Tool Support for Visualizing CSP in UML

Muan Yong Ng, Michael Butler
Department of Electronics and Computer Science,
University Of Southampton,
Southampton, SO17 1BJ,United Kingdom.
Phone: +44-(0)2380-593123
Fax: +44-(0)2380-593045
{myn00r,mjb}@ecs.soton.ac.uk

Abstract. In this paper, we propose an approach to translating UML to CSP that allows one to utilize both UML and CSP in a system design. To do so, we divide the visualization task based on three aspects of CSP: (i) the dynamics behavior, (ii) the static architecture, and (iii) the refinement assertions. In doing so we are able to visualize CSP via UML in a hierarchical manner which enhances understanding in the design process. In a way, we also provide a formal verification for the UML diagrams using CSP, since there are model checkers like FDR available. We have adopted the use of UML state diagram and class diagram for our work. In addition, we have outlined a prototyped tool that translates these diagrams to CSP that could be model-checked in FDR.

Keywords: state diagram, class diagram, CSP, UML

1 Introduction

UML (stands for *Unified Modelling Language*) [9] is a graphical modelling language comprising a collection of graphical notations(diagrams) illustrating different aspect of a software system. It has an extensively structured set of constructs but lacks a comprehensive static or behavioral semantics. CSP (Communicating Sequential Process) [1], on the other hand, is a notation for describing mainly concurrent system with components interacting with one another. It has a well-defined behavioral semantics supported by model checkers such as FDR [5]. To combine the different potential offered by the two in a design process, we propose an approach to translating UML diagrams to CSP. To do so, we use the UML state diagrams to visualize the CSP dynamic behavior such as event occurrence and state transitions. At the same time, we adopt the class diagram to show the static relationships between different CSP processes. To complete the CSP visualization in UML, we have also defined a method to model refinement relationship in the class diagram. In addition, we have developed an automated tool (built into UML Rational Rose) that will translate the UML diagrams into CSP notation, and the outcome could be analyzed by the FDR model-checker.

There are three main contributions in this paper. Firstly, we demonstrate the usefulness of incorporating graphical modelling language in the system design that uses CSP notation. Secondly, the mapping of the UML diagrams to CSP allows correctness checking on the diagrams using CSP model checkers like FDR. Thirdly, the automated translation of UML diagrams to CSP provides a graphical interface to work with CSP, and this could be useful for CSP novices.

The paper is structured as follows. We start by giving a brief introduction to the CSP notation and UML diagrams in section 2 and 3 respectively. In Section 4, we define the approach used to translate UML to CSP. We then demonstrate in section 5 how we apply the translation rules on a case study. We end with a conclusion in section 6.

2 CSP

CSP (Communicating Sequential Processes) [1] is a notation for describing concurrent system whose component, which are called **processes**, interact with each other and the environment by communication. A process can be thought of as an independent entity which has interfaces through which the process interact with the external environment. A process is defined in terms of **events**, which are the basic elements of CSP. The occurrence of an event is assumed to be instantaneous. A single event may contain more than one piece of information, in this case it is called 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.

The CSP process $a \rightarrow P$ will execute event a and then behave as process P. There are two generic CSP processes: $STOP$ and $SKIP$, which represent deadlock and successful termination respectively. When $STOP$ is executed, the system comes to a halt and it does not terminate and hence, we have a deadlock. $SKIP$, on the other hand, denotes successful termination and is identified as $\surd \rightarrow STOP$. $\surd$ is a special event in CSP that represents the act of terminating successfully. Also, given $\sum$ as a set of all possible external events, $\surd \notin \sum$.

In the following, we list out the subset of CSP that is considered in our work and discuss briefly the semantic for each one of them.

- **Parallel.** $P_X \|_Y Q$ means P is in parallel with Q over events $X \cap Y$.
- **Interleave** $P_X |||_Y Q$ means P interleaves with $Q$and there is no synchronization between their events.
- **Boolean Guard.** $B \& P$ is a Boolean guard where B is a Boolean expression such that if B is true then process P will be executed.
- **Choice.** There are two types of choices: external choice ($\Box$) and internal choice ($\sqcap$) whereby the choice is made by the environment in the former and by the process in the later.
- **Event Hiding.** In the process $Q = P \backslash S$, the execution of events in set S for process P is hidden and become internal for process Q. This is useful when a collection of processes has been combined into a system, and certain communications between the components need to be insulated and hidden from others.
- **Refinement.** Refinement is a relationship between two processes such that if the behavior of B is a subset of the behavior of A, then we say B is a refinement of A and this is expressed as $A \sqsubseteq B$ (pronounced A refined by B). The behavior of a CSP process is usually defined in terms of its trace, failure and divergence behavior [1]. Hence, we may have trace, failure or failure-divergence refinement which is stated as $A \sqsubseteq_T B$, $A \sqsubseteq_F B$ or $A \sqsubseteq_{FD} B$.
- **Indexing.** Some operators can be used in replicated versions with the use of index, 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)*, and so on.

FDR (stands for *Failure Divergence Refinement*) is a model checker for CSP. It allows concrete design description to be compared with 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 example that could be used to pin point the fault.

3 UML

3.1 State diagrams

The UML state diagram semantics and notations are substantially those of Harel's Statechart [2] except that it is an object-based variant of Harel's. A State diagram is used to describe the dynamic aspect of a system that includes the system control and timing. It specifies the **states**,which the system resides in, and the **transitions** between the states and how the system responds to various **trigger** events that cause the transition. The basic semantics of state diagrams say that a described system is always in *one* of a finite set of states. A state is a situation during the life of the model element, in which it performs some actions, satisfy some conditions, or wait for some events to occur. States take time. However, the semantics allow the modelling of states being passed either instantaneously or not instantaneously. When a trigger occurs, it will prompt the system to perform an action, which could be taking a transition, changing a variable value or generating a signal. Most often we find **events** being the triggers for the transitions. Events in state diagrams are treated as instantaneous occurrence. 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, 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*. At state *PLAYING*, the event *pause* triggers an opposite transition that brings the system back to the state *PAUSE*. For a state with more than one outgoing transition, only one transition can be fired at the point of exiting the state. 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 resolves to "true".

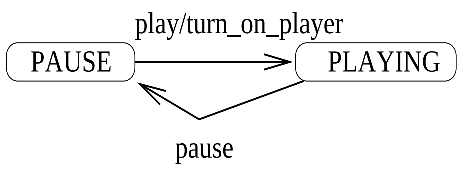

Fig. 1. A state diagram with two states.

3.2 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 2 shows the the various graphical representations of different elements in a class diagram that we make use of. The main entity in a class diagram is the **class**. Each class has its own 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 organize 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 Relationship**. It is a dotted arrow line showing a class realizing the operations offered by the other entity (pointed by the arrow head).

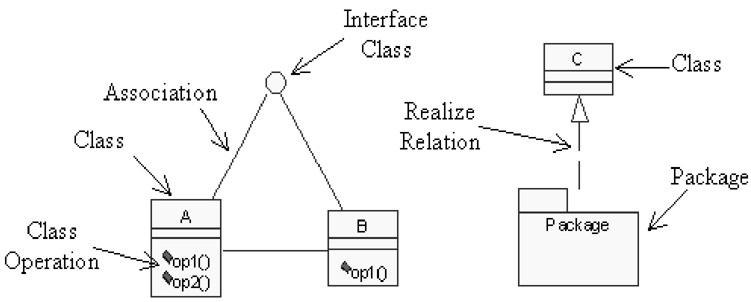

Fig. 2. Elements in a Class Diagram

4 Our Approach

We divide our task into three main aspects. We first define in Section 4.1 the method used to model the dynamic behaviour of individual CSP processes where state transitions and event occurrence are involved. In section 4.2, we propose an approach to model the CSP static structure that is concerned with the relationship between individual processes. We then illustrate how we visualize the CSP refinement assertion in section 4.3.

4.1 Visualizing the Dynamic Behavior of CSP

In the object-oriented context of UML, *start* state signifies the creation of an object and the beginning of the object lifetime, whereas *end* state signifies the termination of the object lifetime. We map each UML state to a CSP process, such that the *start* state represents the start process in CSP, the *end* state represents the termination process - SKIP. For a normal state with no outgoing transition, it denotes a deadlock and hence we equate it with STOP. These mapping are summarized as Rule(1) in Figure 3. Note that all the CSP expressions found in Figure 3 are expressed in ASCII form as supported by FDR.

Next, we move on to consider the CSP **external events**. For CSP, given $Q = a \rightarrow P$, with Q and P being the CSP processes and a as the external event, when a is offered by the environment, the system being modelled will react and pass on the control from Q to P. Similar concept can be found in state diagrams. In the state diagrams context, transitions are triggered by stimuli generated by the environment and these external stimuli could be events or changes to the data values. Once the system detects such inputs, it will react by making a transition from one state to another, i.e. exit from the source state and enter the destination state. Intuitively, we obtain another mapping rule such that any external event in CSP that causes the passing of the system control from one process to the next is represented as a trigger event in State diagrams that causes a transition from one state to another.(Rule(2) in Figure 3).

In addition to simple events, we might have other information attached to an event to get a **compound event** such as *a.x*, *a?x* or *a!x*. In UML, the notation for the event label may include a list of parameters separated by commas such that the format will be *event-name(parameter-name,...)*. We can represent the CSP event information using the UML event parameter list in which case *a.x* will be written as *a(x)*, *a?x* as *a(?x)* and *a!x* as *a(!x)*. The same mapping rule is applied to multiple-part compound event,e.g. *a.x?y* is represented as *a(x?y)* in UML.(see Rule(3) in Figure 3)

There are two types of **choice** representation in UML: a choice state (represented as a diamond shape) or a normal state with more than one outgoing transition. Every transition out of these states represents a branch for the choice and it may have an attached guard . If more than one of the guards are evaluated to true, an arbitrary branch is chosen. Conversely if none of the guards are evaluated true, the design is considered ill-formed. The two representations can be distinguished in that for the choice state, the decision on which branch to take depends on the prior actions in the same execution step[1]. 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 that occurs upon exiting from the current state. We can apply these notions to the representation of CSP choice in UML and obtain the mapping Rule(4) in Figure 3. A CSP internal choice is represented using a choice state(Rule(4a)) while a CSP external choice is represented with a normal state with more than one outgoing transition (Rule(4b)). For simplicity, we only consider boolean guard expression for CSP choice construct.

Often in CSP we wish to call a process with expressions substituted for **process parameters**. To model this in UML, we use the action in state diagrams. Referring to Rule(5) in Figure 3, the action label specifies the substitution of the parameter i with the new value k, j with the new value l and so on in the call to process A. The process A must have adequate number of process arguments to receive the substituted param-

[1] Based on [2], a system is said to carry out an execution step when it performs all the relevant reactions triggered by a stimulus or a group of stimuli. Both the triggers and the reactions occur within the same step.

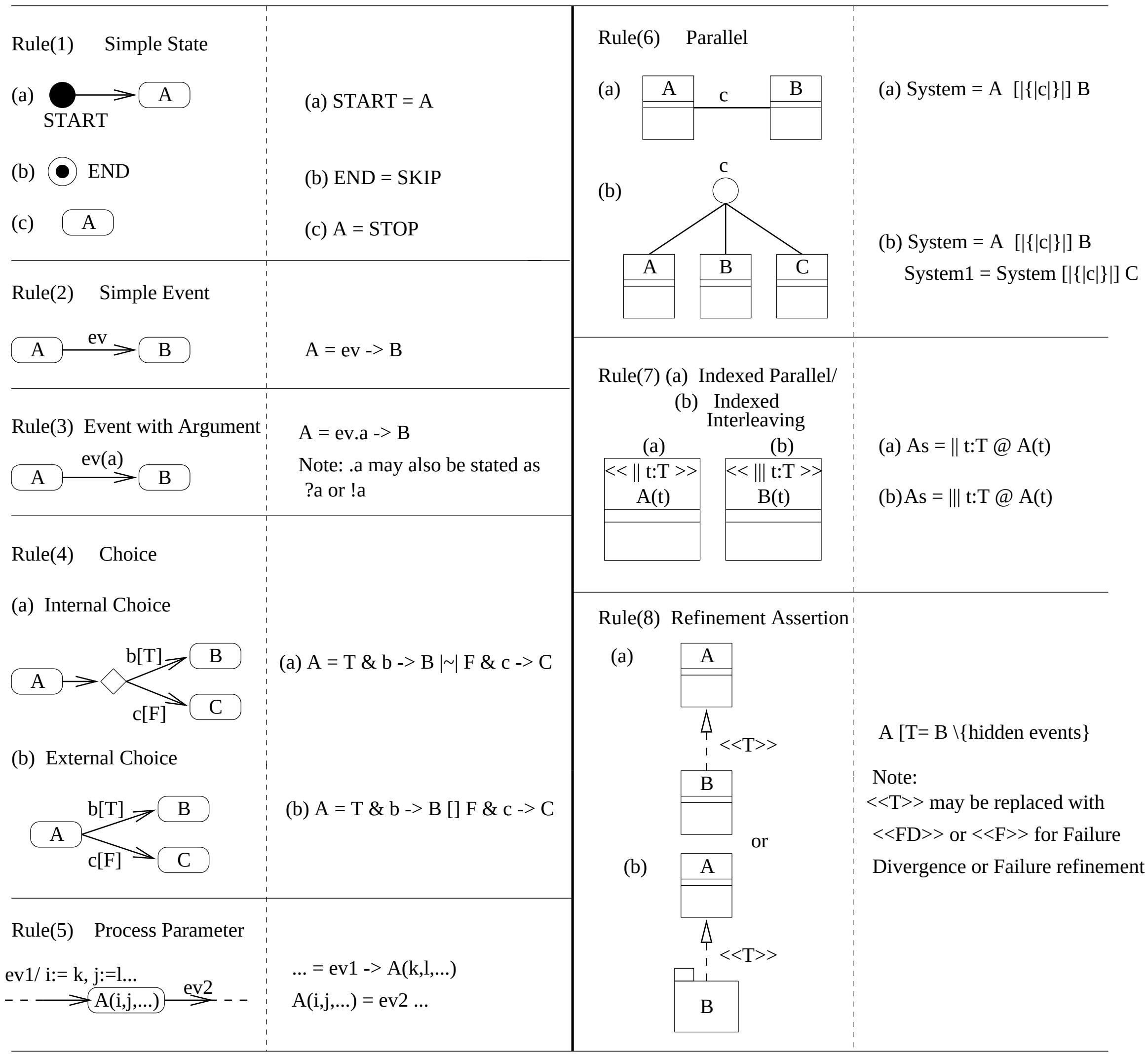

Fig. 3. Mapping Rules from UML to CSP

eters.

4.2 Visualizing the Static Structure of CSP

So far, we have illustrated how we use the UML state diagrams to model the CSP events occurrence and transitions for a process. Now, we move on to discuss how we gather these processes and visualize the static relationships between them. To achieve this, we use the class diagram.

The conventional role of UML class diagram provides a structural architecture for classes and model the static associations between them. We adopt this role for our work but treat each **class** as a CSP *process* instead. All the *events* that are involved in a process is listed under the **class operation** compartment.

The **association** that is used to connect two classes together is modelled as a *channel* that serves as an interface between two processes. The association **label** is used to name the common channel. In the case when more than two processes are sharing a common channel, we use a UML interface class which more than two processes may be connected to. (see Rule(6b) of Figure 3).

The diagram for Rule(7) in figure 3 shows the stereotyped class used to represent processes involved in indexed parallel/interleaving. Rule(9) defines the mapping rules for such representation.

4.3 Visualizing CSP Refinement Assertion

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 relationship** to connect B to A with the arrow pointing to A as in Rule(8a) figure 3. In the case where there is more than one process involves in the implementation, we group the processes (i.e. classes) into a package as shown in Rule8(b). Note the label $\langle\langle T\rangle\rangle$ beside the dotted line. It represents a *trace* refinement. Similarly, we can model the failure-divergence or failure refinement using $\langle\langle FD\rangle\rangle$ or $\langle\langle F\rangle\rangle$ respectively. The hidden events are generated automatically by the tool by comparing the events in the specification with the events in the implementation.

5 Case Study : The Lift System

In this section, we demonstrate how to apply the methods developed in section 4 on a case study.

The case study is based on a lift system that serves four floors in a building. There is a door at each of the four floors. When the lift arrives at a floor, both the lift and the floor door will open. Passengers then enter the lift and press 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 can be pressed and the lift will come to a stop. The lift will remain at halt until the button is released. For simplicity, we do not include the mechanism for requesting the lift.

5.1 Dynamic Behavior View

First of all, we define the dynamic behavior of the system using state diagrams. We identify three main entities in the system: the lift, the floor door (where there are four of them) and the emergency button. For each of these entities, we develop a state diagram to model their dynamic behavior. Figure 4 shows the state diagrams for the entities.

We begin by looking at Figure 4(a) for the lift entity. We define five states : *LIFT, STOPP, BOARDING, COMPLETE* and *CLOSED* which the lift goes through. The whole process begins at state *LIFT*. Using Rule(1a) (in figure 3), we get

$$Start2 = LIFT(i)$$

When the event *liftStop.i* occurs, that is when the lift stops at the i^{th} floor, it triggers the system to transit to state *STOPP(i)*. Notice that here we use a process parameter at the state name to receive the argument from the event. Based on Rule(3), we obtain

$$LIFT(i) = liftStop.i \rightarrow STOPP(i).$$

Applying the same rule to states *STOPP(i)*, *BOARDING(i)* and *COMPLETE(i,k)* we get

$$STOPP(i) = liftOpen.i \rightarrow BOARDING(i).$$

$$BOARDING(i) = liftClose.i \rightarrow COMPLETE(i,k).$$

$$COMPLETE(i,k) = liftClose.i \rightarrow CLOSED(i,k).$$

While the system is in state *BOARDING(i)*, it waits for the passenger to press any button, *k*, except the current floor i such that $k{:}diff(FLOOR,\{i\})$ where $FLOOR = \{1..4\}$.

At state *CLOSED(i,k)*, when event *liftMove* takes place, it not only triggers a transition from *CLOSED(i,k)* to *LIFT(i)*, but at the same time it also triggers an action $i := k$ that will substitute i with *k*. In this case, using Rule(5), we have

$$CLOSED(i,k) = liftMove \rightarrow LIFT(k)$$

whereby we now call process *LIFT()* with its parameter substituted with *k*.

We do the same for the floor door (see figure 4(b)). Bear in mind that there is more than one floor door involved in the system, and hence i in *DOOR(i)* refers to a specific door in the system. Due to space problems we will not list out the CSP for the floor door. As for the emergency button in figure 4(c), we name the states as *X, ACTIVE* and *HALT*. The process begins at state *X*. Using Rule(1a), we obtain

$$Start5 = X$$

Once the lift starts moving, the system transits to state *ACTIVE*. Rule(2) then gives

$$X = (liftMove \ \rightarrow ACTIVE)$$

The system will remain at this state as long as the lift is moving, i.e. event *liftMove* is offered by the environment continuously. At the point when the emergency button is pressed, i.e. event *press* takes place, and the system comes to a *HALT*. Hence, we can say that at state *ACTIVE* there are two events that could possibly occur, and these depend on the environment, i.e. whether the button is pressed. This is expressed in CSP using Rule(4b) as

$$ACTIVE = press \rightarrow HALT \,\square\, liftMove \rightarrow ACTIVE$$

The only way out from state *HALT* is the transition that is triggered by event *release*. Therefore, when the emergency button is released, the system goes back to state *X*, and the whole process is repeated.

$$HALT = release \rightarrow X$$

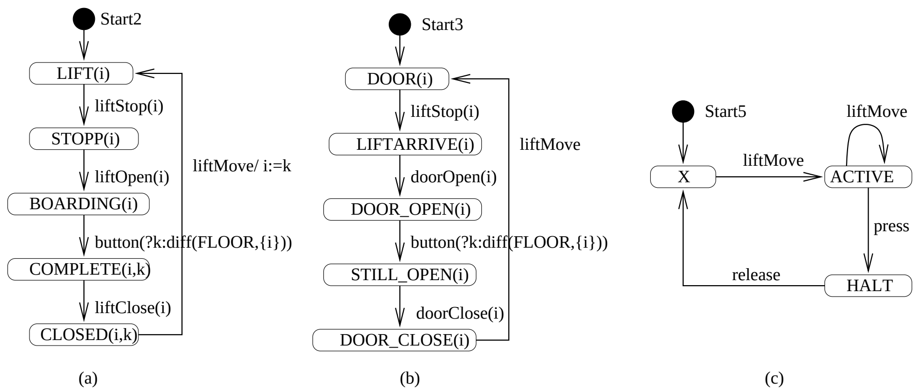

Fig. 4. State diagram for (a) lift (b) floor door and (c) emergency button

5.2 Static Structure View

In the previous section, we have defined three state diagrams that illustrate three main processes 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 concrete design. We achieve this using the method we developed in section 4.2. As shown in figure 5, we have three classes representing the three main processes: *LIFT(1), DOOR(i)* and *X* . Notice the for process *DOOR(i)*, it is stereotyped as $\langle\langle\ |||i{:}FLOOR\ \rangle\rangle$. Since $FLOOR = \{1..4\}$, it indicates that there are four copies of processes DOOR(i) interleave with one another. Translating the class into CSP using Rule(7b) we get

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

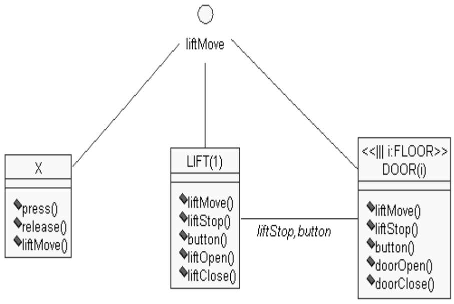

Fig. 5. The Static Behaviour of CSP Processes

Figure 5 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. Applying Rule(6b), we get

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

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

5.3 Refinement View

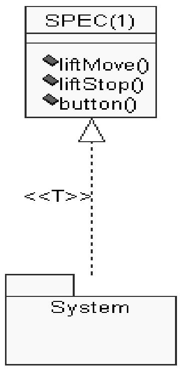

Fig. 6. The refinement assertion

All the three classes defined in section 5.2 is grouped under a package named *System* as shown in figure 6. 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) is generated (using Rule(8b) found in figure 3).

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

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 7. Using Rule(1a & 3) to map the state diagram to CSP we obtain

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

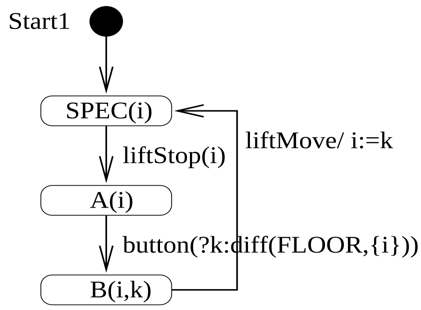

Fig. 7. Modeling the abstract behaviour of the System.

5.4 The Overview

Figure 8 put together all the diagrams we have developed so far and present an architectural structure showing how the lift system in modeled in UML.

6 Conclusion

In this paper, we have proposed a set of mapping rules that could be used to visualize CSP in UML using state diagrams and class diagrams. We have used the state diagrams to model the dynamic behaviors of a system which is made up of different CSP processes, and the class diagram to visualize the static relationships between the CSP processes and also the refinement assertion. The diagrams are then combined to give a complete representation of CSP in UML. Meanwhile, a prototype tool has also been developed based on the mapping rules that will automatically translate the UML diagrams to CSP that is accepted by FDR.

Closely related to our work is that of [6] which involves Activity Graph and CSP. The work takes a different appproach which defines a formal semantics for activity graph and then compares it with CSP, whereas we concentrate on giving a full representation of CSP in UML, and emphasizes on providing a graphical support towards CSP. [7] presented a mapping between Hoare's CSP and a subset of Role Activity

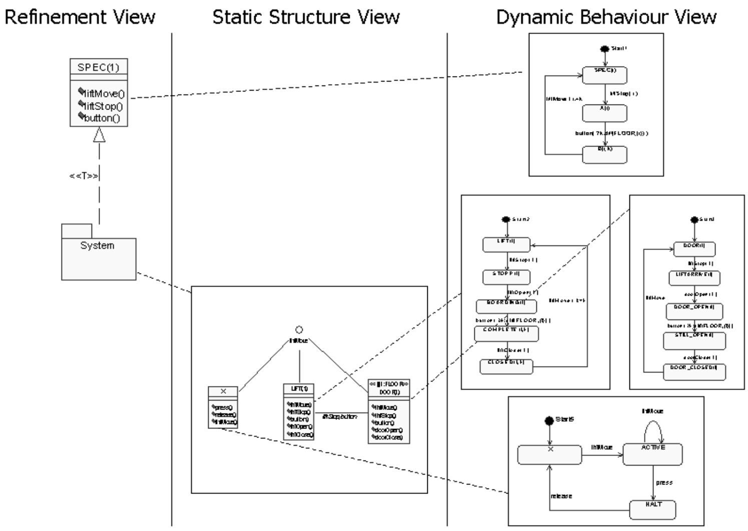

Fig. 8. An overview of the lift system in UML.

Diagrams(RADs) but it does not support the CSP refinement construct. Other related work includes [8] which emphasizes on specifying and analyzing behavioral constraints in UML and using CSP as the semantic domain for formulating the consistency constraint.

We have not covered all the CSP constructs in our visualization and we hope to pursue this in near future. We also plan to extend our work by looking at the composite states in the State diagrams.

References

1. A.W.Roscoe. The Theory and Practice of Concurrency. Prentice Hall. 1998.
2. David Harel and Michal Politi. Modeling Reactive Systems with State diagrams. Computing McGraw-Hill. 1988. ISBN 0-07-026205-5.
3. OMG Unified Modeling Language Specifictaion, Version 1.3 June 1999. http://www.rational.com/uml/resources/documentation/post.pdf
4. Mark Priestly. Practical Object_Oriented Design with UML. MacGraw Hill. 2000.
5. Formal Systems Website. http://www.formal.demon.co.uk/FDR2.html
6. Christie Bolton and Jim Davies. Activity Graphs and Processes. Integrated Formal Methods, 2^{nd} International Conference, IFM 2000 Dagstuhl Castle, Germany, Nov 2000.
7. Geetha Abeysinghe, Keith Phalp. Combining Process Modeling Methods. Information and Software Technology 39 (1997) 107-124.
8. Gregor Engels, Reiko Heckel and Jochen Malte Kuster. Rule-based Specification of Behavioral Consistency based on the UML Meta-Model. 4^{th} International Conference on the Unified Modeling Language:UML 2001. 2001.
9. Martin Fowler and Kendall Scott. UML Distilled. Applying The Standard Object Modeling Language. Addison-Wesley.1997.