure 5.4: The counter example in FDR shows the extra trace (displayed under the column "Performs") found in the design model which is not specified in the specification model.

The results from FDR produces a counter example (see Figure 5.4) indicating an extra trace is identified in the design model that is not found in the specification model. The counter example suggests that the design model is capable of performing two successive *play* events, as a result of the user pressing the *play* button for the second time after the last track has been played which automatically stop the player. On the other hand, looking at the specification in Figure 5.1, after the user presses the *play* button, he/she needs to *stop* the player before the *play* button can be pressed again, and no consideration is given to the situation where the player may stop automatically after the last track is played and the user should be allowed to press the *play* button again after that. For this, we modify the specification model by inserting an implicitly triggered transition from *PLAYING* to *STOP*. This modification is highlighted in Figure 5.5. We repeat the trace refinement check on the design model against the specification and this time we obtain a successful check which suggests that the design model satisfies the safety requirements listed out in the specification model.

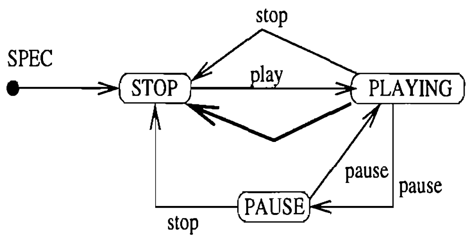


Figure 5.5: The modified specification model.

Furthermore, we run a failure and failure divergence refinement checks on the model. For this, we use the same assertion as in Equation 5.1 except that "T" is replaced with "F" and "FD" respectively. As expected, the models pass the failure check but fail in the failure divergence check because the system may diverge due to the implicit transition introduced */store_next_track_as_current_track* in the design. This happens because at this level, we choose to abstract away from specifying the bound for the number of tracks the CD player may play. In future, we may remove the divergence by introducing a variable to limit the number of tracks in playing a CD. This can be done using a transition guard

Appendix B.1 contains the CSP code for this example in ASCII form.

5.2 Barrier System

This is an example of a safety-critical distributed system which is made up of various components communicating with each other. The example is taken from a case study examined under the ABCD[1] (Automated Validation of Business Critical Systems with Component Based Designs) Project [70] . It involves the monitoring and controlling of a barrier system at the entrance of a protected area where hazardous activity is being carried out. A barrier system comprises four detectors, two barrier mechanisms, a control unit and two displays (see Figure 5.6). Each of these components communicate with one another. A detector detects the presence of a vehicle as it approaches the barrier, and reads information off the tag displayed on the front screen of the vehicle. The information gathered from the tag is then sent back to the control unit. A barrier mechanism receives instruction from the control unit in order to open or close the barrier. Once the barrier status is changed, it sends information back to the control unit to notify the change. The control unit receives information from both the detector and the barrier mechanism before issuing appropriate instruction to the barrier mechanism and the barrier display. Upon receiving request from the control unit, the display board will turn on either the GREEN or the RED light.

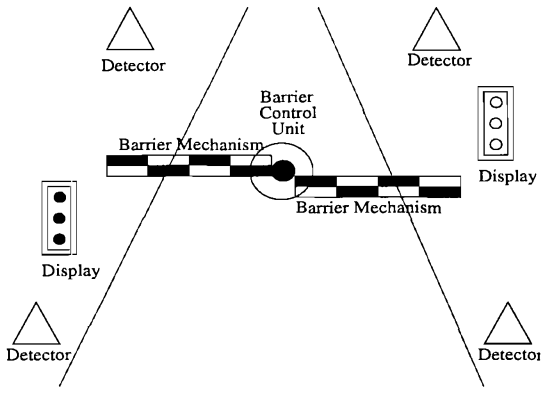


Figure 5.6: An example of a barrier system.

5.2.1 Modelling with State Diagrams and CSP

We will attempt the problem using a simplified version of a barrier system which consists of one detector, one barrier mechanism, one display and one control unit. We model the simplified barrier unit system as a state machine in the UML state diagram comprising a composite-AND-state with five subregions (see Figure 5.7 for the model built using the Rational Rose CASE tool). Each of the subregions models a specific component in the system.

Before we proceed to deal with the complexity of the system involving various components, we start by first drawing up the basic requirements which the system design

[1]Funded by EPSRC GR/M91013/01.

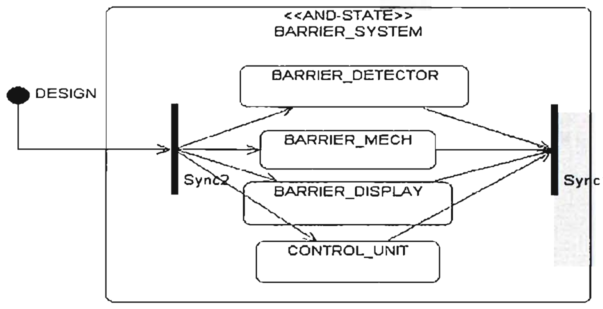


Figure 5.7: A barrier system with five parallel components.

needs to satisfy. These requirements are presented in Figure 5.8. Looking at the figure, when a *vehicleArrive*, its tag will be scanned and if it is a *validTag*, the vehicle may pass. At any point after *vehicleArrive*, the vehicle may choose to retreat regardless of whether the tag is valid. We model this requirement using a transition labelled "*vehicleRetreat*" emanating from state *A3*, where the transition may be taken at any state in *A3*. Observe that we model */validTag* and */invalidTag* as actions which are internal to the system, and when translated to CSP, the choice between them is expressed as $(validTag \rightarrow A6) \sqcap (invalidTag \rightarrow A7)$. This is a reasonable abstraction at the high level to leave the outcome of the tag processing to the system. In a future implementation when more tag information is considered, we may want to model the outcome as external events where the control is in the environment.

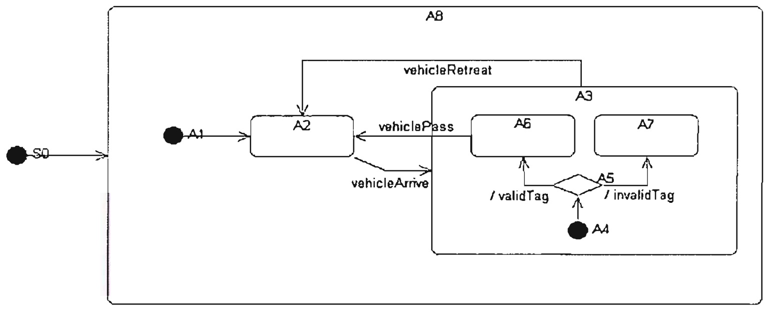
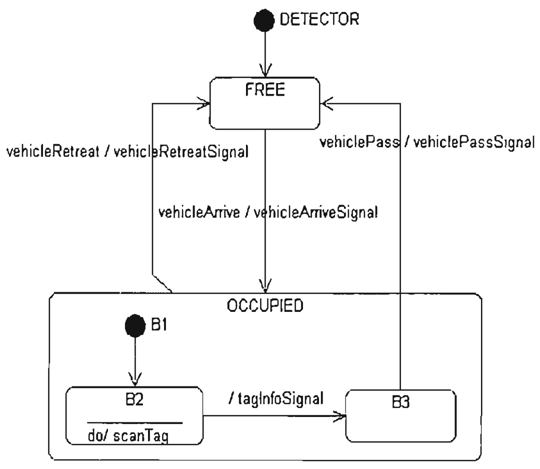


Figure 5.8: The basic requirements of a barrier system.

In the next few sections, we are going to look at the design of different components in the system. As before, we follow the design rule where we use a transition action to model a message generated and broadcasted by a component and a transition event to model an external message which is received and acted upon by a component. In this aspect, we would like to remind the readers that although our formalization models each transition event and action as a CSP event, the difference is highlighted in how we resolve the choice between them (refer section 3.5.2).



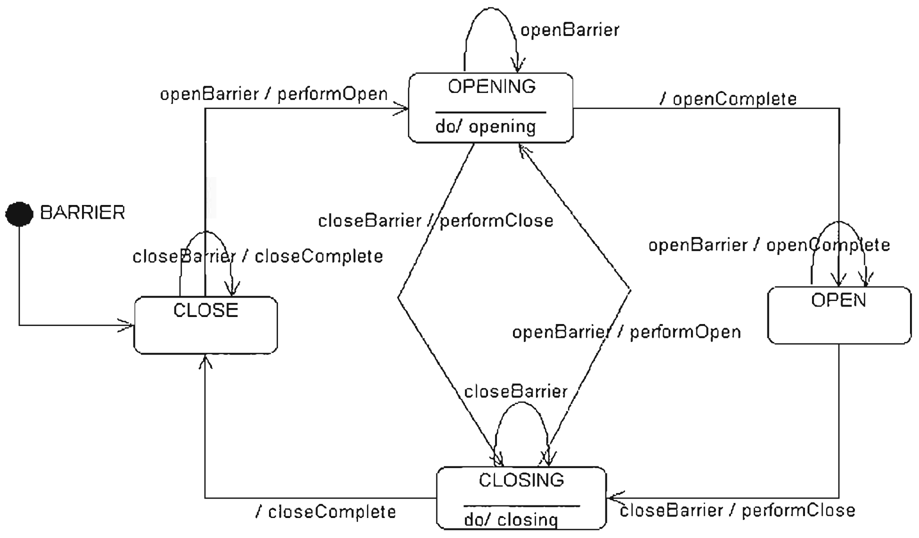
DETECTOR
FREE
vehicleRetreat / vehicleRetreatSignal
vehiclePass / vehiclePassSignal
vehicleArrive / vehicleArriveSignal
OCCUPIED
B1
B2
do/ scanTag
/ tagInfoSignal
B3

Figure 5.9: A Detector

Detector

Figure 5.9 describes the function of a detector in the barrier system. When a detector detects the presence of an approaching vehicle through the external event *vehicleArrive*, it will scan the tag on the vehicle for information. Since the scanning process takes time, it is modelled as a do-activity */scanTag*. While the car tag is being scanned, the vehicle might reverse and leave, modelled by the transition *vehicleRetreat* emanating from *OCCUPIED* to *FREE*. Otherwise, the detector will send the tag information signal back to the control unit for processing, e.g. */tagInfoSIgnal*. A vehicle may pass after its tag is sent to the control unit. Observe that whenever the external events *vehicleArrive*, *vehicleRetreat* or *vehiclePass* takes place, a corresponding signal is generated and sent to the control unit.

Barrier Mechanism



openBarrier
OPENING
do/ opening
openBarrier / performOpen
/ openComplete
BARRIER
closeBarrier / performClose
closeBarrier / closeComplete
openBarrier / openComplete
CLOSE
OPEN
openBarrier / performOpen
closeBarrier
CLOSING
do/ closing
/ closeComplete
closeBarrier / performClose

Figure 5.10: A Barrier Mechanism.

The role of the barrier mechanism (see Figure 5.10) is to control the position of the barrier. There are two possible states in which a barrier might reside: *CLOSE* and *OPEN*. Since the action of closing and opening the barrier takes time, they are modelled as do-activities *opening* and *closing* instead of atomic events. Hence, we have two intermediate states: *OPENING* and *CLOSING*.

At state *CLOSE*, when the barrier mechanism receives *openBarrier* from the control unit, it will issue an internal command called *performOpen* which will initiate the opening of the barrier. While the barrier is opening, the control unit may issue *closeBarrier* which will then trigger *performClose*. Once the opening of the barrier completes, the barrier mechanism issues a *openComplete* signal, denoted by the implicitly triggered transition from *OPENING* to *OPEN*.

Likewise at state *OPEN*, a *closeBarrier* command received by the barrier will trigger *performClose*. While the barrier is closing, the control unit may issue an *openBarrier* commmand which will trigger *performOpen*. Once the closing of the barrier completes, the barrier mechanism issues a *closeComplete* signal, denoted by the implicitly triggered transition from *CLOSING* to *CLOSE*.

Display

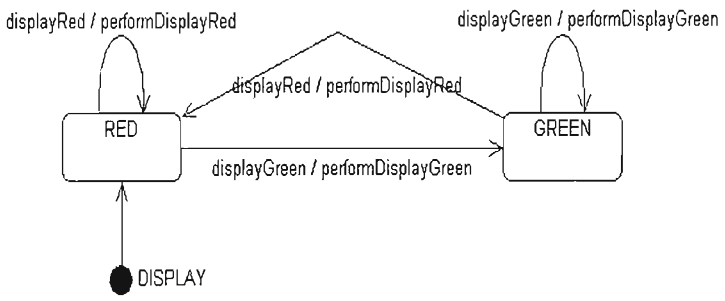

displayRed / performDisplayRed
displayGreen / performDisplayGreen
displayRed / performDisplayRed
RED
GREEN
displayGreen / performDisplayGreen
DISPLAY

Figure 5.11: A Barrier Display.

The barrier display has the simplest design out of all the other components. It provides an indication of the access permission given to the approaching vehicles. There are two states which a display may reside: *RED* or *GREEN*. The display receives instruction *displayRed* or *displayGreen* from the control unit and issues *performDisplayRed* or *performDisplayGreen*. Figure 5.11 shows the behaviour of the display.

Control Unit

The control unit is the mastermind behind the system and it is designed to receive input from the detector and the barrier mechanism before issuing instruction to the latter and the barrier display so that they may act upon the instruction. Looking at Figure 5.12,

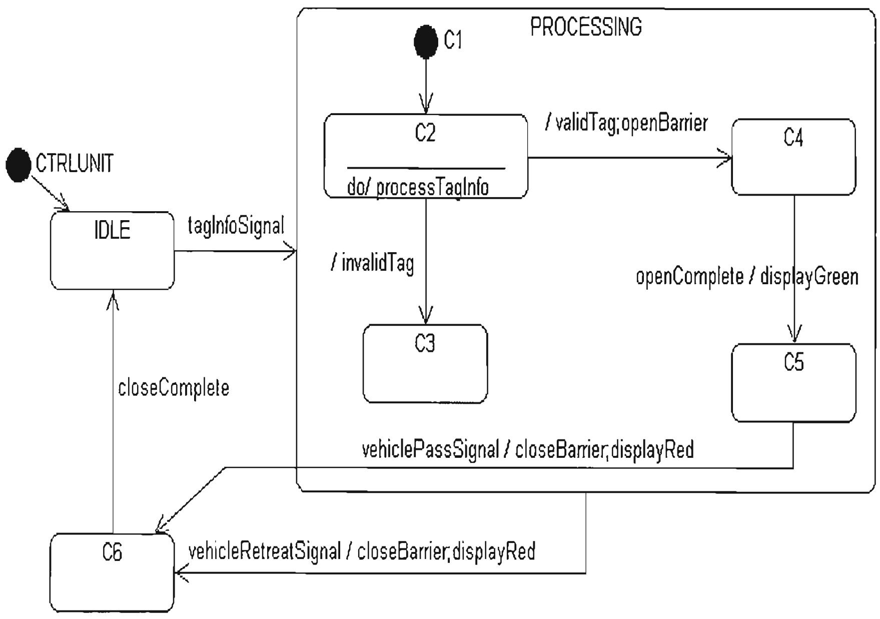

PROCESSING
C1
C2
do/ processTagInfo
/ validTag;openBarrier
C4
CTRLUNIT
IDLE
tagInfoSignal
/ invalidTag
openComplete / displayGreen
C3
C5
closeComplete
vehiclePassSignal / closeBarrier;displayRed
C6
vehicleRetreatSignal / closeBarrier;displayRed

Figure 5.12: A Control Unit.

there are two main states in which the control unit may reside: *IDLE* or *PROCESSING*. At *IDLE*, upon receiving the *tagInfoSignal*, the control unit will process the information where the outcome is determined internally by the system and thus issued with an internal action */validTag* or */invalidTag*. If the tag is valid, the control unit proceeds to issue the */openBarrier* command. Once the barrier is up, the control unit will be notified via the signal *openComplete* which upon being received, the control unit sends out */displayGreen* to the display. At this point, the vehicle may pass which will generate *vehiclePassSignal* that triggers control unit to issue */closeBarrier* followed by *displayRed*. At any state within *PROCESSING*, a vehicle may retreat which when this happens, the control unit receives *vehiclePassSignal* and generates */closeBarrier* follows by *displayRed*. Upon receiving *closeComplete*, the control unit may proceed to process the tag of the next car.

The Complete System

Figure 5.13 is an architectural diagram describing the communication between different components in the system. The diagram simply illustrates the message passing between the components in order to provide a clearer picture to the reader and it is not part of the UML diagram design.

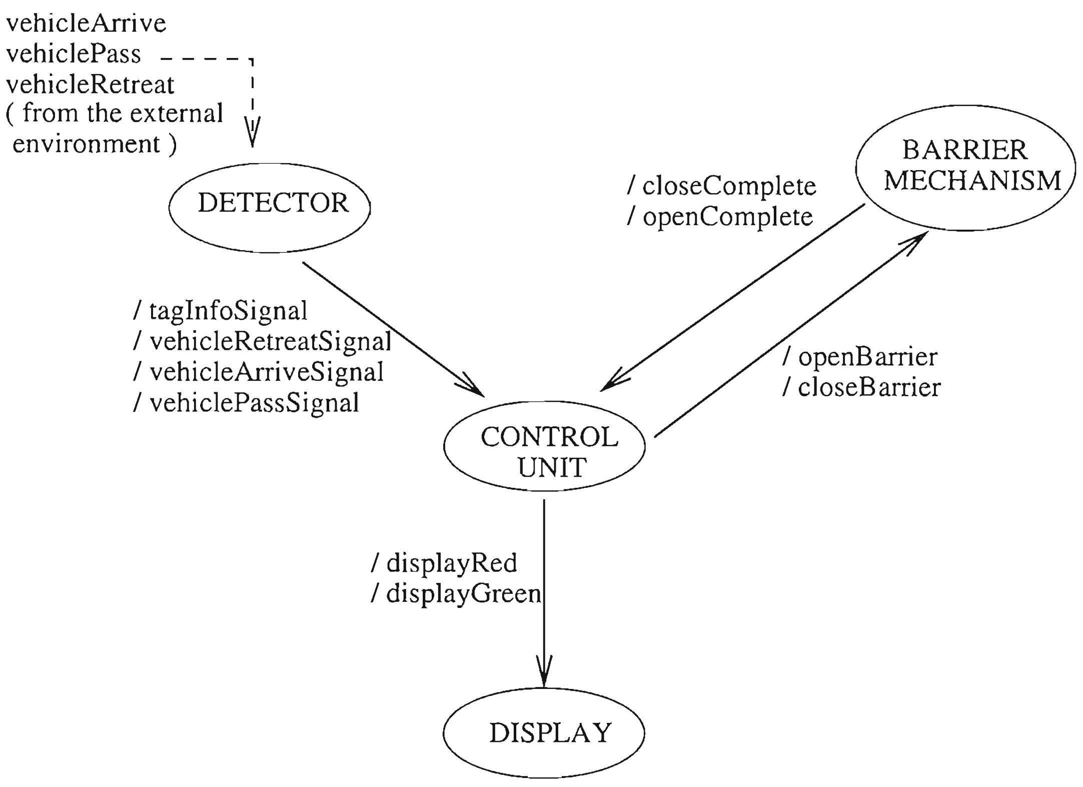


Figure 5.13: The message passing between different components.

5.2.2 Verification with FDR

For this case study, we are interested to find out if the four components which have been designed separately are able to work collectively to deliver the main function of the system. For this, we use FDR to perform a deadlock check on the design of the collective system.

The results from FDR reveals that deadlocks exist. A closer inspection at the traces performed by the detector and the control unit (show in Table 5.1) suggests that a deadlock arises when a vehicle arrives and then retreats before the information of its tag is being sent to the control unit. In this case, the detector wants to issue */vehicleRetreatSignal*, but is not able to do so without synchronizing with the receiving of the signal at the control unit side. Meanwhile, the control unit (see Figure 5.12) resides at state *IDLE* and it is only willing to accept *tagInfoSignal*. To remove the deadlock, we add a self-transition to state *IDLE* which then allows the control unit to receive *vehicleRetreatSignal* at state *IDLE*. Figure 5.14 reflect the changes.

Detector
Performs $\langle$ $vehicleArrive, vehicleArrSignal, vehicleRetreat$ $\rangle$
Accepts $\{$ $vehicleRetreatSignal$ $\}$
Control Unit
Performs $\langle\rangle$
Accepts $\{$ $tagInfoSignal$ $\}$

Table 5.1: Event traces from FDR.

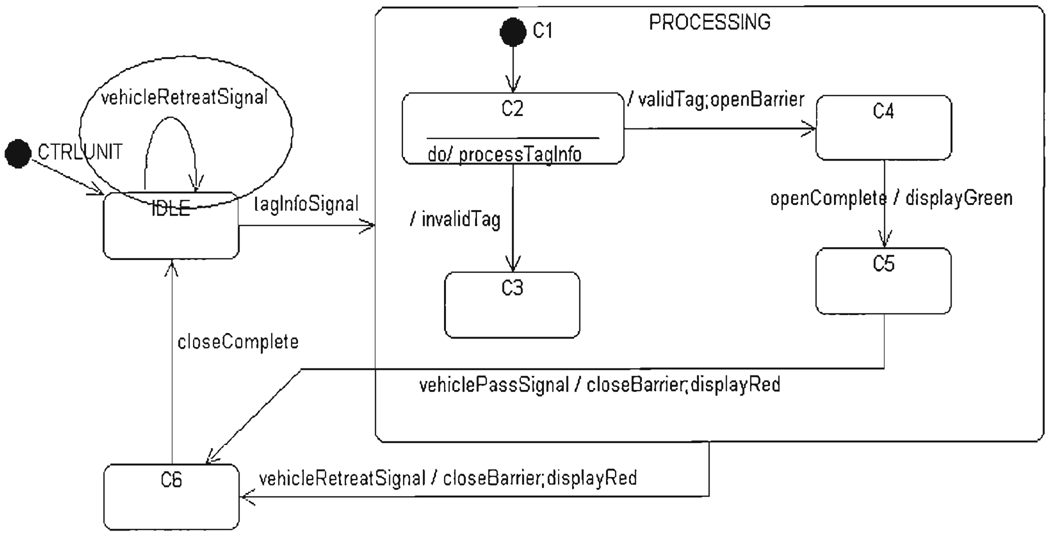


Figure 5.14: A Control Unit (version 2).

We repeat the deadlock check on the modified design and as before, deadlocks are detected. This time, FDR exposes a major flaw in the design where we actually allow the environment to perform *vehiclePass* even though the results from the tag processing shows that it is invalid. To rectify the problem, we constrain the freedom given to the environment by only allowing the environment to perform *vehiclePass* after the control unit issues *displayGreen*. Modification is made by adding to the detector a new state *B4* and a transition labelled *displayGreen* from *B3* to *B4* (see Figure 5.15). This means that the control unit will need to send *displayGreen* to the detector at the same time it issues the signal to the display.

Detector Performs $\langle$ *vehicleArrive, vehicleArrSignal, beginscanTag, endscanTag, tagInfoSignal, vehiclePass* $\rangle$. Accepts $\{$ *vehiclePassSignal* $\}$
Control Unit Performs $\langle$ *tagInfoSignal, beginprocessTagInfo, endprocessTagInfo, _tau, invalidTag* $\rangle$ Accepts $\{$ *vehicleRetreatSignal* $\}$

Table 5.2: Event traces from FDR.

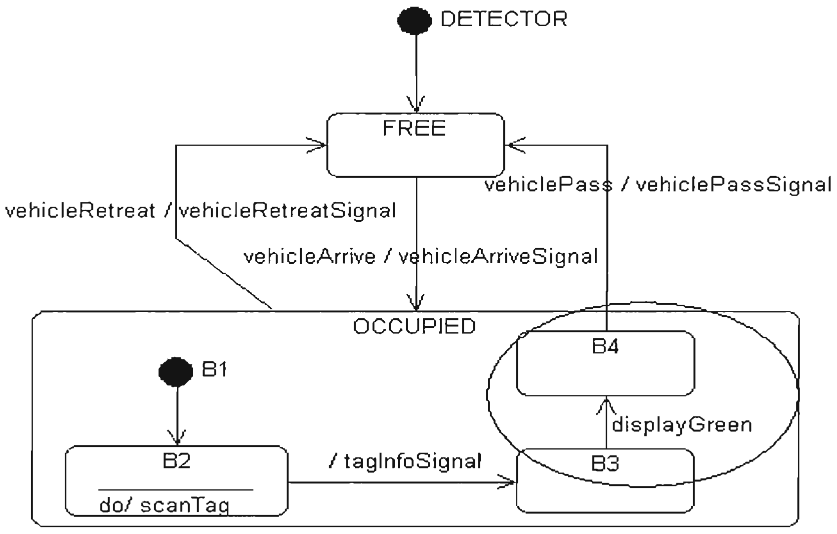

DETECTOR
FREE
vehicleRetreat / vehicleRetreatSignal
vehiclePass / vehiclePassSignal
vehicleArrive / vehicleArriveSignal
OCCUPIED
B1
B4
displayGreen
B2
do/ scanTag
/ tagInfoSignal
B3

Figure 5.15: A Detector (version 2).

When we run the design in FDR again, we encounter another deadlock. Comparing the traces listed in Table 5.3 with Figures 5.14 and 5.15, we can see that the control unit deadlocks at the transition between *C4* and *C5* because it cannot issue *displayGreen*, while the detector deadlocks at the transition between *OCCUPIED* and *FREE* and only willing to issue *vehicleRetreatSignal*. To resolve the conflict, we decided to impose a restriction on the model where a vehicle is not allowed to retreat between the state after the control unit issues *openBarrier* and before it issues *displayGreen*. What we mean by this is, looking at Figure 5.14, instead of allowing transition *vehicleRetreat* to be taken from every nested state in *PROCESSING*, we forbid the transition from *C4* by moving *C4* out of *PROCESSING* (see Figure 5.16). Similarly, we make a change to the detector by moving *B3* out of *OCCUPIED* to reflect the restriction (see Figure 5.17).

Detector Performs ⟨vehicleArrive, vehicleArriveSignal, beginscanTag, endscanTag, tagInfoSignal, vehicleRetreat ⟩. Accepts { *vehicleRetreatSignal* }
Control Unit Performs ⟨ $tagInfoSignal, beginprocessTagInfo, endprocessTagInfo, \_tau,$ $validTag, openBarrier, openComplete$ ⟩ Accepts { *displayGreen* }

Table 5.3: Event traces from FDR.

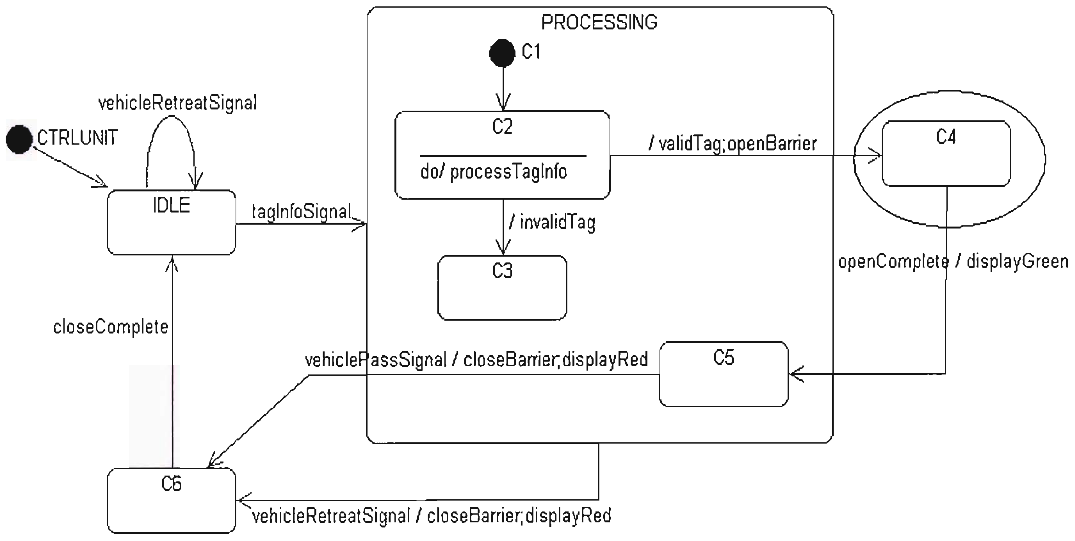


Figure 5.16: A Control Unit (version 3).

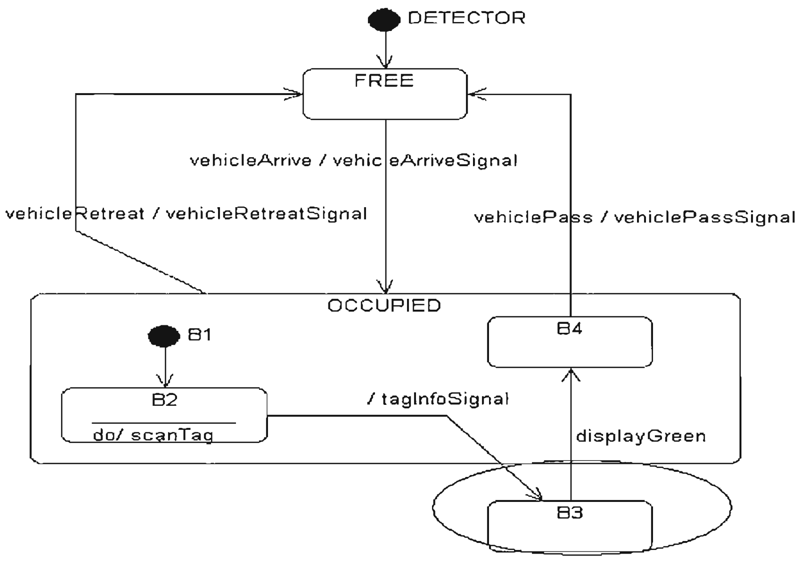
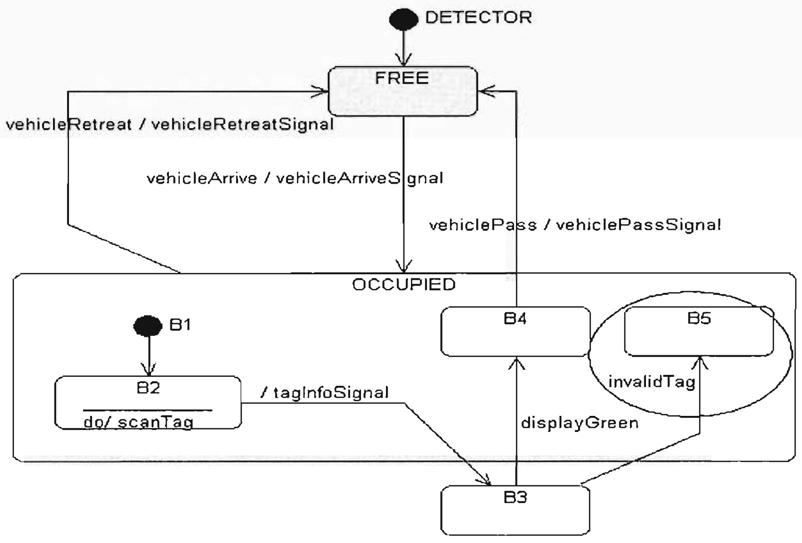

DETECTOR
FREE
vehicleArrive / vehicleArriveSignal
vehicleRetreat / vehicleRetreatSignal
vehiclePass / vehiclePassSignal
OCCUPIED
B1
B4
B2
do/ scanTag
/ tagInfoSignal
displayGreen
B3

Figure 5.17: A Detector (version 3).

However, when we run the check again, FDR detects yet another deadlock. Looking at the traces provided in Table 5.4, we found a fault in the design which reveals that vehicle retreat is not allowed when the tag is invalid. For this, we make some changes to the detector by inserting a new state *B5* with transition *invalidTag* (see Figure 5.18). (A final state can be used instead of the simple state *B5* and this will not make any difference in Figure 5.18). By placing *B5* within *OCCUPIED*, we allow a vehicle to retreat if its tag is found to be invalid. Again, this means the control unit will need to send *invalidTag* to the detector each time the signal is generated.

Detector Performs $\langle\ vehicleArrive, vehicleArriveSignal, beginscanTag, endscanTag, tagInfoSignal\ \rangle$. Accepts $\{\ displayGreen\ \}$
Control Unit Performs $\langle\ tagInfoSignal, beginprocessTagInfo, endprocessTagInfo, \_tau, invalidTag\ \rangle$. Accepts $\{\ vehicleRetreat\ \}$

Table 5.4: Event traces from FDR.



DETECTOR
FREE
vehicleRetreat / vehicleRetreatSignal
vehicleArrive / vehicleArriveSignal
vehiclePass / vehiclePassSignal
OCCUPIED
B1
B2
do/ scanTag
/ tagInfoSignal
B4
B5
invalidTag
displayGreen
B3

Figure 5.18: A Detector (version 4).

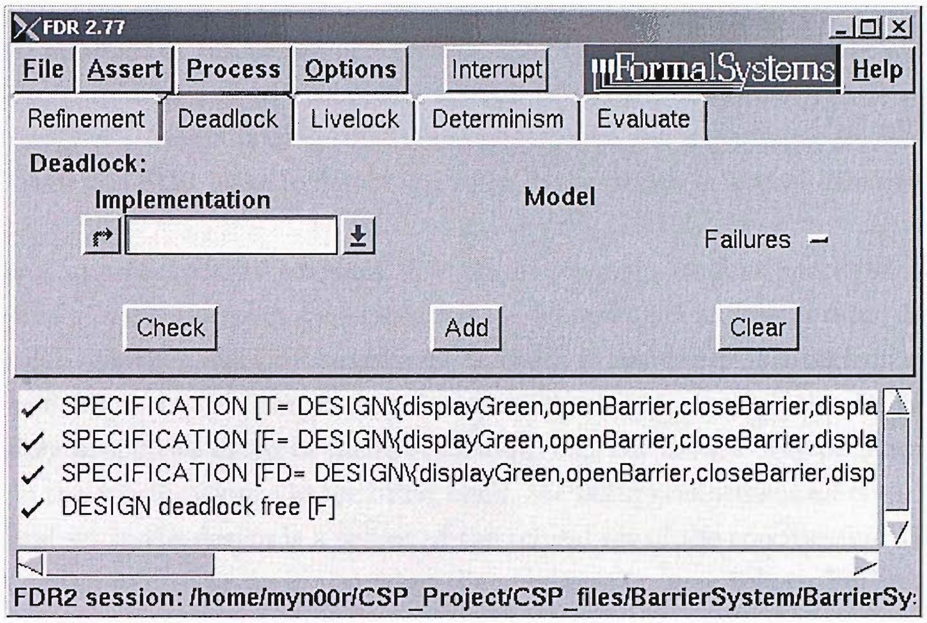
Finally, when we run the deadlock check on the overall design, FDR passes the check which proves that the design is deadlock free. Now, we may use FDR to verify if our design is working correctly with respect to the requirements we have set out earlier in Figure 5.8. For this, we perform three types of refinement check using the following three refinement assertions:

$$\begin{aligned} assert\ \ SPECIFICATION &\sqsubseteq_T\ DESIGN \setminus X \\ assert\ \ SPECIFICATION &\sqsubseteq_F\ DESIGN \setminus X \\ assert\ \ SPECIFICATION &\sqsubseteq_{FD}\ DESIGN \setminus X \end{aligned}$$

where

$$\begin{aligned} X \ = \ \{ & \ displayGreen, openBarrier, closeBarrier, displayRed, tagInfoSignal, \\ & closeComplete, openComplete, beginscanTag, endscanTag, performOpen, \\ & performClose, beginopening, endopening, beginclosing, endclosing, \\ & performDisplayGreen, performDisplayRed, beginprocessTagInfo, \\ & endprocessTagInfo, vehicleArriveSignal, vehicleRetreatSignal, vehiclePassSignal \ \} \end{aligned}$$

The results produced by FDR proves that the three assertions are true. Figure 5.19 presents a screenshot of the results. At this point we redraw the system architectural design shown earlier in Figure 5.13. All the communication channels remain the same as before except we have introduced a new communication channel from the control unit to the detector for the signals *displayGreen* and *invalidTag*, which are required to control the events of the vehicle movement in the external environment. Appendix B.2 contains the CSP code for this case study in ASCII form.



✓ SPECIFICATION [T= DESIGN\{displayGreen,openBarrier,closeBarrier,displa
✓ SPECIFICATION [F= DESIGN\{displayGreen,openBarrier,closeBarrier,displa
✓ SPECIFICATION [FD= DESIGN\{displayGreen,openBarrier,closeBarrier,disp
✓ DESIGN deadlock free [F]

FDR2 session: /home/myn00r/CSP_Project/CSP_files/BarrierSystem/BarrierSy

Figure 5.19: Verification results from FDR.

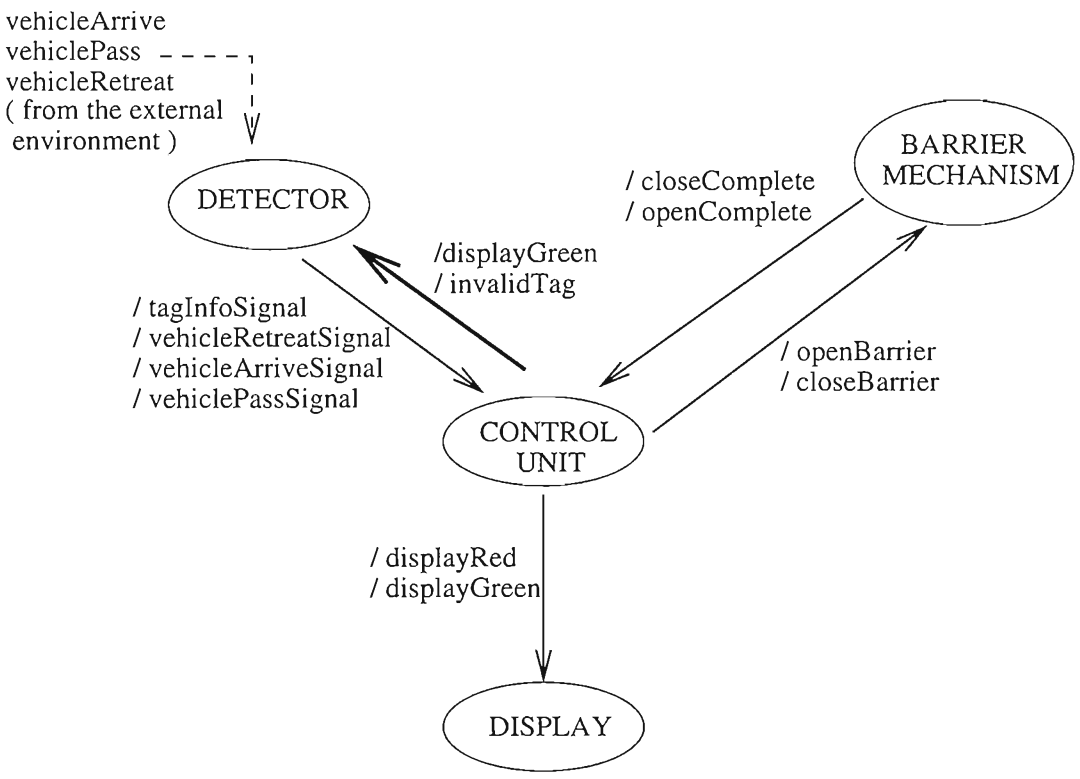


Figure 5.20: The message passing between different components (updated version).

5.3 Conclusion

We have demonstrated with the two case studies how we may incorporate visual diagrams and formal methods in the design of a system. The UML state diagrams are able to provide a clear graphical tool to visualize a system in design, while CSP/FDR provides a mean to verify if a design is of desired behaviour. From the experience, we found working with UML and FDR helps to detect any error in the design in a more effective manner.

We have approached the compact disc player example by first specifying the basic requirements of a CD player in a specification model and then adding in more details in a design model. We then use CSP refinement to check if the design is a correct refinement of the specification. In this, the trace refinement allows us to check if the design satisfies all the safety properties listed in the specification, e.g. the design only performs what is specified in the specification. On the other hand, the failure refinement allows us to check if the refusal set in the design is a subset of the refusal set of the specification model, e.g. what the design refuses to do is also what the specification may refuse. Futhermore, the divergence refinement allows us to detect if there is any livelock. We found the refinement checking provided by CSP/FDR is valuable especially in the iterative design of a system. Refinement checking may be carried out repeatedly to ensure a design still adheres to the basic requirements set out in the specification while additional features are included gradually.

When dealing with a complex design such as the barrier system, our experience with the case study shows how we may express the requirements of the system in a simple specification, and then divide and design the complex system in smaller modules. In this, we have designed each component in the system separately using an independent OR-state before combining all the components under an AND-state. With the help of CSP/FDR, we can model check each of the components to make sure they are free of deadlocks and livelocks before integrating them into the whole system. Hence, when it comes to the system level, we only need to deal with the correctness of the communication between the components without having to worry about their internal behaviour. At this level, we have used CSP/FDR to check for deadlock-free for the communication among the parallel components. Lastly, we have also shown how we may use CSP/FDR to verify that the overall design satisfies the requirements first set out in the specification.

An important observation we have gathered from our experience with the two case studies shows that we only need to work with the UML diagrams and the event traces provided by FDR without having to study the CSP code underpinning a model. This is obviously good news for those who are novices in CSP. Here, we do not wish to discount the benefits of knowing the syntax and semantics of CSP. Rather, we would like to point out that the combined approach of using UML and CSP has greatly reduced the technical overheads of introducing formal methods into the design of a system.

Chapter 6

Analysis

In this chapter, we carry out three main analyses in an attempt to answer a few questions.

First of all, the primary objective set for this thesis is to find ways to make FM more accessible by investigating the possibility of combining informal and formal methods. In this, we have proposed two approaches to combine the use of UML with CSP. For the convenience of further discussion, we refer the work on Visualizing CSP in UML in Chapter 2 as Approach A; and we name the work produced on Formalizing UML in CSP in Chapter 3-5 as Approach B. We make a comparison between the two approaches in order to find out how they differ from each other. For this, a case study is carried out using the methods proposed and we are interested to find out if one is better than the other, and how they have contributed towards our aforementioned objective. Section 6.1 reports the results gained from this analysis.

Secondly, the main issue involved in model-checking is the size of the state space involved. We have experimented with the Dining Philosopher case study to compare the performance of the CSP generated from UML with the CSP written in the usual way. (The FDR manual [45] has suggested ways to compress this case study). With the results obtained, we do not attempt to make any definite claim for we are not in position to do so with just one case study. Rather, we wish to carry out a preliminary investigation to see if there is any downside in using CSP together with UML for modelling. The results for this analysis may be found in Section 6.2.

Lastly, we make a comparison between the proposed CSP model and the OMG model. In this, we attempt to point out the similarities and differences between the two models. This work may be found in Section 6.3.

6.1 Comparison between Approach A and B

6.1.1 Table of Comparison

The table below shows a comparison between approach A and B.

	Approach A: Visualizing CSP in UML	Approach B: Formalizing UML State Diagrams in CSP
Aim	This approach intends to visualize CSP in UML. Hence, the emphasis is placed on giving a graphical representation to CSP by making compromise to the syntax and semantics rules of UML. With this, we attempt to cover as many CSP constructs as possible in the approach.	This approach aims to formalize the UML state diagrams within the framework of CSP. In this, we emphasize on providing a formal meaning to UML without violating the informal semantics suggested by the OMG group and Harel. We seek to support as many UML state diagram constructs as possible in the approach.
UML constructs involved	Class Diagram: class, association, interface class, realize relation, package. State Diagram: initial state, end state, simple state, choice state, transition event and action.	Support state diagrams only but with additional features, i.e. composite state, state action, transition action and multiple transitions.
CSP constructs involved	SKIP, STOP, simple event prefix, compound event, process parameter, event hiding, deterministic & nondeterministic choice, simple parallel, indexed parallel, indexed interleaving and refinement assertion.	SKIP, STOP, RUN, simple event prefix, compound event, process parameter, deterministic & nondeterministic choice, simple parallel, time-out and event hiding.
Sequential Behaviour	In this approach, the sequential behaviour of a system is modelled using states in sequence under the UML state diagrams. However, this approach does not support nested states and only allows for single level state.	The sequential behaviour is modelled in a similar way as for Approach A except it supports nested states.

Parallel Structure	It models the parallel processes using classes and association under the class diagram.	Parallel processes are modelled using subregions in a composite-AND-state.
Refinement Assertion	It models the refinement relationship in class diagrams using classes, packages and realize relations.	Do not support refinement assertion. The refinement assertion needs to be inserted manually.
Choice State	A choice state is used to model the CSP internal choice.	A choice state is treated as a normal state without any state action.
Tool Support	U2CSP*v1* is developed which inputs a class diagram with one/more state diagrams and generate CSP from the UML.	U2CSP*v2* is an extension from U2CSP*v1* which covers more constructs for the UML state diagrams.
Hiding	Support event hiding at the refinement level which hide events in the system that do not appear in the specification.	Event hiding is used to hide implicit events that need to be named in order to allow synchronization among subregions in a composite-AND-state.

Table 6.1: Comparing Approach A and B

6.1.2 Experiment

In this section, we proceed to compare the two approaches by experimenting using a case study. The case study models the process a student goes through in order to complete a university course. In order to pass the course, the student must complete two laboratory sessions in sequence, carry out a project and sit for a final test. If the student fails either one of these, he will not be allowed to proceed and will be considered to have failed the whole course. For the whole duration of the course, a student may choose to drop-out at any point of time. The lab session, project and test may take place concurrently.

Modelling Using Approach A

There are three processes involved in the example: lab, project and test. Under this approach, we use a state diagram to represent each of these processes, as in Figure 6.1-6.3. Since these three processes are running in parallel, they synchronize on the common events, which are *pass*, *fail* and *dropOut*. We model this using a class diagram as shown in Figure 6.4 with the common channels *pass*, *fail* and *dropOut* showing the common events which the parallel components synchronize over. The three classes represent the three state diagrams in Figure 6.1-6.3. The full CSP representation for the model can be found in Appendix B.3.

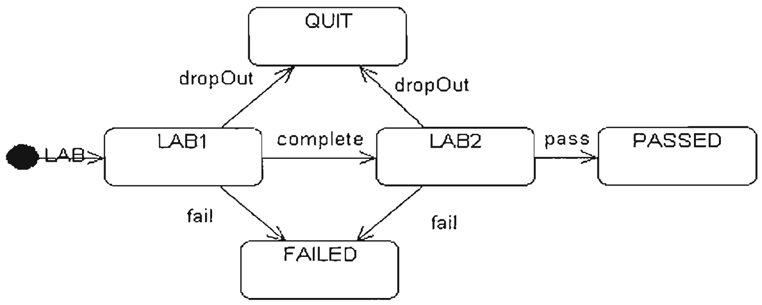


Figure 6.1: Taking two laboratory sessions.

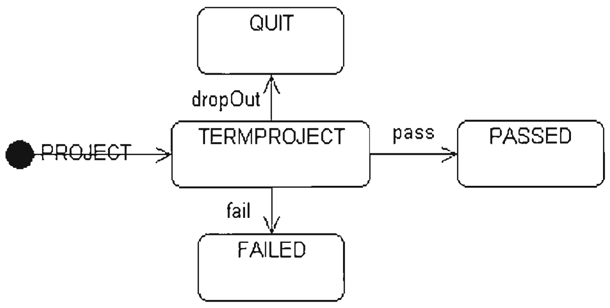


Figure 6.2: Carrying out a term project.

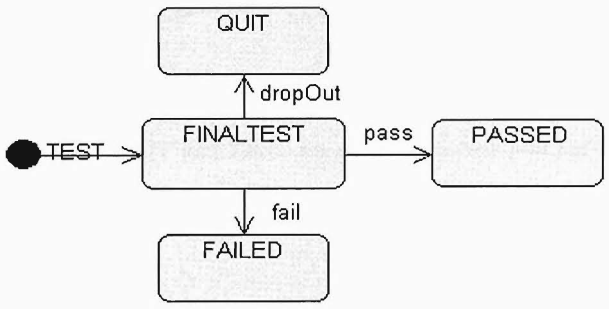


Figure 6.3: Sitting for a test.

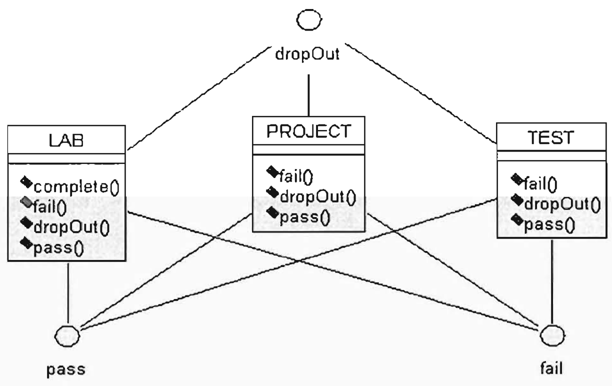


Figure 6.4: *LAB*, *PROJECT* and *TEST* are sharing channels *pass*,*fail* and *dropOut*.

Modelling Using Approach B

Under the second approach, we only need to use one state diagram to model the whole process, as in Figure 6.5. The composite state *COURSE* contains three subregions which model the three parallel components in the course: lab, project and test. Based on Figure 6.5, a student may leave the course if he/she fails in one of the components, or choose to drop out at any time during the duration of the course. These two possibilities are modelled by the transitions *dropOut* and *fail.* Taking any one of these transitions will mean an exit from all the subregions in the AND-state. A student is only considered to have passed the course if all the subregions have reached their respective end states, where the implicit transition labelled */pass* will then be activated. The full CSP representation for this model may be found in Appendix B.4.

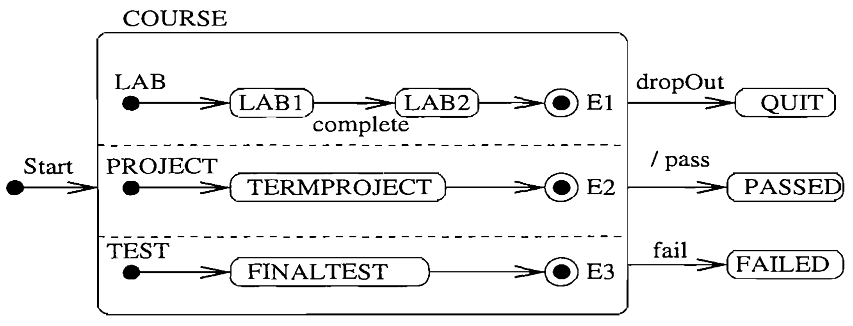


Figure 6.5: Modelling the COURSE example with Approach B.

Discussion

In this section, we will attempt to point out the differences between the two approaches. Before we do so, we want to know if the models produced by both approaches behave equally. For this, we perform a mutual refinement on the models, e.g. if $Model_{ApproachA} \sqsubseteq Model_{ApproachB}$ and $Model_{ApproachB} \sqsubseteq Model_{ApproachA}$ are true. The result from FDR confirms that they have the same behaviour.

Looking from the perspective of graphical representation:

- We can see that Approach A uses three state diagrams and one class diagram, as compared to Approach B which uses only one state diagram to illustrate the same process. From this, we may say that Approach B is more effective in providing a simple diagram to model the overall process. On the other hand, Approach A provides the advantage of showing clearly what channels are being shared among the parallel components involved whereas one needs to study Figure 6.5 under Approach B closely in order to work out the synchronized events between the parallel subregions.
- The model created using Approach A consists of two levels: the top level is made up of a class diagram showing the parallel composition; and the lower level comprises

individual state diagrams showing each of the parallel components. This hierarchical feature may prove useful when a big model is involved. Having said this, Approach B is able to provide a similar feature using the nested state contruct, i.e. the nested subregions in an AND-state.

- With Approach A, it is easier to see all the possible outgoing transitions from a state since the structure for each state diagram is a flat hierarchy. However, in this way, we may have many repetitive transitions, e.g. the transition labelled *dropOut* that appears twice in Figure 6.1. The repetitive transitions could be better replaced by one transition originating from a composite state enclosing all the states which have the same transition, as proposed by the composite state *COURSE* under Approach B.

In terms of the representation of CSP, we made the following observation:

- In terms of the code-efficiency for the model-checking in FDR, we found that the state-space explored by FDR for the model by Approach A is 3 states and 6 transitions, as opposed to 23 states and 48 transitions for the model by Approach B. This suggests that Approach A may be more amenable to model-checking using FDR, perhaps due to Approach A is geared more towards CSP, with FDR being CSP purpose built model checker. On the other hand, Approach B is more inclined towards modelling UML, where extra events and parallel contructs are introduced under the approach to model the same system. These extra events and constructs resulted in extra number of states during model-checking. This observation suggests that Approach A may generate more effective CSP for model-checking than those produced under Approach B.

All in all, we may conclude that each approach has its own strong points as opposed to the other. Therefore, it depends on the nature of the design to decide which approach will suit best.

6.2 Comparison between CSP and UML-CSP

In this section, we would like to investigate the FDR model-checking performance on the CSP codes generated from the UML models with those written by hand. For this, we carry out some experiments using a simple case study. Before we proceed further, perhaps it is useful if we discuss briefly the mechanism of model-checking utilized by FDR.

The most notable factor that influences the performance of model-checking is the size of the state-space involved. For FDR, the effectiveness of the state-space is greatly influenced by how a system is being composed. The FDR2 user manual [45] suggests some of the rule of thumbs that we may follow to achieve the optimum state-space:

- Put together two processes that are communicating with each other as early as possible when composing a system.
- Hide any unnecessary events at as low level as possible.
- Hide all events that are not relevant to the specification we try to prove.

Various compression techniques are used in FDR to reduce the state-space of the system being checked. Interested readers are referred to chapter 5 of [45]. These compression techniques are automatically employed by FDR during model-checking.

6.2.1 An Experiment

We carry out an experiment using a case study on Taking Buses (taken from FDR demo example due to Simon Gay, Royal Holloway). It is a simple system which models two bus services: 37 and 111A together with a passenger who is only willing to take bus numbered 37. The arrival of either bus 37 or 111A is nondeterministic. The model CSP is shown below:

$$\begin{aligned}
BUS37 &= board37A \rightarrow (\ (pay90 \rightarrow alight37B \rightarrow STOP) \ \Box \ (alight37A \rightarrow STOP)\ ) \\
BUS111 &= board111A \rightarrow (\ (pay70 \rightarrow alight111B \rightarrow STOP) \ \Box \ (alight111A \rightarrow STOP)\ ) \\
SERVICE &= BUS37 \sqcap BUS111 \\
PASS &= board37A \rightarrow pay90 \rightarrow alight37B \rightarrow STOP \\
SYSTEMi &= SERVICE \underset{\{board37A, pay90, alight37B\}}{\|} PASS
\end{aligned}$$

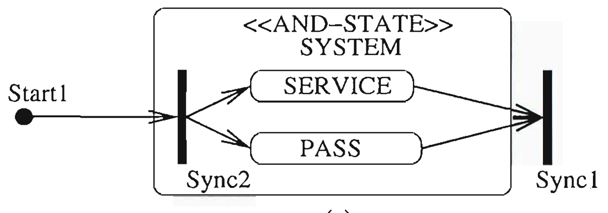
To obtain the UML/CSP, we model the system in UML as in Figure 6.6. The resulted CSP generated from the diagrams using tool U2CSP*v2* is then

$$
\begin{array}{rcl}
SERVICE & = & S1 \\
PASS & = & S2 \\
Start & = & Sync2 \\
A & = & BUS37 \sqcap BUS111 \\
BUS37 & = & board37A \rightarrow B \\
BUS111 & = & board111A \rightarrow C \\
B & = & (pay90 \rightarrow D) \,\square\, (alight37A \rightarrow E) \\
C & = & (pay70 \rightarrow G) \,\square\, (alight111A \rightarrow H) \\
D & = & alight37B \rightarrow F \\
E & = & STOP \\
F & = & STOP \\
G & = & alight111B \rightarrow J \\
H & = & STOP \\
J & = & STOP \\
S1 & = & A \\
K & = & board37A \rightarrow L \\
L & = & pay90 \rightarrow M \\
M & = & alight37B \rightarrow N \\
N & = & STOP \\
S2 & = & K \\
Sync2 & = & SYSTEM \\
SYSTEM & = & Sync2SR \\
Sync2SR & = & SERVICE \underset{\{board37A, pay90, alight37B\}}{\|} PASS
\end{array}
$$

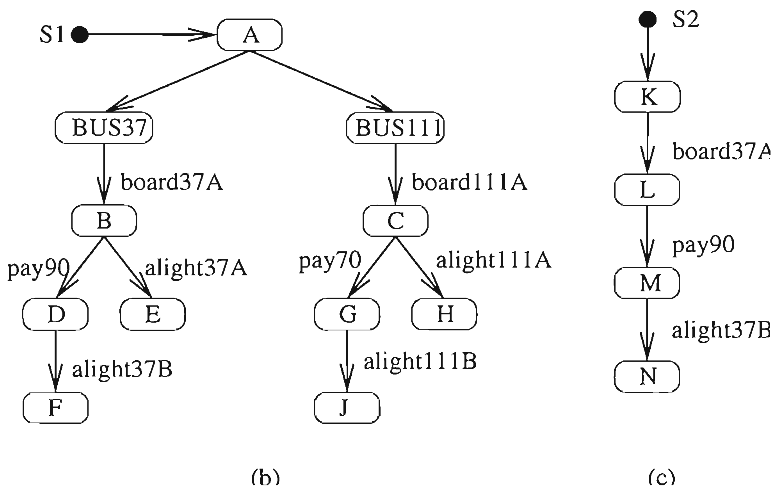
With the two CSP specifications in hand, we want to know if they are equal. For this, we check if they mutually refine each other, e.g. $SYSTEMi \sqsubseteq SYSTEM$ and $SYSTEM \sqsubseteq SYSTEMi$. The result from FDR (see Figure 6.7) proves that they behave equally the same.

Next, we model check each of them using FDR to find out about their state-space performance. We obtain the same results such that both produce one counter example after FDR refine-checked 6 states with 8 transitions. Now, suppose we make one slight change to both CSP and UML-CSP specifications by rewriting the parallel composition as

$$SERVICE \;{}_X\|_X\; PASS$$

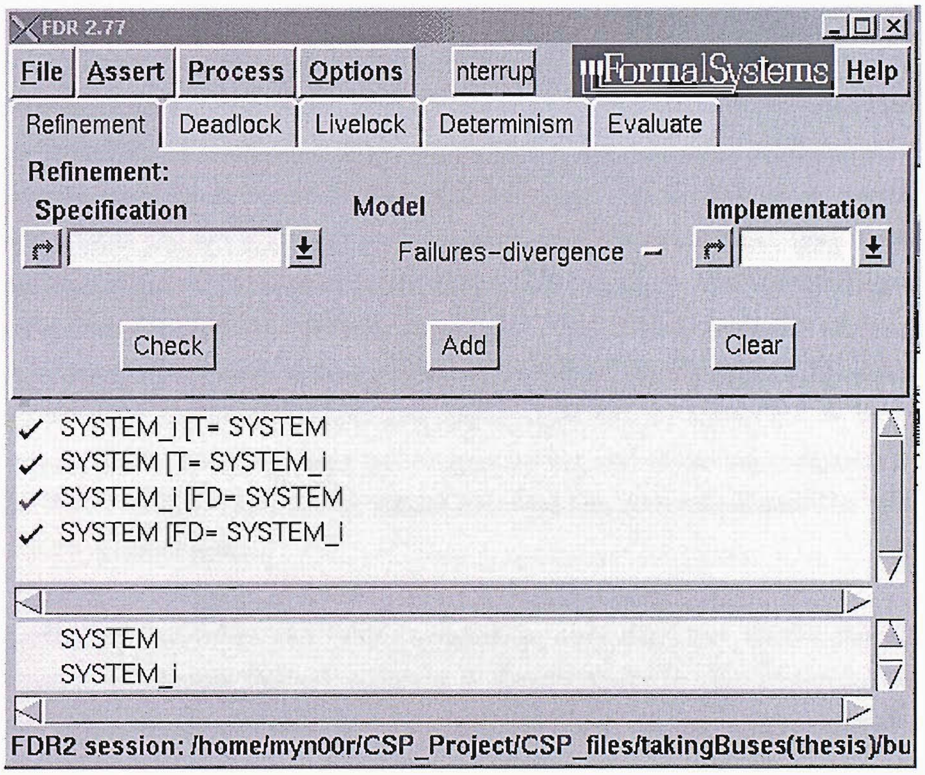


(a)



(b) (c)

Figure 6.6: Taking Buses



Refinement:

Specification Model Implementation

Failures-divergence

✓ SYSTEM_i [T= SYSTEM
✓ SYSTEM [T= SYSTEM_i
✓ SYSTEM_i [FD= SYSTEM
✓ SYSTEM [FD= SYSTEM_i

SYSTEM
SYSTEM_i

FDR2 session: /home/myn00r/CSP_Project/CSP_files/takingBuses(thesis)/bu

Figure 6.7: FDR proves that the two specifications are equal.

where $X = \{|board37A, alight37A, alight37B, board111A, alight111A, alight111B, pay70, pay90|$

Again, we obtain a similar results for the two, but this time, the model checker produces a counter example after refine-checked 3 states with 3 transitions. The difference in the performance figure is mainly because in the first instance where the parallel composition is expressed as SERVICE $\underset{\{board37a,pay90,alight37b\}}{||}$ $PASS$, the two processes only synchronize on the three events: *board37a,pay90* and *alight37b*. For this, *board111A* and *alight111A* are allowed to happen before a deadlock is encountered. This explains why more states (i.e. 6 states) are checked before a refusal is encountered. Therefore, this also demonstrates that the performance of the model-checking depends on how a system is being composed.

Here, we wish to suggest that the style of the CSP generated from the UML diagrams has similar model-checking performance as those CSP written by hand. However, even though it seems that UML-CSP has equal performance as the normal CSP when it comes to model-checking, the later fares better than the former by having a smaller number of lines of code. Having said so, UML-CSP has the tradeoff of having graphical annotation which renders better readability than reading lines of CSP code.

6.3 Comparison between Our Formalization Model and the OMG Model

The similarity found between our model defined in CSP framework and that of the OMG's on the semantics of the UML State diagrams is that both models make the same assumption by assumming the occurence of an event to be instantaneous and ignore time. The main difference prevails in terms of execution of event. OMG defines an event queue to collect events from the environment and dispatch them later. In this, they treat the generation and execution of the same event in two different steps. Unlike OMG's, our model treats the two occurence to be executed in parallel. Our model views the environment of a process to be another process, with its behaviour defined by a similar CSP notation. The interaction between the environment and the system can be modelled as two processes evolving concurrently, synchronizing on the generating and receiving of signals from either party. Therefore, unlike OMG's, our model assumes the generation and the consumption of events to be synchronous.

Comparing our model to the OMG semantics, ours does not model the UML event queue. Instead, we assume the environment is always ready to offer the event required by the process. Also, we do not consider deferred event list. Rather, we choose to ignore an event which does not invoke any transition or action.

Chapter 7

Conclusions

7.1 Conclusions

It is widely recognised that the use of formal methods(FMs) have not received its deserved attention due to the barrier imposed by its rigorious mathematical foundation. FMs such as B, Z and CSP impose use of semantically well-defined constructs and rigorously justified methods, which are its strength but unfortunately are also its weakness. Indeed, there are not many programmers who have a good background of mathematics to be able to deal comfortably with the notation in the heart of the FMs.

With this reason in mind, we set our goal to improve the use of FMs. We believe the formal verification aspect of a software is important, but it should not be forced on the end-users. To achieve a solution, we should think of ways to keep FMs in the background. For this, we start by investigating how graphical language may be combined with FMs so that they may be used collaboratively to reap the potential offered by both. The use of graphical notation is intuitive, and designs that are expressed in graphical notation have higher readability. At the same time, the use of FMs provides formal analysis and verification to the design. By combining the two methods, we are able to achieve better quality system design by having both good readability and high correctness.

Specifically, we set out to explore the possibility of combining UML with CSP. The reason UML is chosen is due to its increasing demand and attention from the industry. On the other hand, CSP is selected, largely because it is a process algebra which has relatively simpler syntax and semantics as compared to other methods such as B or Z. Furthermore, the nature of CSP fits well with the process behaviour modelling of reactive systems which our work is aiming for.

We started by proposing a lightweight approach which uses UML to visualize the syntactic behaviour of CSP (see Chapter 2). The results are encouraging in that we are able to express the complexity of a CSP formal specification using UML class diagrams and state diagrams: a class diagram is used to visualize the refinement relationships and the parallel compositions between individual processes; and a state diagram which is embedded under a class is used to describe the sequential behaviour of an individual process. In this, we achieve a two-level graphical representation of CSP, with the top level illustrating the relationships between processes, and the lower level describing in further details the behaviour for each process. Obviously, presenting CSP through this kind of intuitive graphical representation has significantly improved the readability to the outside world. Moreover, the work also suggests that we may use UML to insulate the use of CSP, that is, we may carry out a design in UML and then, using the automated translation tool we have developed, we can generate CSP from UML without having to learn to write the code ourselves. This will definitely appeal to system designers who are generally FMs illiterate but would wish to exploit the full advantage of using FMs to verify formally the correctness of their design.

The positive results obtained from the first approach has motivated us to explore further the combined use of UML and CSP. In order to allow designers to deal confidently with UML, we need to ensure the graphical notations are supported by a well-defined semantics. To achieve this, we take a step further by proposing a second approach which uses CSP to give a formal semantics to UML. To this end, we propose a set of formalization which formalises the behaviour of different states in a state machine in terms of CSP (refer Chapter 3 & 4). This then allow us to carry out model-checking on the state diagrams using FDR. Our approach is practical since our formalization is supported by a readily available verification tool. The case studies presented in Chapter 5 demonstrate the many benefits of incorporating FDR in a system design process.

Our semantics model has enabled us to formally reason about the behaviour of the state diagram in various aspects. For example, we may carry out refinement checking between two state diagrams. To do this, typically, a CSP refinement hides events in the refining process. This idea may be mapped onto UML where we can hide events and treat them as implicit in the refining model. On the other hand, we may also carry out deadlock checking on the diagram, whereby deadlocks are commonly found on parallel components that need to synchronize on certain events, or on states with guarded transitions which the guards can never be satisfied. Also, with the help of FDR, we can detect any divergence, which commonly occurs when we have a cycle of implicit events in a state machine.

Our formalization is rather simple and straighforward due to the intuitive mapping from UML to CSP. Compared to other semantics model, ours does not cover the complete set of UML state diagram constructs. Instead, we only support a subset of them that are commonly used. Clearly, our aim is to achieve a semantics model that could easily support model-checking and formal-reasoning instead of an extensive but difficult semantics which will only make model-checking complicated. The outcomes of the second approach are twofold: first, we are able to provide a formal and standardised semantics on which designers across the organisation are able to work with, and secondly, this in turns strengthens the possibility of using UML as a platform to gain access to CSP.

To conclude, we strongly feel that FMs is still the promising way towards constructing highly reliable software. Therefore we need to encourage the use of FMs by making it more accessible to the software designers. As demonstrated by the work in this thesis, combining informal methods such as UML with FMs could be a realistic way forward to use FMs without having an in-depth knowledge of how FMs work.

7.2 Further Works

i. The main difficulty we face in developing the formalization model for UML state diagram is the non-compositionality involved in UML state diagrams (i.e. the behaviour of a subregion relies on the behaviour of other subregions). This resulted in a formalization model which is rather complicated. In view of this, we propose to work with a subset of UML which is compositional, e.g. by not allowing any cross-border transitions. A compositional state diagram will produce a simpler and compositional formalization model that allows modelling of nested AND-states in another AND-state which is not supported by our existing model.

ii. We may extend our work further by looking at including B in the combined use of UML and CSP. In line with our aim to promote a better use of FMs, we feel that there are potential benefits to be reaped in combining different FMs in the system verification. To this end, Treharne & Schneider [71, 72] have been investigating the collaborative use of B and CSP. For each B-machine, they propose a controller specified in CSP to drive the interaction between different machines and ensure the operations within a machine are called within their preconditions. We hope to look at how UML can be used to support the specification framework underpined by both B and CSP. In this, we see the potentials offered by UML class diagrams and state diagrams. Also, the work by Snook & Butler [64, 65, 66, 67] on mapping B to UML will provide great insight if we choose to pursue this direction in future.

iii. Besides class diagrams and state diagrams, UML offers many other diagram views such as use cases, sequence and collaboration diagrams which intend to support a complete design by looking at different aspects of a system. In defining the semantics for state diagrams, we have not considered its relationship with other diagrams in UML. We may extend our formalization further to look at how other diagrams may be incorporated into the combined use of UML and CSP.

Bibliography

[1] Geetha Abeysinghe and Keith Phalp. *Combining Process Modeling Methods. Information and Software Technology*, 39:107–124, 1997.

[2] J-R Abrial. *The B-Book.* Camridge University Press, 1996.

[3] Sinan Si Alhir. *UML In A Nutshell.* O'Reilly & Associates, Inc., 1998.

[4] Sinan Si Alhir. *Guide to Applying the UML.* Springer-Verlag, 2002.

[5] Jim Arlow and Ila Neustadt. *UML and The Unified Process. Practical Object-Oriented Analysis and Design.* Addison-Wesley, 2002.

[6] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. *Modelling the Dynamics of UML State Machine.* In *Proceedings of the Abstract State Machine Workshop(ASM),Monte Verita, Switzerland.* Springer Verlag, Berlin, March,2000.

[7] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. *An ASM Semantics for UML Activity Diagrams.* In *Proceedings of the Algebraic Methodology and Software Technology, 8th International Conference,AMAST 2000, Iowa City, Iowa, USA.* Springer Verlag,LNCS 1816, May 2000.

[8] Christie Bolton and Jim Davies. *Activity Graphs and Processes.* In *Integrated Formal Methods, 2^{nd} International Conference, IFM 2000 Dagstuhl Castle, Germany, Nov 2000.* Springer Verlag, Berlin, 2000.

[9] A. Bouali, S. Gnesi, and S. Larosa. *The integration project for the JACK environment.* Bulletin of the EATCS (54):207223, 1994.

[10] Philip J Brooke and Richard F Paige. *The Design of a Tool-Supported Graphical Notation for Timed CSP.* In *Proceedings 3^{rd} International Conference on Integrated Formal Methods May 15-17, 2002, Turku, Finland.* Springer Verlag, 2002.

[11] Edmund M. Clarke and Jeannette M Wing. *Formal methods: state of the art and future directions. ACM Computing Surveys*, 28(4):626–643, 1996.

[12] Rational Software Corporation. *Using Rose - Rational Rose 2000e.* Part Number 800-023321-000.

[13] Rational Software Corporation. *Rose Extensibility User's Guide - Rational Rose 2000e.* Part Number 800-023328-000.

[14] Rational Software Corporation. *Rose Extensibility User's reference - Rational Rose 2000e.* Part Number 800-023329-000.

[15] Jim Davies and Charles Crichton. *Concurrency and Refinement in the Unified Modeling Language. Electronic Notes in Theoretical Computer Science*, 70(3), 2002.

[16] Steria Technologies del Information. *AtelierB Reference Manual, version 1.8.1.*

[17] Bruce Powel Douglass. *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison Wesley, 1999.

[18] Sophie Dupuy and Lydie du Bousquet. *A Multi-formalism Approach for the Validation of UML Models. Formal Aspects of Computing*, 12(4):228–230, 2000.

[19] Gregor Engels, Reiko Heckel, and Jochen Malte Kuster. *Rule-based Specification of Behavioral Consistency based on the UML Meta-Model.* In *Proceedings* 4th *International Conference on the Unified Modeling Language: UML 2001.* Springer Verlag, 2001.

[20] R. Eshuis and R. Wieringa. *A Real-Time Execution Semantics for UML Activity Diagrams.* In *Proc. Fundamental Approaches to Software Engineering (FASE 2001), LNCS 2029.* Springer Verlag, April, 2001.

[21] R. Eshuis and R. Wieringa. *A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models. CTIT Technical Report 01-04. University of Twente*, February 2001.

[22] David Harel et al. *STATEMATE: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering*, 16(4):403–413, 1990.

[23] Clemens Fischer, Ernst-Ruediger Olderog, and Heike Wehrheim. *A CSP View on UML-RT Structure Diagrams.* In *FASE 2000, Fundamental Approaches to Software Engineering, LNCS 1783*, 2000.

[24] Martin Fowler and Kendall Scott. *UML Distilled, Second Edition. A Brief Guide to the Standard Object Modelling Language.* Addison Wesley, 2000.

[25] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Malcolm Shroff. *Exploring the Semantics of UML Type Structures with Z.* In *Proceedings of the Formal Methods for Open Object-based Distributed Systems (FMOODS'97)*, 1997.

[26] S. Gnesi, D. Latella, and M. Massink. *Model checking UML statechart diagrams using JACK.* 1999. ISBN 0-7695-0418-3.

[27] David Harel. *Statecharts: A Visual Formalism For Complex System. Science of Computer Programming*, 8:231–274, 1987.

[28] David Harel. *The STATEMATE semantics of Statecharts. ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[29] David Harel, A Pnueli, J P Schmidt, and R Sherman. *On the Formal Semantics of Statecharts.* In *Proceedings Symposium on Logic in Computer Science*, 1987.

[30] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts.* Computing McGraw-Hill. ISBN 0-07-026205-5, 1988.

[31] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[32] Gerard J. Holzmann. *The Model Checker SPIN. IEEE Transaction on Software Engineering*, 23(5):279–295, May,1997.

[33] James K. Huggins and Charles Wallace. *An Abstract State Machine Primer. Michigan Technology University Computer Science Technical Report*, CS-TR-02-04, December 4, 2002.

[34] J.B.Wordsworth. *Software Engineering with B.* Addison Wesley, 1996.

[35] SRI International's Computer Science Laboratory. *The PVS Specification and Verification System.* http://pvs.csl.sri.com/overview.html.

[36] K. Lano and H Haughton. *Specification in B: an Introduction using the B-Toolkit.* Imperial College Press, ISBN 1-86094-008-0 (paperback: 1-86094-018-8), 250 pages.

[37] D. Latella, I. Majzik, and M. Massink. *Automatic Verification Of A Behavioural Subset of UML Statechart Diagrams Using The SPIN Model-Checker. Formal Aspects of Computing. The International Journal of Formal Methods, Springer*, 11(6):637–664, 1999.

[38] D. Latella, I. Majzik, and M. Massink. *Towards A Formal Operational Semantics Of UML Statechart Diagram.* In *Proceedings of the 3rd International Conference on the Formal Methods for Open OO Distributed Systems, Boston. Kluwer Academic Publishers*, 1999.

[39] Hung Ledang and Jeanine Souquières. *Contributions for Modeling UML State-Charts in B.* In *Proceedings* 3^{rd} *International Conference on Integrated Formal Methods May 15-17, 2002, Turku, Finland.* Springer Verlag, 2002.

[40] David Lightfoot. *Formal Specification using Z (Second Edition).* Palgrave, 2001.

[41] J. Lilius and Iván Porres Paltor. *Formalizing UML state machines for model checking.* In *Proceedings of UML'99, volume 1723 of Lecture Notes in Computer Science, p430-445, Springer Verlag*, 1999.

[42] J. Lilius and Iván Porres Paltor. *vUML: A Tool For Verifying UML Models.* In *Proceedings of ASE'99,pages 255-25. In 8. IEEE Computer Society*, 1999.

[43] J. Lilius and Iván Porres Paltor. *The Semantics Of UML State Machines.* In *Technical Report 273, Turku Centre for Computer Science TUCS, Turku, Finland*, June 1999.

[44] Formal Systems (Europe) Ltd. *ProBE User Manual.* http://www.fsel.com/documentation/probe/probe-doc-html/html/index.html.

[45] Formal Systems (Europe) Ltd. *Failure-Divergence Refinement, FDR2 User Manual, version 5.* 3 May 2000.

[46] Alain J Martin. *The PROBE: An Addition to Communication Primitives. Information Processing Letters*, 20:125–130, 1985.

[47] D. Latella M. Massink. *A formal testing framework for UML Statechart Diagrams behaviours: From theory to automatic verification.* 2001.

[48] Eric Meyer and Jeanine Souquieres. *A Systematic Approach to Transform OMT diagrams to a B Specification.* In *Proceedings of the FM'99 World Congress of Formal Methods.* Springer Verlag, Berlin, 1999.

[49] Erich Mikk, Yassine Lakhnech, C. Petersohn, and Michael Siegel. *On formal semantics of Statecharts as supported by STATEMATE.* In *In 2nd BCS-FACS Northern Formal Methods Worksho. Springer-Verlagp*, July 97.

[50] Erich Mikk, Yassine Lakhnech, and Michael Siegel. *Hierarchical automata as model for statecharts.* In *Asian Computing Science Conference (ASIAN'97), volume 1345 of LNCS, Springer Verlag*, December 97.

[51] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. *Implementing Statecharts in Promela/SPIN.* In *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques IEEE Computer Society*, October 21-23 1998.

[52] Muan Yong Ng and Michael Butler. *Tool Support for Visualizing CSP in UML.* In *Proceedings* 3^{rd} *International Conference for Formal Engineering Methods, Shanghai, 2002.* Springer Verlag, 2002.

[53] Muan Yong Ng and Michael Butler. *Towards Formalizing UML State Diagrams in CSP.* In *Proceedings* 1^{st} *IEEE International Conference for Software Engineering and Formal Methods, Brisbane, 2003.* IEEE Society Computer Press, 2003.

[54] OMG. *OMG Unified Modelling Language Specification version 1.4 September 2001.* http://www.rational.com/uml/resources/documentation/, 2001.

[55] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory,SRI International, September 1998.

[56] Ivan Porres. *Modelling and Analyzing Software Behavior in UML.* In *PhD Thesis.* Department of Computer Science, Abo Akademi University, Finland, 2001.

[57] Mark Priestley. *Practical Object-Oriented Design with UML.* McGraw-Hill, 2000.

[58] Rikki Prince. *Notes on using ZTC type checker and ZAN animator.* http://www.ecs.soton.ac.uk/ rfp102/cm140/.

[59] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[60] Steve Schneider. *Concurrent and Real Time Systems, The CSP Approach.* John Wiley and Sons Ltd, 2000.

[61] Steve Schneider. *The B-Method - An Introduction.* Palgrave, 2001.

[62] Emil Sekerinski and Rafik Zurob. *Translating Statecharts to B.* In *Proceedings* 3^{rd} *International Conference on Integrated Formal Methods May 15-17, 2002, Turku, Finland.* Springer Verlag, 2002.

[63] Emil Sekerinski and Rafik Zurob. *iState: A Statechart Translator.* In *Proceedings UML 2001 - The Unified Modeling Language, Toronto, Canada, M. Gogolla and C. Kobryn, Eds., Lecture Notes in Computer Science 2185, Springer-Verlag, pp. 376 – 390.* Springer Verlag, October 2001.

[64] Colin Snook. *Combining UML and B.* In *Proceedings of Forum on Specification and Design Languages Marseille*, 2002.

[65] Colin Snook, M Butler, and I Oliver. *Towards a UML profile for UML-B.* In *Technical Report DSSE-TR-2003-3, Electronics and Computer Science, University of Southampton*, 2003.

[66] Colin Snook and Michael Butler. *Using a Graphical Design Tool for Formal Specification.* In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, 2001.

[67] Colin Snook and K Sandstrom. *Using UML-B and U2B for formal refinement of digital components.* In *Proceedings of Forum on Specification and Design Languages, Frankfurt*, 2003.

[68] Ian Sommerville. *Software Engineering 6th Edition.* Addison-Wesley, 2001.

[69] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2001. May also be found at http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html.

[70] Declarative System and Software Engineering Research Group (DSSE) University of Southampton. *Research Project in Automated Validation of Business Critical Systems with Component Based Designs (ABCD).* http://www.ecs.soton.ac.uk/ ph/abcd.htm.

[71] Helen Treharne and Steve Schneider. *How To Drive a B Machine.* In *Proceedings of the International conference of Z and B Users, LNCS 1878, Springer*, 2000.

[72] Helen Treharne and Steve Schneider. *Communicating B Machines.* In *Proceedings of the International conference of Z and B Users, LNCS 2272, Springer*, 2002.

[73] Michael von der Beeck. *Formalization of UML-Statecharts.* In *M. Gogolla and C. Kobryn, editors, Proceedings of the 4th International Conference on the Unified Modeling Language (UML'2001), pages 406-421, LNCS 2185, Toronto, Canada*, 2001.

[74] Michael von der Beeck. *A Structured Operational Semantics for UML-Statecharts. Software and Systems Modeling*, 1(2):130–141, 2002.

[75] Formal Systems Website. *http://www.fsel.com/.*

[76] IBM Rational Rose Website. *http://www.rational.com/products/rose/index.jsp.*

[77] Together/J Website. *http://www.extreme-java.de/features/20001200/.*

[78] Heike Wehrheim. *Specification of An Automatic Manufacturing System - A Case Study In Using Integrated Formal Methods.* In *FASE 2000, Fundamental Approaches to Software Engineering, LNCS 1783*, 2000.

[79] Jeannette M Wing. *A Specifier's Introduction to Formal Methods. IEEE Computer*, 23(9):8–24, 1990.

Appendix A

Examples

A.1 Lift System

```
FLOOR = {1..4}
channel liftStop: FLOOR
channel button: FLOOR
channel liftMove
channel liftOpen: FLOOR
channel liftClose: FLOOR
channel press
channel release
channel doorOpen: FLOOR
channel doorClose: FLOOR

Start1 = SPEC(i)
SPEC(i) = ( liftStop.i → LIFTSTOP(i) )
LIFTSTOP(i) = ( button?k:diff(FLOOR,{i}) → BUTTONSELECTED(i,k) )
BUTTONSELECTED(i,k) = ( liftMove → SPEC(k) )
LIFT(i) = ( LiftStop.i → STOPP(i) )
Start2 = LIFT(i)
STOPP(i) = ( liftOpen.i → BOARDING(i) )
BOARDING(i) = ( button?k:diff(FLOOR,{i}) → COMPLETE(i,k) )
COMPLETE(i,k) = ( liftClose.i → CLOSED(i,k) )
CLOSED(i,k) = (liftMove → LIFT(k) )
X = ( liftMove → ACTIVE )
Start3 = X
ACTIVE = ( liftMove → ACTIVE )[]( press → HALT )
HALT = (release → X )
DOOR(i) = ( liftStop.i → LIFTARRIVE(i) )
LIFTARRIVE(i) = (doorOpen.i → DOOR_OPEN(i) )
DOOR_OPEN(i) = (button?k:diff(FLOOR,{i}) → STILL_OPEN(i) )
STILL_OPEN(i) = ( doorClose.i → DOOR_CLOSED(i) )
DOOR_CLOSED(i)= ( liftMove → DOOR(i) )
Start4 = DOOR(i)
DOORs = ||| i:FLOOR @ DOOR(i)
System1 = LIFT(1) [ | { | liftStop,button,liftMove | } | ] DOORs
```

```
System2 = System1 [ | { | liftMove | } | ] X
System = System2
assert SPEC(1) [T= System { | press,release,liftOpen,liftClose,doorOpen,doorClose | }
```

A.2 Multiplexed Buffer

```
datatype Tags = t1 | t2 | t3
datatype Data = d1 | d2

channel left: Tags.Data
channel right: Tags.Data
channel snd_mess: Tags.Data
channel rcv_ack: Tags
channel mess: Tags.Data
channel ack: Tags
channel rcv_mess: Tags.Data
channel snd_ack: Tags

Buffer(i) = ( left.i?x → L(i,x) )
Start1 = Buffer(i)
L(i,x) = ( right.i!x → Buffer(i) )
Start2 = Tx(i)
Tx(i) = (left.i?x → E(i,x) )
E(i,x) = ( snd_mess.i!x → F(i) )
F(i) = ( rcv_ack.i → Tx(i) )
SndMess = ( snd_mess?i?x → A(i,x) )
Start3 = SndMess
A(i,x) = ( mess!i.x → SndMess )
Start4 = RcvAck(i)
RcvAck(i) = ( ack?i → D(i) )
D(i) = ( rcv_ack.i → RcvAck(i) )
Start5 = RcvMess
RcvMess = ( mess?i.x → B(i,x) )
B(i,x) = ( rcv_mess.i!x → RcvMess )
Start6 = SndAck
SndAck = ( snd_ack?i → C(i) )
C(i) = ( ack!i → SndAck )
Rx(i) = ( rcv_mess.i?x → G(i,x) )
Start7 = Rx(i)
G(i,x) = ( right.i!x → H(i) )
H(i) = ( snd_ack.i → Rx(i) )
Txs = ||| i:Tags @ Tx(i)
Rxs = ||| i:Tags @ Rx(i)
System1 = Txs [|{|snd_mess|}|] SndMess
System2 = System1 [|{|rcv_ack|}|] RcvAck(1)
System3 = System2[|{|mess|}|] RcvMess
System4 = System3 [|{|ack|}|] SndAck
System5 = System4 [|{|rcv_mess,snd_ack|}|] Rxs
System = System5
Buffers = ||| i:Tags @ Buffer(i)
assert Buffers [FD= System \ {|snd_mess,rcv_ack,mess,ack,rcv_mess,snd_ack|}
```

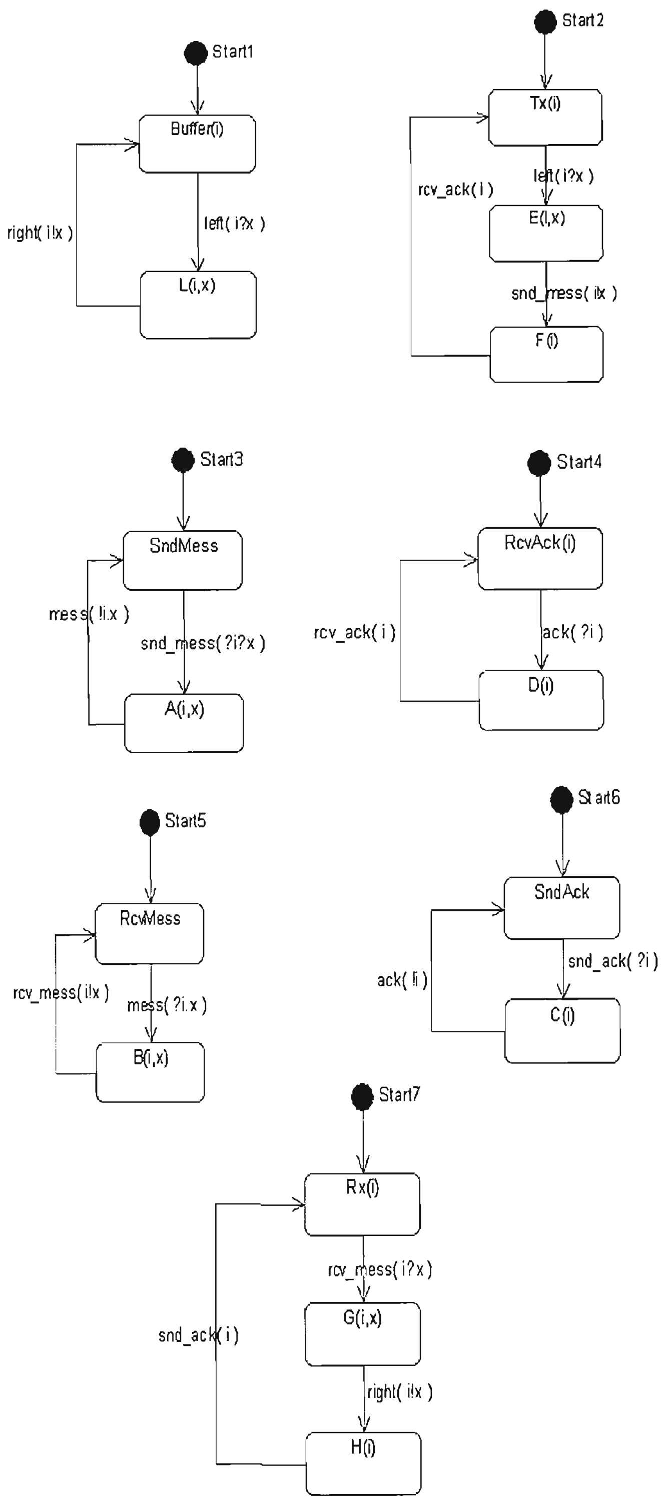


Figure A.1: State Diagrams for the Multiplexed Buffer System.

Appendix B

CSP Listing

B.1 Compact Disc Player

```
channel play
channel stop
channel pause
channel end
channel store_first_track_as_current_track
channel find_current_track
channel beginPlaying
channel endPlaying
channel store_next_track_as_current_track

SPEC = ( ( STOP) )
STOP = ( (play → PLAYING ) )
PLAYING = ( ( stop → STOP ) [] ( pause → PAUSE )) [> STOP
PAUSE = ( (pause → PLAYING )[](stop → STOP ) )
PLAYING2 = store_first_track_as_current_track → S
DESIGN = ( ( STOP2) )
STOP2 = ( (play → PLAYING2 ) )
S = ( ( PLAY_A_TRACK) )
PLAY_A_TRACK = find_current_track → ( (beginplaying → ( ( endplaying → PLAY_A_TRACK_2
) [] PLAY_A_TRACK_1 ) ) [] PLAY_A_TRACK_1 )
PLAY_A_TRACK_1 = ( ( pause → PAUSE2) [] ( stop → STOP2) )
PLAY_A_TRACK_2 = ( ( pause → PAUSE2 ) [] ( stop → STOP2 ) ) [> INCREMENT
INCREMENT = ( ( stop → STOP2 ) ) [> ( (store_next_track_as_current_track → PLAY_A_TRACK)
|~| ( end → STOP2) )
PAUSE2 = ( (pause → PLAY_A_TRACK )[](stop → STOP2 ) )

assert SPEC [T= DESIGN \ {end, store_first_track_as_current_track, find_current_track, begin-
Playing,endPlaying,store_next_track_as_current_track}

assert SPEC [F= DESIGN \ {end, store_first_track_as_current_track, find_current_track, begin-
Playing,endPlaying,store_next_track_as_current_track}
```

assert SPEC [FD= DESIGN \ {end, store_first_track_as_current_track, find_current_track, beginPlaying,endPlaying,store_next_track_as_current_track}

B.2 Barrier System

```
channel vehicleArrive
channel vehicleRetreat
channel displayGreen
channel invalidTag
channel vehiclePass
channel openBarrier
channel closeBarrier
channel displayRed
channel tagInfoSignal
channel vehicleRetreatSignal
channel closeComplete
channel openComplete
channel vehiclePassSignal
channel vehicleArriveSignal
channel beginscanTag
channel endscanTag
channel performOpen
channel performClose
channel beginopening
channel endopening
channel beginclosing
channel endclosing
channel performDisplayGreen
channel performDisplayRed
channel beginprocessTagInfo
channel endprocessTagInfo
channel validTag

SPECIFICATION = S0
BARRIER_DETECTOR = DETECTOR
BARRIER_MECH = BARRIER
BARRIER_DISPLAY = DISPLAY
CONTROL_UNIT = CTRLUNIT
OCCUPIED = B1
PROCESSING = C1
A8 = A1
A3 = A4
DESIGN = ( ( Sync2) )
DETECTOR = ( ( FREE) )
FREE = ( (vehicleArrive → vehicleArriveSignal → OCCUPIED ) )
B3 = ( (displayGreen → B4 )[](invalidTag → B5 ) )
B1 = ( ( B2) )
B2 = ( (beginscanTag → ( ( endscanTag →B2_2 ) [] B2_1 ) ) [] B2_1 )
B2_1 = ( ( vehicleRetreat → vehicleRetreatSignal → FREE) )
B2_2 = ( ( vehicleRetreat → vehicleRetreatSignal → FREE ) [> (( tagInfoSignal → B3) ) )
B4 = ( (vehiclePass → vehiclePassSignal → FREE )[](vehicleRetreat → vehicleRetreatSignal →
```

FREE))
B5 = ((vehicleRetreat → vehicleRetreatSignal → FREE))
CLOSE = ((openBarrier → performOpen → OPENING)[](closeBarrier → closeComplete → CLOSE))
OPEN = ((closeBarrier → performClose → CLOSING)[](openBarrier → openComplete → OPEN))
OPENING = ((beginopening → ((endopening →OPENING_2) [] OPENING_1)) [] OPENING_1)
OPENING_1 = ((closeBarrier → performClose → CLOSING) [] (openBarrier → OPENING))
OPENING_2 = (((closeBarrier → performClose → CLOSING) [] (openBarrier → OPENING)) [> ((openComplete → OPEN)))
CLOSING = ((beginclosing → ((endclosing →CLOSING_2) [] CLOSING_1)) [] CLOSING_1)
CLOSING_1 = ((openBarrier → performOpen → OPENING) [] (closeBarrier → CLOSING))
CLOSING_2 = (((openBarrier → performOpen → OPENING) [] (closeBarrier → CLOSING)) [> ((closeComplete → CLOSE)))
BARRIER = ((CLOSE))
DISPLAY = ((RED))
RED = ((displayGreen → performDisplayGreen → GREEN)[](displayRed → performDisplayRed → RED))
GREEN = ((displayRed → performDisplayRed → RED)[](displayGreen → performDisplayGreen → GREEN))
CTRLUNIT = ((IDLE))
IDLE = ((tagInfoSignal → PROCESSING)[](vehicleRetreatSignal → IDLE))
C6 = ((closeComplete → IDLE))
C4 = ((openComplete → displayGreen → C5))
C2 = ((beginprocessTagInfo → ((endprocessTagInfo →C2_2) [] C2_1)) [] C2_1)
C2_1 = ((vehicleRetreatSignal → closeBarrier → displayRed → C6))
C2_2 = ((vehicleRetreatSignal → closeBarrier → displayRed → C6) [> ((invalidTag → C3)| ~ | (validTag → openBarrier → C4)))
C1 = ((C2))
C3 = ((vehicleRetreatSignal → closeBarrier → displayRed → C6))
C5 = ((vehiclePassSignal → closeBarrier → displayRed → C6)[](vehicleRetreatSignal → closeBarrier → displayRed → C6))
S0 = ((A8))
A1 = ((A2))
A2 = ((vehicleArrive → A3))
A6 = ((vehiclePass → A2)[](vehicleRetreat → A2))
A7 = ((vehicleRetreat → A2))
A4 = ((A5))
A5 = ((vehicleRetreat → A2) [> ((validTag → A6) | ~ | (invalidTag → A7)))
Sync2 = BARRIER_SYSTEM
BARRIER_SYSTEM = ((Sync2SR))
Sync2_1 = BARRIER_DETECTOR [|{ | |}|] BARRIER_MECH
Sync2_2 = Sync2_1 [|{|displayGreen|}|] BARRIER_DISPLAY
Sync2SR = Sync2_2 [|{|vehicleRetreatSignal,displayGreen,invalidTag,tagInfoSignal,vehiclePassSignal, openBarrier,closeBarrier,closeComplete,openComplete,displayRed|}|] CONTROL_UNIT

assert SPECIFICATION [T= DESIGN \ {displayGreen,openBarrier,closeBarrier,displayRed, tagInfoSignal,closeComplete,openComplete,beginscanTag,endscanTag,performOpen,performClose, beginopening,endopening,beginclosing,endclosing,performDisplayGreen,performDisplayRed, beginprocessTagInfo,endprocessTagInfo, vehicleArriveSignal, vehicleRetreatSignal, vehiclePassSignal}

assert SPECIFICATION [F= DESIGN \ {displayGreen,openBarrier,closeBarrier,displayRed, tagInfoSignal,closeComplete,openComplete,beginscanTag,endscanTag,performOpen,performClose, beginopening,endopening,beginclosing,endclosing,performDisplayGreen,performDisplayRed, beginprocessTagInfo,endprocessTagInfo,vehicleArriveSignal, vehicleRetreatSignal, vehiclePassSignal}

assert SPECIFICATION [FD= DESIGN \ {displayGreen,openBarrier,closeBarrier,displayRed, tagInfoSignal,closeComplete,openComplete,beginscanTag,endscanTag,performOpen,performClose, beginopening,endopening,beginclosing,endclosing,performDisplayGreen,performDisplayRed, beginprocessTagInfo,endprocessTagInfo,vehicleArriveSignal, vehicleRetreatSignal, vehiclePassSignal}

B.3 Taking Classes (based on Approach A)

```
channel complete
channel dropOut
channel fail
channel pass

LAB = ( ( LAB1) )
LAB1 = ( (complete → LAB2 )[](dropOut → QUIT )[](fail → FAILED ) )
LAB2 = ( (pass → PASSED )[](dropOut → QUIT )[](fail → FAILED ) )
PROJECT = ( ( TERMPROJECT) )
TERMPROJECT = ( (pass → PASSED )[](dropOut → QUIT )[](fail → FAILED ) )
TEST = ( ( FINALTEST) )
FINALTEST = ( (pass → PASSED )[](dropOut → QUIT )[](fail → FAILED ) )
PASSED = STOP
QUIT = STOP
FAILED = STOP

COURSE = (LAB [|{|pass, dropOut, fail|}|] PROJECT) [|{|pass, dropOut, fail|}|]TEST
```

B.4 Taking Classes (based on Approach B)

```
channel complete
channel dropOut
channel fail
channel pass
channel tau
channel epsilon

Start = COURSE
LAB = LAB1
LAB1 = (complete → LAB2) [] ( fail → epsilon → RUN ) [] (dropOut → epsilon → RUN)
LAB2 = ( ( fail → epsilon → RUN ) [] (dropOut → epsilon → RUN ) ) [> E1
E1 = (dropOut → epsilon → RUN) [] (tau → epsilon → RUN) [] ( fail → epsilon → RUN )
PROJECT = TERMPROJECT
TERMPROJECT = ( (fail → epsilon → FAILED) [] (dropOut → epsilon → RUN) ) [> E2
E2 = (dropOut → epsilon → RUN) [] (tau → epsilon → RUN) [] (fail → epsilon → RUN)
TEST = FINALTEST
FINALTEST = ((fail → epsilon → RUN) [] (dropOut → epsilon → RUN)) [> E3
E3 = (dropOut → epsilon → RUN) [] (tau → epsilon → RUN) [](fail → epsilon → RUN)
QUIT = STOP
PASSED = STOP
FAILED = STOP
RUN = (fail → RUN)[] (dropOut → RUN) [] (tau → RUN) [] (epsilon → RUN)

COURSE = ( (LAB [|{| fail, dropOut, tau, epsilon |}|] PROJECT) [|{| fail, dropOut, tau, epsilon
|}|] TEST ) [|{| fail, dropOut, tau, epsilon |}|] ( (fail → epsilon → FAILED) [] (dropOut → epsilon
→ QUIT)[] (tau → epsilon → pass → PASSED) ) \ tau
```