Towards Formalizing UML State Diagrams in CSP

Muan Yong Ng, Michael Butler
Department of Electronics and Computer Science
University of Southampton
Southampton, United Kingdom
{myn00r,mjb}@ecs.soton.ac.uk

Abstract

The UML (Unified Modelling Language) state diagram notation [1] is a graphical language which comprises an extensive set of constructs with good structural semantics but lack of a formal behavioural semantics. With this regard, we have used the Hoare's CSP (Communicating and Sequential Processes) to formalize the behaviour of UML SD. The fact that CSP is supported by model-checkers such as FDR enables a system design using a state diagram to be formally checked during design stage. This paper presents the formalization which would allow us to reason about the behaviour of a UML SD in CSP.

1. Introduction

UML state diagrams (SDs) have well-established structural constructs but lack of any formally defined behavioural semantics. The OMG group [8] has carried out on-going work to formally define and document the standard semantics for UML SD, however, the work is restricted to the structural semantics rather than the behavioural semantics for a state machine. In this regard, much work has been carried out to give a formal semantics to the UML SD. These include those of J.Lilius et al. [6, 7] and D.Latella et.al [4, 5] which translate UML SD to Promela/SPIN that allow linear temporal logic model-checking. The works in [13, 14] formalize UML SD in B, [15] in ASM and [16] uses labeled transition systems, but these works do not support formal model-checking. In this paper, we take a different approach in which we set out to define a general but formal semantics for the behaviour of the SD in terms of CSP [2], which allows model-checking using FDR [3]. We consider a subset of UML SD, and in particular we address issues regarding interlevel transitions, implicit and explicit triggered transitions, interruption on on-going activities as well as entry and exit actions. Our previous work in [9] has paved the way for matching the SD constructs with those of the CSP. Also, we have produced a prototype tool, which is an extension to the tool in [9] to translate automatically the SDs in UML to CSP specification that is acceptable by FDR.

The rest of this paper is structured as follows: We start by giving a brief introduction to the CSP notation and UML state diagram. We then define a structural model for the UML state machine. Using this model, we will explain how the formalization is done to describe the UML State Diagram structures in CSP. Next, we will demonstrate how to apply this formalization to a case study before we conclude.

2. Background

2.1. UML State Diagram

A UML state diagram is used to describe a system behaviour in terms of its events and state changes. Its notations and semantics are substantially those of Harel's Statechart [10, 11, 12] except it is an object-based variant of Harel's. A UML SD 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 [8], an event is an observable occurrence such that it may be generated by the system 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 SDs 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 a few 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 events being the receiving of a signal or the effect of an operation call, and actions 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.(Refer Figure 8 for an example). 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.

2.2. CSP

CSP is a notation for describing concurrent system 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 and objects 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 identifier may be introduced into the system via input attached to an event, such as $a?x : X$, which describes an input value of type *X* along event *a*.

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 while synchronizing over event a. $B \& P$ 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. A deterministic choice over a set of processes is written as $\Box\, i : I \cdot P(i)$ whereby the environment is given a set of processes to choose from. 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. A nondeterministic choice over a set of processes is written as $\sqcap\, i : I \cdot P(i)$.

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 interfaces and become internal to the process. Consequently, the process will have no synchronization with other components over all the events in *S*.

Finally, 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. The behaviour of a CSP process is usually defined in terms of its trace and failure [2]. A trace is a finite sequence of events that a process performs. Given processes *A* and *B*, if $traces(B) \subseteq traces(A)$, then we say B is a trace refinement of A ($A \sqsubseteq_T B$). A failure for a process is a pair (t, R) whereby t is a trace observed in the process and R is a set of events refused by the process after the execution of the trace. R is also called a refusal. If $failures(B) \subseteq failures(A)$ then we can say B is a failure refinement of A ($A \sqsubseteq_F B$).

3. State Diagram Structure

A UML SD represents a hierarchical state machine that includes initial states, final states, choice states, simple states, composite states and transitions between states. We 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 states found in M is denoted by M_s. The set of transitions found in M is denoted by T_M and T_M is finite. E_M and $\mathcal{A}_M$ represent respectively the labels of the set of events and actions the state machine is involved in. Some actions involved in a state machine could also be events of the machines.

3.1 State Configuration

M_s is partitioned into six disjoint sets as follows: simple states M_{ss}, composite states M_{cs}, initial states M_{is}, final states M_{fs}, choice states M_{choice} and subregions M_{region}.

Initial states and final states are both pseudostates, which are transient points in a state machine in which case the former marks the beginning and the later indicates the end of a process for a state machine. Simple states are states with no other states nested within them, whereas composite states refer to states that enclose other states. Every simple state or composite state contains a label name, an entry action, an exit action and do-activity. The label name is compulsory for a state whereas the entry, exit actions and do-activity are optional. For state $k \in M_s$, it is represented by the following functions:

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

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_0, with $S_0 \in M_s$. We define a function *ENCL* : $M_s \rightarrow \mathbb{P} M_s$ that defines the set of states enclosing a substate, e.g. given a composite state X with Y nested within it and a composite state Z enclosing both X and Y, *ENCL*$(Y) = \{X, Z, S_0\}$ and *ENCL*$(X) = \{Z, S_0\}$. To distinguish X from all the enclosing states for Y where X is the immediate enclosing state for Y, we define an immediate enclosing state function *IMM* such that *IMM* : $M_s \nrightarrow M_s$, e.g. *IMM*$(Y) = X$ and *IMM*$(X) = Z$.

The composite states are partitioned into two sets: M_{cos} which refers to the composite-OR-states and M_{cas} for the composite-AND-states. An OR state contains substates that are OR-ed together whereby only one of them can be active at one time, while an AND-state contains subregions that are AND-ed together, in that when the state is entered, all the subregions become active at the same time. Each composite-AND-state is divided into a few subregions with each separated from the others by a dotted line. A subregion may contain substates that may consist of any state in M_s. Figure 1 shows an example 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 M_{region} as a global set which contains all the subregions of all the composite-AND-states in a state machine *M*, e.g. $S1, S2 \in M_{region}$. We define a total function *SUPERSTATE_FOR* which specifies the AND-state in which a subregion resides, *SUPERSTATE_FOR* : $M_{region} \rightarrow M_{cas}$. For instance, $SUPERSTATE\_FOR(S1) = X$. In order to determine the subregions to which a substate belongs, we define a partial function *SUBREGION_FOR* : $M_s \nrightarrow M_{region}$. In our example, $SUBREGION\_FOR(A) = S1$, $SUBREGION\_FOR(C) = S2$ and so on.

We define a well-formedness rule for the composite-OR-states such that an initial state must present within the first level of a composite-OR-state hierarchy to point to a state

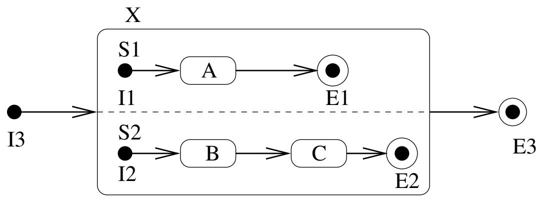

Figure 1. An Example of a composite-AND-state.

within the same level in order to indicate the start state upon entry into the composite state. Each state within the composite state must be able to be reached from the initial state syntactically in order for our formalization to work properly, but not necessary every state will reach a final state. (In cases when a state does not reach a final state, this signifies a deadlock situation in the system). Moreover, at most one initial state is allowed within the same level of a composite-OR-state but 0 or more final states could be present within the level. For the composite-AND-states, exactly one initial state must be present at each of its subregions to indicate the start state for a particular subregion.

3.2 Transition Configuration

A transition t such that $t \in T_M$ (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 target state is optional. Each transition *t* is represented by the following functions:

- the source state is represented by *source* : $T_M \rightarrow M_s$.
- the target state is represented by *target* : $T_M \rightarrow M_s$.
- the trigger event is represented by *event* : $T_M \rightarrow E_M$.
- the guard for the transition is expressed in Boolean expression. Assuming we have a language $\mathcal{B}$ that describes the guard Boolean predicate, 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 (i.e. external event offered by the environment) 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 explicit triggered transitions as T_{EM} and the set of implicit triggered transitions as T_{IM} such that $T_{EM} \cup T_{IM} = T_M$, with T_{EM} and T_{IM} partitioning T_M. Given the set of explicit events in M as E_e and the set of implicit events as E_i such that $E_M = E_e \cup E_i$ and E_e and E_i partition E_M, we have $(t \in T_{EM}) \Leftrightarrow (event(t) \in E_e)$ and $(t \in T_{IM}) \Leftrightarrow (event(t) \in E_i)$.

Similarly, for a state K such that $K \in M_s \backslash (M_{fs} \cup M_{region})$, the set of explicit triggered outgoing transitions emanating from K is T_{EK} and the set of implicit triggered outgoing transitions emanating from K is T_{IK} such that $T_{EK} \cup T_{IK} = T_K$, with T_{EK} and T_{IK} partitioning T_K and $T_K = \{t \mid t \in T_M \wedge source(t) = K\}$.

4. Formalizing the Behaviour in CSP

To approach the formalization of the state diagram behaviour, we define a function $\mathcal{H}$ that maps the structure and behaviour of a state machine to a CSP process, that is $\mathcal{H}$: State Machine $\rightarrow$ CSP. Each mapping rule is defined to represent the behaviour of a state machine starting at a particular state, as such we may have $\mathcal{H}(M,K)$ which defines the behaviour of the state machine starting at state K. Hence, we can deduce the behaviour of the entire state machine M to be $\mathcal{H}(M) = \mathcal{H}(M,I)$ where I is the initial state of M.

4.1 Basic Concept

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 explicit event. In addition, we assume that for each mapping rule defined in this paper, we are modelling the behaviour of a state machine named *M*.

In cases where there is more than one outgoing transition emanating from a state, the decision of which transition will be chosen depends on what trigger event is being offered, or in situation when more than one transition is triggered by the same event, the guards will decide which transition is to take place. For a multiple explicit triggered outgoing transitions originating from a state A, we formalize the behaviour starting at *A* as $\mathcal{H}(M,A) = \Box_{t \in T_A} event(t) \rightarrow \mathcal{H}(M, target(t))$, where the CSP external choice is used to show that the choice of transition is decided by the external environment. If the transitions are implicit, the CSP internal choice is used such that $\mathcal{H}(M,A) = \sqcap_{t \in T_A} \mathcal{H}(M, target(t))$, which shows that the choice is resolved by the process itself. Notice that we do not model the implicit events explicitly, instead we use CSP internal choice.

In the case when both implicit and explicit triggered outgoing transitions are present at *A*, a problem arises in determining whether the process or the environment has the right to resolve the choice. The semantics based on Harel and [8] does not specify the behaviour for this kind of scenario. Henceforth, we adopt the reasoning proposed by Roscoe [2] in CSP for such scenario: when the hidden event becomes available, if the unhidden event does not occur, the hidden one will be carried out eventually. For instance, if state *A* has an explicit and an implicit outgoing transitions denoted as *m* and *n* respectively, we may express the behaviour starting at *A* as

$$\begin{aligned} \mathcal{H}(M,A) \;&= (\; (event(m) \rightarrow \mathcal{H}(M, target(m))) \sqcap STOP\;) \\ &\quad \Box\; \mathcal{H}(M, target(n)) \end{aligned} \qquad \textbf{(1)}$$

The process $(\; (event(m) \rightarrow \mathcal{H}(M, target(m))) \sqcap STOP\;)$ may be internally resolved to STOP and Eq.1 becomes $\mathcal{H}(M,A) = STOP \Box \mathcal{H}(M, target(n))$. Based on the law of CSP external choice, this is equal to $\mathcal{H}(M, target(n))$. In other words, $\mathcal{H}(M,A)$ may offer *event(m)*, but will at least offer $\mathcal{H}(M, target(n))$. Roscoe refers to this phenomena as a "time-out" situation, where we can use the CSP time-out operator($\triangleright$) to express Eq.1 as $\mathcal{H}(M,A) = (event(m) \rightarrow \mathcal{H}(M, target(m)) \triangleright \mathcal{H}(M, target(n))$.

In UML, we may have actions attached to a transition or nested within a state such as entry action, exit action or do-activity. In this paper, we are only going to consider modelling UML actions as CSP events. (Meanwhile, we are working on a semantics where a state machine may have variables and actions may make assignments to them). For the actions that are attached to a transition, when the transition is triggered, it 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. For a simple or a composite state with entry and exit 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. The sequence of occurrence for these actions may be expressed in CSP as $\ldots entry(A) \rightarrow event(t) \rightarrow exit(A) \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \mathcal{H}(M, target(t))$, where *A* is a state with an outgoing transition *t* and transition action $a = a_1; a_2; \cdots; a_n$. In cases where we have a transition with multiple source states, we define an expression $EXIT_t$ which specifies the correct order for which the exit actions are being carried out. For the example in Figure 2, if the current state is $S0$, $EXIT_x = exit(S0) \rightarrow exit(S1) \rightarrow exit(S2) \rightarrow \cdots \rightarrow exit(Sn)$. We would like to point out that $EXIT_x$ is not a process but just a definition used to define processes.

The do-activity for a state represents the execution of an interruptable sequence of actions that occurs while the state

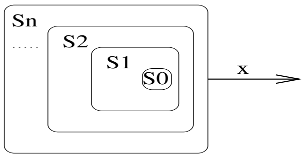

Figure 2. An example of a transition with multiple source states.

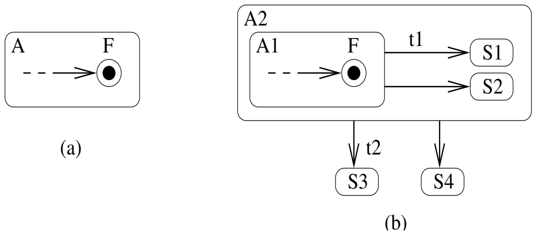

Figure 3. (a) A final state F without any outgoing transition where $\mathcal{H}(M,F) = SKIP$, (b) A final state F with outgoing transitions where $\mathcal{H}(M,F) = ((t_1 \rightarrow \mathcal{H}(M,S_1))\Box(t_2 \rightarrow \mathcal{H}(M,S_3))) \triangleright \mathcal{H}(M,S_2)$.

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 implicit triggered transition (if it is available). 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. Assuming the do activity is named *X*, we express its beginning and termination as *beginX* and *endX*. Given a state *A* with do-activity *X* and an outgoing transition *t*,

$$\mathcal{H}(M,A) = (\ beginX \rightarrow \ (\ (endX \rightarrow A1) \ \Box \ A1\ )\ ) \ \Box \ A1$$

where $A1 = event(t) \rightarrow exit(A) \rightarrow \mathcal{H}(M, target(t))$. In the above formalization, we can clearly see that *event(t)* is offered as a choice before, during and after the execution of activity *X*. This conforms to the OMG informal behavioural semantics for UML SD.

The CSP formalization we have defined for the UML constructs so far will be used as the fundamental blocks to define the behaviour of state machines that comprise initial states, end states, simple states, choice states and composite states.

4.2 Initial State

The OMG defined UML SD semantics [8] places a restriction on the use of initial state such that at most one outgoing transition is allowed from an initial state. 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. As such, for our work, we choose to constraint the use of initial state to only one within the same level of a composite state. This complies with the limitation imposed by the Rational Rose® UML tool which is used in our work.

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:

Def. 1 (Initial State) *Given A such that $A \in M_{is}$ and $t \in T_A$. If $t \in T_{IA}$, then $\mathcal{H}(M,A) = \mathcal{H}(M, target(t))$, else if $t \in T_{EA}$, then $\mathcal{H}(M,A) = event(t) \rightarrow \mathcal{H}(M, target(t))$*

4.3 Final State

Suppose we have a final state *F* nested within a set of enclosing states, *ENCL(F)*. 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 .

Def. 2 (Final State Without Outgoing Transitions) *Given $F \in M_{fs}$ and $IMM(F) \notin M_{cas}$. If $(\forall A \cdot A \in ENCL(F) \bigwedge T_A = \emptyset)$, then $\mathcal{H}(M,F) = SKIP$. (See Fig.3(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 that triggers the transition out of the enclosing state.

Def. 3 (Final State with Outgoing Transitions) *Given $F \in M_{fs}$ and $IMM(F) \notin M_{cas}$.*
Assume $(\exists A \cdot A \in ENCL(F) \bigwedge T_A \neq \emptyset)$ and if $T_{EENCL(F)} \neq \emptyset$ and $T_{IIMM(F)} \neq \emptyset$ then

$$\begin{aligned} & \mathcal{H}(M,F) \\ = \ & (\ \Box_{x \in T_{EENCL(F)}} (event(x) \rightarrow \ \mathcal{H}(M, target(x))\ )\ ) \\ & \triangleright \\ & (\sqcap_{y \in T_{IIMM(F)}} \mathcal{H}(M, target(y))) \end{aligned}$$

where $T_{EENCL(F)}$ is the set of explicit-triggered transitions that belong to any of the states that enclose *F*, and $T_{IIMM(F)}$ is the set of implicit-triggered transitions that belong to the immediate enclosing state of *F* (see Fig.3(b)). We would like to point out that: of all the implicit-triggered transition found on the states that enclose *F*, only those of the immediate enclosing state of *F* may be triggered when the final state *F* is reached. (When *F* is reached, this shows that the immediate enclosing state is terminating successfully.)

For Def. 2, 3 and later definitions, there are special cases that deal with either multiple explicit triggered transitions only or multiple implicit triggered transitions only. We have not included formalization for these special cases due to space limit.

4.4 Simple State

A simple state is a state which does not contain any substate. For a simple state *Q*, the state may be exited in a few different ways as follows: (a) State *Q* completes its activity and produces a completion event that triggers an outgoing transition through one of its implicit 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 implicit triggered transition at any of the enclosing states that encloses *Q*. (b) The external environment offers an event that triggers an outgoing transition at the border of state *Q*. The activity within the state is then abandoned and the state is exited. (c) An outgoing transition at one of *Q*'s enclosing states, say *K*, is fired and this triggers an exit from *K*, *Q* and all *Q*'s subsequent enclosing states that are nested within *K*. Based on the conditions stated above, we formalize the behaviour of the state machine starting at a simple state, and the formalization only applies to those simple states where their immediate enclosing states are not a composite-AND-state.

Def. 4 (Simple States without do-activity) *Given* $Q \in M_{ss}$. *Assume* $IMM(Q) \notin M_{cas}$ *and* $ENCL(Q) \neq \emptyset$ *and* $(\exists K \cdot K \in ENCL(s) \bigwedge T_{EK} \neq \emptyset)$. *If Q has no do-activity, and Q has both the explicit and implicit triggered transitions at its boundary such that* $T_{EQ} \neq \emptyset$ *and* $T_{IQ} \neq \emptyset$, *then*

$$\begin{aligned}
& \mathcal{H}(M,Q) \\
= \quad & entry(Q) \rightarrow (\\
& (\; (\Box_{n \in T_{EQ}} (event(n) \rightarrow EXIT_n \rightarrow \\
& \qquad \mathcal{H}(M, target(n))) \;) \\
& \; \Box(\Box_{K \in ENCL(Q)} \Box_{w \in T_{EK}} (event(w) \rightarrow EXIT_w \\
& \qquad \rightarrow \mathcal{H}(M, target(w)) \;) \;) \\
&) \\
& \triangleright \\
& (\; \sqcap_{m \in T_{IQ}} \; (EXIT_m \rightarrow \mathcal{H}(M, target(m))) \;) \\
& \qquad)
\end{aligned}$$

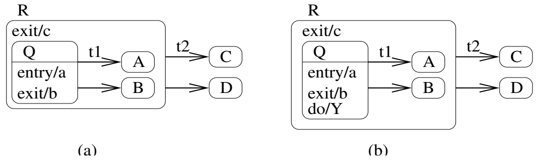

Figure 4. (a) A simple state Q without do-activity. (b) A simple state Q with do-activity.

Def. 5 (Simple States with do-activity) *Given* $Q \in M_{ss}$. *Assume* $IMM(Q) \notin M_{cas}$ *and ENCL(Q)* $\neq \emptyset$ *and* $(\exists K \cdot K \in$ *ENCL(Q)* $\bigwedge T_{EK} \neq \emptyset)$. *If Q contains a do-activity named* $\mathcal{Y}$, *and Q has both the explicit and implicit triggered transitions at its boundary such that* $T_{EQ} \neq \emptyset$ *and* $T_{IQ} \neq \emptyset$ *then*

$$\begin{aligned}
\mathcal{H}(M,Q) \quad = \quad & entry(Q) \rightarrow (\; (\; begin\mathcal{Y} \rightarrow (\; (end\mathcal{Y} \rightarrow \mathcal{S}_1 \\
& \qquad\qquad) \; \Box \; \mathcal{S}_2 \;) \\
& \qquad) \; \Box \; \mathcal{S}_2 \;)
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{S}_1 \quad = \quad & (\quad (\Box_{n \in T_{EQ}} (event(n) \rightarrow EXIT_n \rightarrow \\
& \qquad \mathcal{H}M, target(n))) \;) \\
& \quad \Box(\; \Box_{K \in ENCL(Q)} \Box_{w \in T_{EK}} \; (event(w) \\
& \qquad \rightarrow EXIT_w \rightarrow \mathcal{H}(M, target(w))) \;) \\
&) \; \triangleright \\
& \Box(\; \sqcap_{m \in T_{IQ}} \; (EXIT_m \rightarrow \mathcal{H}(M, target(m))) \;) \\
\mathcal{S}_2 \quad = \quad & (\; \Box_{n \in T_{EQ}} (event(n) \rightarrow EXIT_n \rightarrow \\
& \qquad \mathcal{H}(M, target(n))) \;) \\
& \Box(\; \Box_{K \in ENCL(Q)} \Box_{w \in T_{EK}} (event(w) \rightarrow EXIT_w \\
& \qquad \rightarrow \mathcal{H}(M, target(w))) \;)
\end{aligned}$$

Observe that transitions in the set of T_{IQ} is not available until after the do-activity completes. For the examples shown in Fig.4, 4(a) shows a simple state *Q* without do-activity such that based on Def.4

$$\begin{aligned}
\mathcal{H}(M,Q) \quad = \quad & a \rightarrow (\\
& (\quad (t_1 \rightarrow b \rightarrow \mathcal{H}(M,A)) \\
& \quad \Box(t_2 \rightarrow b \rightarrow c \rightarrow \mathcal{H}(M,C)) \\
&) \; \triangleright \\
& (b \rightarrow \mathcal{H}(M,B)) \\
& \quad)
\end{aligned}$$

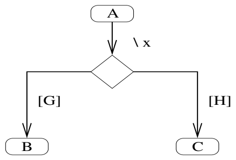

Figure 5. A choice state acting as a pseudostate between normal states.

For 4(b) where there is a simple state Q with do-activity, based on Def.5

$$
\begin{aligned}
\mathcal{H}(M,Q) &= a \rightarrow ((beginY \rightarrow ((endY \rightarrow S_1 \\
&\qquad\qquad\qquad) \,\Box\, S_2) \\
&\qquad) \,\Box\, S_2) \qquad \text{where} \\
S_1 &= (\ \ (t_1 \rightarrow b \rightarrow \mathcal{H}(M,A)) \\
&\quad \Box(t_2 \rightarrow b \rightarrow c \rightarrow \mathcal{H}(M,C)) \\
&\quad)\ \ \triangleright \\
&\quad (b \rightarrow \mathcal{H}(M,B)) \\
S_2 &= (\ (t_1 \rightarrow b \rightarrow \mathcal{H}(M,A)) \\
&\quad \Box(t_2 \rightarrow b \rightarrow c \rightarrow \mathcal{H}(M,C))\ )
\end{aligned}
$$

4.5 Choice State

Choice states are regarded as pseudostates under the OMG definition of UML State Diagrams. Since a choice state is a pseudostate, it does not contain any state action. It is different from a normal state in that it acts as an intermediate state for a compound transition between two states, as shown in Figure 5. The reason we have a partial transition between *A* and the choice state is to allow action *x* to occur which will determine the value of the guard and hence the next active state. Therefore, a choice state is regarded as a *dynamic conditional branch* as the decision of which path to take depends on the prior action ([8](part2:p152)). The formalization for a choice state is similar to that of a simple state without do-activity (refer Def. 4), except there is no entry or exit action for a choice state.

Def. 6 (Choice State) *Given* $Q \in M_{choice}$. *Assume* $IMM(Q) \notin M_{cas}$ *and* $ENCL(Q) \neq \emptyset$ *and* $(\exists K \cdot K \in ENCL(Q) \bigwedge T_{EK} \neq \emptyset)$, *then*

$$
\begin{aligned}
&\mathcal{H}(M,Q) \\
= \ (\ \ &(\ \Box_{n \in T_{EQ}}(event(n) \rightarrow \\
&\qquad \mathcal{H}(M, target(n)))\ ) \\
&\Box(\ \Box_{K \in ENCL(Q)} \Box_{w \in T_{EK}}\ (event(w) \\
&\quad \rightarrow \mathcal{H}(M, target(w)))\ ) \\
)\ \ \ &\triangleright \\
(\ &\sqcap_{m \in T_{IQ}}\ (\mathcal{H}(M, target(m)))\ )
\end{aligned}
$$

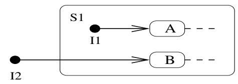

Figure 6. S1 has two possible start states: A or B.

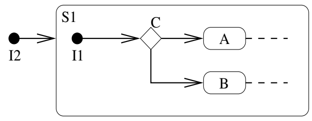

Figure 7. Modeling Fig.6 in a different way.

4.6 Composite-OR-State

A transition to a composite-OR-state represents a transition to the first state within the composite state. The first state is a state which is immediately enclosed by the composite state and it is usually indicated by a transition emanating from an initial state. Def.7 is derived based on the well-formedness rule defined earlier in section 3.1 for the composite-OR-state.

Def. 7 (Composite-OR-state) *Given* $Q \in M_{cos}$. *If there exists a single transition t where* $t \in T_M$ *and* $source(t) \in M_{is}$ *and* $source(t) = IMM^{-1}(Q)$ *and* $target(t) = IMM^{-1}(Q)$ *then*
$\mathcal{H}(M,Q) = entry(Q) \rightarrow \mathcal{H}(M, source(t))$.

Our formalization does not allow for cases where there are more than one initial state pointing to more than one start state in a composite-OR-state, as shown in Fig.6. Instead, we model Fig.6 in a different way as shown in Fig.7. Since at most one outgoing transition is allowed from an initial state, we use a pseudostate, i.e. a choice state *C* to model two implicit transitions that allow the system to proceed to either *A* or *B* .

5. Case Study

Figure 8 shows a state machine which models how a CD player works: When the player is in the NOTPLAYING state, pressing the play button causes the player to locate and play a track. This may be described in CSP using Def. 1 and 4 as

$$\mathcal{H}(M, Start1) = \mathcal{H}(M, NOTPLAYING)$$
$$\mathcal{H}(M, NOTPLAYING) = play \rightarrow \mathcal{H}(M, BUSY)$$

Note that the second expression above is an application of a special form of Def.4, that is without any implicit triggered outgoing transitions from state *NOTPLAYING*. Once the player enters state *BUSY*, it finds the most recent track and it is ready to play. We may express this (with Def. 7 and 1) as

$$\begin{aligned}\mathcal{H}(M, BUSY) &= findCurrentTrack \rightarrow \mathcal{H}(M, Start2)\\ \mathcal{H}(M, Start2) &= \mathcal{H}(M, PLAYING)\end{aligned}$$

When state *PLAYING* is entered, the system will **l**ocate the **s**tart **p**oint (**lsp**) and play the recent track which when finishes the memory will **s**tore the **s**top **p**oint (**ssp**). During the course of *PLAYING*, the system may be interrupted by pressing the *pause*, *stop* or *play* button and their subsequent state is that of *PAUSED*, *NOTPLAYING* or *BUSY* respectively. The behaviour may be described using Def. 5 as follows:

$$\begin{aligned}& \mathcal{H}(M, PLAYING)\\ =\ & lsp \rightarrow (\ (beginPlayTrack \rightarrow (\ (endPlayTrack \rightarrow S_1\\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)\Box\ S_2)\\ & \qquad\qquad)\Box\ S_2) \qquad \text{where}\\ S_1 =\ & (\quad (pause \rightarrow ssp \rightarrow \mathcal{H}(M, PAUSED))\\ & \quad \Box\ (play \rightarrow ssp \rightarrow \mathcal{H}(M, BUSY))\\ & \quad \Box\ (stop \rightarrow ssp \rightarrow \mathcal{H}(M, NOTPLAYING))\\ &) \quad \triangleright\ (ssp \rightarrow \mathcal{H}(M, End1))\\ S_2 =\ & (pause \rightarrow ssp \rightarrow \mathcal{H}(M, PAUSED))\\ & \Box\ (play \rightarrow ssp \rightarrow \mathcal{H}(M, BUSY))\\ & \Box\ (stop \rightarrow ssp \rightarrow \mathcal{H}(M, NOTPLAYING))\end{aligned}$$

When a track finishes, if button *stop* or *play* is not pressed for a certain duration of time, the system will eventually proceed to state *COMPLETE_TRACK*, in which the recent track will be incremented for the next one. Else, if the current track is the last track on the disc, the system will proceed to *NOTPLAYING*. This is represented in CSP using Def. 3 as

$$\begin{aligned}& \mathcal{H}(M, End1)\\ =\ & (\quad (stop \rightarrow \mathcal{H}(M, NOTPLAYING))\\ & \quad \Box\ (play \rightarrow \mathcal{H}(M, BUSY))\\ &) \quad \triangleright\ \mathcal{H}(M, COMPLETE\_TRACK)\end{aligned}$$

with (using Def. 4)

$$\begin{aligned}& \mathcal{H}(M, COMPLETE\_TRACK)\\ =\ & (incrementTrack \rightarrow \mathcal{H}(M, BUSY))\\ & \Box\ (lastTrack \rightarrow \mathcal{H}(M, NOTPLAYING))\end{aligned}$$

While the current track is being *pause*d, a second *pause* will resume the playing from where it last stops, while *play* will repeat the current track from the beginning, and *stop*

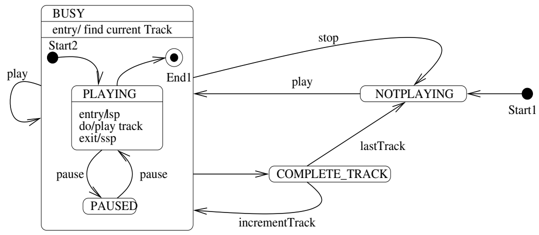

Figure 8. An Example.

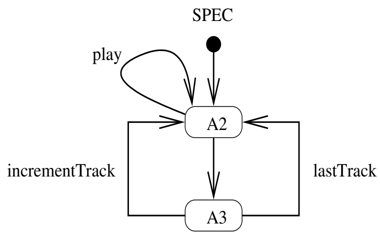

Figure 9. An Abstract Specification for the system.

will bring the player to *NOTPLAYING*. Hence, using Def. 4, we may express this behaviour as

$$\begin{aligned}& \mathcal{H}(M, PAUSED)\\ =\ & (pause \rightarrow \mathcal{H}(M, PLAYING))\\ & \Box\ (play \rightarrow \mathcal{H}(M, BUSY))\\ & \Box\ (stop \rightarrow \mathcal{H}(M, NOTPLAYING))\end{aligned}$$

So far, we have demonstrated how we could use the formalization defined in this paper to describe the behaviour of a CD player modelled using a UML SD. The CSP expression obtained from the formalization is fed into FDR for model-checking and it was found to be deadlock-free. Suppose now we introduce an error into the diagram by removing the transition with event *play* from *NOTPLAYING* to *BUSY*. When we model-check the diagram with FDR again, a deadlock is found and FDR produces a counter example which turns out to be an empty trace. This indicates that no event can be performed and there is already a deadlock at the first state (*NOTPLAYING*) encountered by the system. This behaviour is expected since we have removed the outgoing transition from *NOTPLAYING*. Deadlock is prevalent especially when designing a system involving parallel components, where a deadlock occurs if the parallel components do not succeed in synchronizing on the same event. Suppose now we want to check if the design adheres to some basic requirements of the system. We represent these basic requirements in an abstract specification (as depicted in Fig.9) and check the deadlock-free design against these requirements using the trace refinement [2] property in CSP. To do the refinement, we will need to hide events *stop, find-*

CurrentTrack, lsp, beginPlayTrack, ssp, pause and *endPlayTrack* from the abstract specification. The results from FDR suggests that the design satisfies the safety property stated in the abstract specification. (see Fig.10 for a screenshot from FDR). However, when we run a failure refinement [2]

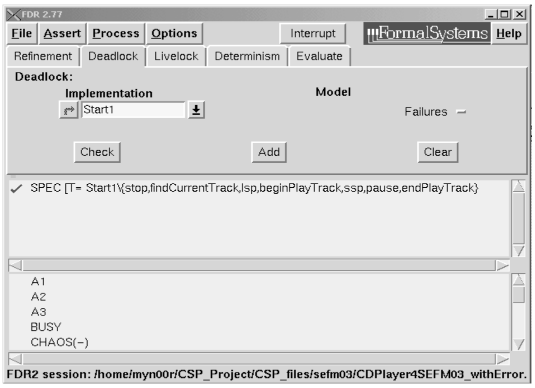

Figure 10. The trace refinement check between the specification and the design starting at Start1 is proven correct by FDR.

check on the design, the refinement assertion fails and FDR produces a counter example of a trace $\langle$*lastTrack*$\rangle$ for the specification (see Fig.11). The counter example suggests that the specification may perform *lastTrack* initially which is not permitted in the design. This is but a simple example

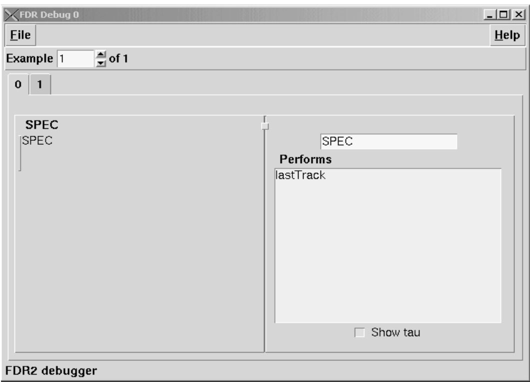

Figure 11. The failure refinement check produces a counter example in FDR.

to show how we may use the CSP refinement property to check if the design in a state diagram satisfies the required behaviour of a system.

Figure 12.

6. Discussion and Conclusions

In this paper, we have proposed a way of formalizing the behaviour of UML SDs using CSP. 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 the behaviour of the UML structures in terms of CSP processes. The basic concept we adopted in doing the translation from UML to CSP is such that we map a UML state to a CSP process and a UML event to a CSP event. The work presented in this paper has so far covered the formalization for initial state, final state, simple state, choice state and composite-OR-state. This formalization is built on the definitions we have developed for some basic structures in UML such as event, single/multiple outgoing transition, state action and transition action. Our formalization is designed to be general and hence it does not restrict to any specific domain application. Due to the limitation of space, we have not included the formalization for identifiers in UML SD which consists of event parameter and object attribute, as well as transition guard and composite-AND-state.

CSP was chosen 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 implicit transitions, the deterministic choice for the multiple explicit transitions, the time-out feature for states with both explicit and implicit 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 especially in the case where there are multiple outgoing transitions that can cause interruption.

To address the issue of an implicit event which triggers the implicit triggered transition of a state, the implicit event is not modelled using "*tau_event*" in CSP. Instead, we express, for example the state in Figure 12 as $P = a \to R$. In addition, we use the CSP nondeteministic choice to model the choice between more than one available implicit triggered transition.

The work we have produced in this paper which translates a state diagram into FDR acceptable CSP allows us to formally reason about the behaviour of the SD. 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 state with guarded transition which the guard 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.

One deficiency of using CSP in our work is that we are not able to model transition priority for a state machine. The issue of priority prevails when more than one event is being offered such that more than one transition from the same source is activated simultaneously. Since UML semantics only allow one transition from a state to take place at one time, the semantics sets the priority of transition in order to resolve the conflict. Unfortunately, CSP does not have any construct available to support the notion where more than one event may be available at one time and that each of these events may be offered on a priority basis.

Lastly, we have developed a prototype tool, which is an extension to the tool produced in [9] to generate CSP automatically from the SDs in order to model-check the diagrams using FDR.

7. Acknowledgement

We would like to thank the anonymous referees for their constructive and useful comments.

References

[1] Martin Fowler and Kendall Scott. UML Distilled Second Edition. A Brief Guide to the Standard Object Modeling Language. Addison-Wesley. 2000.

[2] A.W. Roscoe. The Theory and Practice of Concurrency. Prentice Hall. 1998. Addison-Wesley.1997.

[3] The Formal Systems Website, vendor for the FDR2. http://www.fsel.com/

[4] 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.

[5] 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.

[6] J. Lilius and Iván Porres Paltor. vUML: A Tool For Verifying UML Models. In Proceedings of ASE'99, pages 255-258. IEEE Computer Society, 1999.

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

[8] OMG(The Object Management Group,Inc). OMG Unified Modelling Language Specification version 1.4 September 2001. http://www.rational.com/uml/resources/documentation/

[9] Muan Yong Ng and Michael Butler. Tool Support for Visualizing CSP in UML. In Proceedings of the International Conference for Formal Engineering Method, Shanghai, 2002. LNCS 2495, pp. 287-298.

[10] David Harel and Michal Politi. Modeling Reactive System with Statecharts. Computing McGraw-Hill. ISBN 0-07-026205-5,1998.

[11] David Harel. Statecharts: A Visual Formalism for Complex System. Science of Computer Programming, 8:231-274,1987.

[12] 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.

[13] E. Sekerinski and R. Zurob. Translating Statecharts to B. In Proceedings of Integrated Formal Methods, IFM 2002, May 2002, Turku, Finland, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[14] Hung Ledang and Jeanine Souquires. Contributions for Modelling UML State-Charts in B. In Proceedings of Integrated Formal Methods, IFM 2002, May 2002, Turku, Finland, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[15] E Börger, A Cavarra, E Riccobene. Modeling the Dynamics of UML State Machines. Abstract State Machines. Theory and Applications. International Workshop on Abstract State Machines ASM'2000. Springer LNCS 1912, 223-241, 2000.

[16] 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, October 2001.