*Journal of Universal Computer Science, vol. 10, no. 11 (2004), 1540-1558
submitted: 10/5/04, accepted: 20/8/04, appeared: 28/11/04*

Checking Consistency between UML Class and State Models Based on CSP and B

W. L. Yeung
(Dept of Computing and Decision Sciences, Lingnan University, Hong Kong
wlyeung@ln.edu.hk)

Abstract: The B Abstract Machine Notation (AMN) and the notation of Communicating Sequential Processes (CSP) have previously been applied to formalise the UML class and state diagrams, respectively. This paper discusses their integrated use in checking the consistency between the two kinds of UML diagrams based on some recent results of research in integrated formal methods. Through a small information system example, the paper illustrates a clear-cut separation of concerns in employing the two formal methods. Of particular interest is the treatment of recursive calls within a single class of objects.
Key Words: UML, formal methods, Communicating Sequential Processes, B Abstract Machine Notation
Category: D.2.1,D.2.2

1 Introduction

The Unified Modeling Language (UML) [OMG 2001] has emerged as an industrial standard for documenting requirements, specifications, designs, and implementations of information systems. UML supports not only the fundamental object-oriented concepts including objects, classes, methods, and inheritance, but also several contemporary approaches to information systems development: use case and interaction diagrams have their origins in the scenario-based approach [Jacobson et al. 1992]; state diagrams are closely related to Harel's statecharts [Harel 1987] for reactive systems; class diagrams are based on the entity-relationship approach. While the integration of different approaches is still very much a subject of on-going research, the syntax and semantics of UML have also attracted a great deal of attention and debate. The meta-model of UML specifies rules on the composition of each kind of diagram as well as correspondences among diagrams of different kinds. Engels *et al.* [2001] refer to these as well-formedness rules. Given a set of well-formed UML diagrams about the same software, there are various ways to validate their meanings individually as well as collectively. For instance, state and interaction diagrams can be validated by animation [Koskimies et al. 1998].

An important way of validation involves checking logical consistency: given two UML diagrams of different kinds, any logical statements asserted in one diagram must not be contradicted by the other diagram, and *vice versa*. For instance, if a class diagram says a book cannot be on loan and reserved by the same person at the same time, the behaviour of a person must not be said to the contrary in a state diagram. While such kind of analysis is facilitated by the intuitiveness of UML diagrams to a certain extent, it can be much more rigorously carried out with the help of formal logic provided that we can put the meanings of these diagrams in a formal theoretical setting. This would involve formalising the semantics of UML.

Various ways of formalising different parts (subsets) of UML have been proposed. In most cases, a single formalism is employed for capturing the semantics of UML. However, different kinds of UML diagram involve disparate sets of concepts and meanings that can be more naturally and conveniently expressed in different formalisms. For example, a process algebra such as Communicating Sequential Processes (CSP) [Hoare 1985] or LOTOS [Bolognesi and Brinksma 1988], is arguably more suitable than a state-based formalism, such as the B Abstract Machine Notation (AMN) [Abrial 1996] or the Z Notation [Spivey 1992], for formalising the meaning of a UML behavioural diagram (e.g. a state diagram). On the other hand, B and Z are more convenient for capturing the meaning of a UML class diagram.

Integrated formal methods are currently an active research area. In particular, there has been much interest in integrating state-based and event-based (behavioural) formal methods [Bowman and Derrick 1999]. This paper discusses the application of some recent results of research in integrated formal methods by Treharne and Schneider [1999, 2002] for checking the logical consistency between class and state diagrams. While CSP and B have separately been applied for similar purposes, their integrated application to UML discussed in this paper is novel and constitutes the main contribution of this paper. Furthermore, previous attempts (e.g. [Meyer and Souquières 1999; Ledang and Souquières 2001]) to capture the UML behavioural semantics in B did not handle recursive calls within a class. Tenzer and Stevens [2003] proposed the modelling of objects that receive recursive calls as recursive state machines but did not address the consistency between the class and state models. The integrated use of CSP and B here allows us to tackle recursive calls within a class in a natural manner.

The next section defines the consistency problem addressed in this paper with a motivating example. Section 3 briefly describes the application of the B AMN to formalising the class model. Section 4 illustrates the use of a single CSP control-loop process to describe the state-machine behaviour of a system of interacting objects. Section 5 explains the role of abstract machine operations in relating state-machine behaviour to class structure. Section 6 outlines the application of Treharne and Schneider's formal technique for checking the consistency between state-machine behaviour and class structure. Section 7 gives the conclusion and identifies some further work.

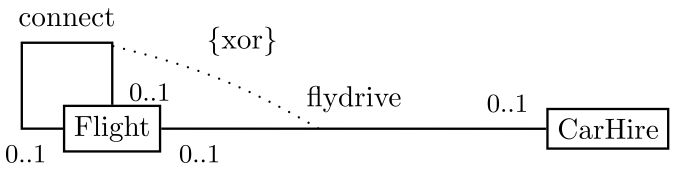

Figure 1: A class diagram

2 A Motivating Example

As a motivating example, let us consider a small information system that handles travel bookings. There are two classes of objects, namely, Flight and Car Hire. Figure 1 shows a UML class diagram for the system. A passenger's itinerary may consist of several connecting flights and the system maintains one booking for each flight in such an itinerary; hence, there is the "connect" association among flight bookings. The system also maintains car hire bookings for flight passengers hiring cars at their destination airports. Since a passenger may "fly" and then "drive" away from the destination airport in the same itinerary, there is a "flydrive" association between the two classes. Note that a flight booking can be associated with either another flight (in a connect association) or a car hire (in a flydrive association) but not both at the same time.

We can determine the functionality of a system by considering how the system is to be used, i.e. use cases of the system. Each use case may be elaborated by one or more scenarios of interaction between the actor(s) and the system's objects as well as interaction among the objects. Figure 2 shows a UML interaction diagram describing a particular scenario of the "Cancel a flight" use case. It shows the actor and objects involved in the scenario and the order in which they send call messages to each other. The actual receipt of a (operation) call message by an object is termed a "call event" and the desired sequences of call events happening to an object can be modelled by a state machine and represented by a state diagram in UML. Figure 3 shows a state diagram for individual Flight objects.

If the state machine of an object involves any actions that affect the object's associations with other objects, it needs to be checked against the class structure; if it also involves transitions that are conditional upon the machine-states of other objects of the same class, then the combined state-machine behaviour of the whole class of objects should be checked against the class structure. More generally, since objects of different classes may interact by sending call messages to each other, the combined behaviour of the classes of objects involved need to be checked against the class structure.

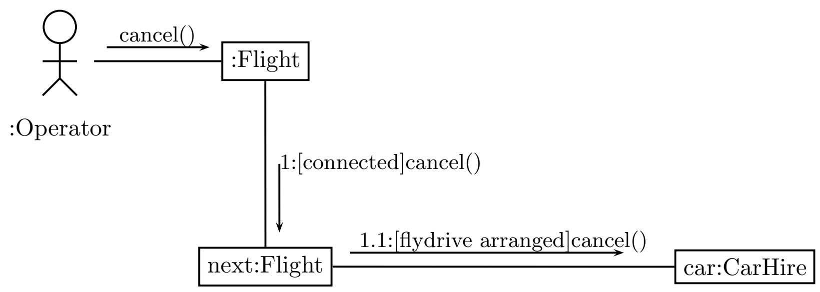

Figure 2: An interaction diagram for the "Cancel a flight" use case

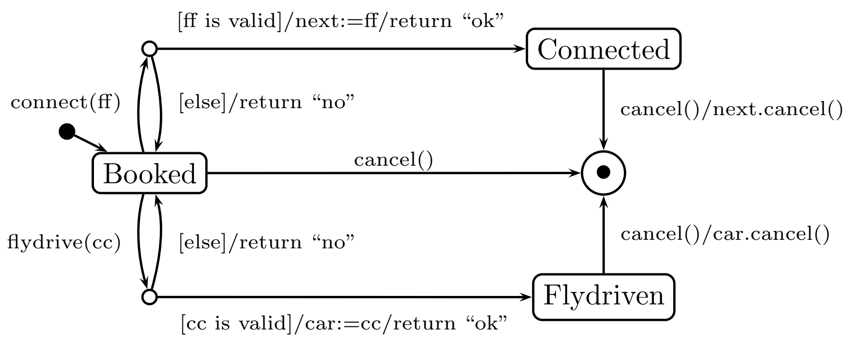

Figure 3: A state diagram for Flight Booking objects

3 Class Structuring in B

The B-Method is a formal method for developing software based on a single uniform notation known as Abstract Machine Notation (AMN) [Abrial 1996; Schneider 2001]. Using the B-Method, a system is modelled as an abstract machine consisting of some state variables and some operations on the state variables. Following Lano [Lano 1996], a class of objects can be modelled as a single abstract machine that carries a set with elements identifying the (currently) existing objects and a number of functions corresponding to the class attributes. For instance, a class C with attributes a, b, and c can be modelled by a B machine of the form:

MACHINE CAM
SETS $CSET$
VARIABLES $cset, afun, bfun, cfun$
INVARIANT
$cset \subseteq CSET \wedge$
$afun \in cset \rightarrow aTYPE \wedge$
$bfun \in cset \rightarrow bTYPE \wedge$
$cfun \in cset \rightarrow cTYPE \wedge$
...
END

where $CSET$ is the given set identifying all possible objects of class C and $aTYPE$, $bTYPE$, and $cTYPE$ are the types of attributes a, b, and c, respectively.

Associations between classes are in general translated into **INVARIANT** relationships between variables and given sets of the classes involved. This normally requires the use of **SEES** and **USES** clauses in the corresponding B machines. For instance, if class C mentioned above has a strictly one-to-one association relationship with class D, we can capture the relationship with a total injective function in the abstract machine for class D as follows:

MACHINE DAM
USES CAM
SETS $DSET$
VARIABLES $dset, assoc, \ldots$
INVARIANT
$dset \subseteq DSET \wedge$
$assoc \in dset \rightarrowtail cset \wedge$
......
END

One important advantage of modelling classes as abstract machines is that we can describe associations more precisely in AMN than we can do with a UML class diagram. For instance, the following abstract machines model the Flight and Car Hire classes as shown in Figure 1:

MACHINE $FlightAM$
USES $CarHireAM$
SETS $FLIGHTSET$
VARIABLES $flight, next, car$
INVARIANT $flight \in \mathbb{F}(FLIGHTSET) \wedge next \in flight \rightarrowtail\!\!\!\!\!\!\!\!+\; flight \wedge$
$next^{+} \cap id(flight) = \emptyset \wedge car \in flight \rightarrowtail\!\!\!\!\!\!\!\!+\; carhire \wedge dom(next) \cap dom(car) = \emptyset$
INITIALISATION $flight := \emptyset \parallel next := \emptyset \parallel car := \emptyset$
OPERATIONS
...
END

MACHINE $CarHireAM$
SETS $CARHIRESET$
VARIABLES $carhire$
INVARIANT $carhire \in \mathbb{F}(CARHIRESET)$
INITIALISATION $carhire := \emptyset$
OPERATIONS
...
END

The "connect" association is represented by a partial injective function *next* from the *flight* set to itself, i.e. a flight can be a connecting flight of at most one other flight and not every flight has a connecting flight. We have further stipulated that a flight cannot be a connecting flight of itself directly or indirectly via some intermediate connecting flight(s), i.e. $next^{+} \cap id(flight) = \emptyset$, which cannot be expressed in the UML class diagram. Note that $next^{+}$ is the non-reflexive transitive closure of *next*. The "flydrive" association is represented by another partial injective function *car*. The operations of the above abstract machines will be considered in Section 5.

4 Behavioural Modelling in CSP

A UML state diagram describes the state-machine behaviour of an object in terms of states, events, and transitions, as well as any actions that accompany the transitions. We may extract from a state diagram the desired sequences of *call events* happening to an object and describe them by a CSP process. For the discussion in this paper, we ignore other kinds of events such as change events and we use the following simplified syntax of CSP (for the time being):

$$\begin{aligned} P ::= {} & a \rightarrow P \mid c?x\langle E(x)\rangle \rightarrow P \mid d!v\{E(v)\} \rightarrow P \mid P_1 \,\Box\, P_2 \mid \\ & P_1 \sqcap P_2 \mid \textstyle\bigsqcap_{x \mid E(x)} P \mid \text{if } b \text{ then } P_1 \text{ else } P_2 \text{ end} \mid S(p) \end{aligned}$$

where a is a synchronisation event and can be in compound forms such as $a.i$, c and d are communication channels for inputs and outputs, respectively, x represents input variables, v represents output values, $E(x)$ is a predicate on x, b is a boolean expression, and $S(p)$ refers to a process expression parameterised by expression p. The process $a \rightarrow P$ first engages in event a and then behaves as P. The process $c?x\langle E(x)\rangle \rightarrow P$ is prepared to input a value along channel c into variable x provided that $E(x)$ is true and then behaves as P. The process $d!v\{E(v)\} \rightarrow P$ is prepared to output any value v for $E(v)$ is true along channel d and then behaves as P. $P_1 \Box P_2$ is a process that is prepared to engage in one of the initial events of either P_1 or P_2; once an initial event of P_i (i =1 or 2) chosen by the environment has happened, it behaves as P_i afterwards. $P_1 \sqcap P_2$ is a process that may choose to behave as either P_1 or P_2 but the choice is nondeterministic. $\bigsqcap_{x|E(x)} P$ is the indexed nondeterministic choice in which P may take any value x such that $E(x)$ is true. The if expression makes the choice between P_1 and P_2 depending on the boolean expression b in the usual way. Finally, $S(p)$ is a process name with a parameter expression p; we can define a process with the name $S(p)$ recursively by mentioning its own name $S(p')$ (where p' is an expression for the parameter p) in its definition.

The following CSP process describes the desired sequences of events happening to a Flight object according to the state diagram in Figure 3 :

$$
\begin{aligned}
\mathit{Flight} &\mathrel{\widehat=} \mathit{new} \rightarrow \mathit{Booked} \\
\mathit{Booked} &\mathrel{\widehat=} (\mathit{cancel} \rightarrow \mathit{Stop}) \\
&\quad \Box\ (\mathit{connect} \rightarrow (\mathit{Booked} \sqcap \mathit{Connected})) \\
&\quad \Box\ (\mathit{flydrive} \rightarrow (\mathit{Booked} \sqcap \mathit{Flydriven})) \\
\mathit{Connected} &\mathrel{\widehat=} \mathit{cancel} \rightarrow \mathit{Stop} \\
\mathit{Flydriven} &\mathrel{\widehat=} \mathit{cancel} \rightarrow \mathit{Stop}
\end{aligned}
$$

A Flight object is initially prepared to engage in the *new* event and then behaves as process *Booked*. *Booked* accepts either *cancel*, *connect*, or *flydrive* initially. If a *cancel* event takes place, the process will cease to engage in any further events (as described by the special process *Stop*). If a *connect* event occurs instead, the process may then choose to behave as *Booked* again from the beginning, or as another process *Connected*; the (nondeterministic) choice depends on the validity of the input parameter of the *connect* event, which we shall elaborate below. Similarly, if a *flydrive* event occurs initially, the *Booked* process may choose to return to its initial state, or behave as process *Flydriven*. Both the *Connected* and *Flydriven* processes are prepared for a *cancel* event initially and then stops.

We can model a state machine more precisely in CSP by taking into account any input parameters and (some) actions associated with individual events. For a Flight object, we have:

$$
\begin{aligned}
\mathit{Flight}(i) &\mathrel{\widehat=} \mathit{new}.i \rightarrow \mathit{Booked}(i)\\
\mathit{Booked}(i) &\mathrel{\widehat=} \mathit{cancel}.i \rightarrow \mathit{Stop}\\
&\quad \Box\ \mathit{connect}©.i?x : \mathit{FLIGHTSET} \rightarrow\\
&\qquad (\mathit{connect}®.i!\text{“no”} \rightarrow \mathit{Booked}(i)\\
&\qquad\quad \sqcap\ \mathit{connect}®.i!\text{“ok”} \rightarrow \mathit{Connected}(i))\\
&\quad \Box\ \mathit{flydrive}©.i?y : \mathit{CARHIRESET} \rightarrow\\
&\qquad (\mathit{flydrive}®.i!\text{“no”} \rightarrow \mathit{Booked}(i)\\
&\qquad\quad \sqcap\ \mathit{flydrive}®.i!\text{“ok”} \rightarrow \mathit{Flydriven}(i))\\
\mathit{Connected}(i) &\mathrel{\widehat=} \mathit{cancel}.i \rightarrow \mathit{Stop}\\
\mathit{Flydriven}(i) &\mathrel{\widehat=} \mathit{cancel}.i \rightarrow \mathit{Stop}
\end{aligned}
$$

We have parameterised process names and indexed events and actions with the identify of an object i. The “$\mathit{connect}©.i?x : \mathit{FLIGHTSET}$” notation stands for inputting a value of the set $\mathit{FLIGHTSET}$ along the $\mathit{connect}©.i$ channel and storing it in x; it models the receipt of the call event *connect* by object i with an input parameter x. On the other hand, the “$\mathit{connect}®.i!m$” notation stands for outputting the value of the expression m along the $\mathit{connect}®.i$ channel; it models the “return” of the call event *connect* by object i with an output parameter value m. The suffixing of *connect* with © and ® is necessary because channels are for one-way message passing in CSP.

Note that we have ignored the sending of call messages to other objects in the above CSP processes. Such actions could be modelled in CSP as synchronised input and output operations between objects if we model objects as concurrent processes. Instead, we can use a single “control-loop” process to describe the state-machine behaviour of a whole class of objects, *provided that we give up any concurrency among them.* The following process models the desired sequences of events and actions for the whole class of Flight objects:

$$
\begin{aligned}
\mathit{FlightSM} &\mathrel{\widehat=} F(\emptyset,\emptyset,\emptyset,\emptyset)\\
F(b,c,f,\phi) &\mathrel{\widehat=} \mathit{FLIGHTSET} \neq b \cup c \cup f \cup \phi\ \&\\
&\quad \textstyle\sqcap_{i|i\in\delta}\ \mathit{new}!i\{i \in \delta\} \rightarrow F(b \cup \{i\}, c, f, \phi)\\
&\qquad (\text{where } \delta = \mathit{FLIGHTSET} - (b \cup c \cup f \cup \phi))\\
&\quad \Box\ \mathit{connect}©?(i,x)\langle i \in b \wedge x \in \mathit{FLIGHTSET}\rangle \rightarrow
\end{aligned}
$$

$$
\begin{aligned}
&(connect Ⓡ!\text{“no”} \rightarrow F(b,c,f,\phi)\\
&\quad \sqcap\ connect Ⓡ!\text{“ok”} \rightarrow F(b-\{i\}, c \cup \{i\}, f, \phi))\\
\Box\ & \mathit{flydrive}Ⓒ?(i,y)\langle i \in b \wedge y \in CARHIRESET\rangle \rightarrow\\
&(\mathit{flydrive}Ⓡ!\text{“no”} \rightarrow F(b,c,f,\phi)\\
&\quad \sqcap\ \mathit{flydrive}Ⓡ!\text{“ok”} \rightarrow F(b-\{i\}, c, f \cup \{i\}, \phi))\\
\Box\ & cancel?i\langle i \in b \cup c \cup f\rangle \rightarrow\\
&\text{if } i \in c \text{ then}\\
&\quad F'(b, c-\{i\}, f, \phi \cup \{i\})\\
&\text{else if } i \in f \text{ then}\\
&\quad F(b, c, f-\{i\}, \phi \cup \{i\})\\
&\text{else}\\
&\quad F(b-\{i\}, c, f, \phi \cup \{i\})
\end{aligned}
$$

$$
F'(b,c,f,\phi) \mathrel{\widehat{=}} \textstyle\sqcap_{j \mid j \in b \cup c \cup f}\ cancel.j \rightarrow \left(\begin{array}{l}
\text{if } j \in c \text{ then}\\
\quad F'(b, c-\{j\}, f, \phi \cup \{j\})\\
\text{else if } j \in f \text{ then}\\
\quad F(b, c, f-\{j\}, \phi \cup \{j\})\\
\text{else}\\
\quad F(b-\{j\}, c, \phi \cup \{j\})
\end{array}\right)
$$

In order to keep track of the machine-state of each individual object, the *FlightSM* process carries four sets of object identities, b, c, f, and ϕ corresponding to the Booked, Connected, Flydriven, and Final (⨀) states, respectively. Every event except *new* now takes an object identity as an input parameter and is guarded according to the applicable state(s) of the event. For instance, the *connect* event is only applicable to objects in the Booked state. The *new* event does not take an input parameter but nondeterministically outputs a “free” object identity. The choice following the *connect* event is nondeterministic because they depend on the “connect” associations *among* objects and the process does not maintain such information.

Cancelling a connected flight is supposed to cancel also its chain of connecting flights; each Flight object in the chain receives a *cancel* call message and sends the same message to the next object in the chain, if there is one. The recursive definition of *FlightSM* lends itself to the description of such “recursive” calls among Flight objects. Each recursive call of $F'(b,c,f,\phi)$ in the above process definition begins with a *cancel* event for a connecting Flight object, whose identity is nondeterministic again because the process does not maintain information about connections (associations) among Flight objects.

Note that the sending of *cancel* call messages to Car Hire objects is still ignored in *FlightSM* above. Cancelling a “flydriven” flight is supposed to cancel

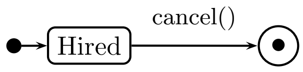

Figure 4: A state diagram for Car Hire objects

also its associated car hire; the Flight object should send a *cancel* call message to the associated Car Hire object. We shall take care of that after considering the control-loop process for Car Hire objects.

Figure 4 shows a state diagram for Car Hire objects. The state machine of a Car Hire object has only two states, namely Hired and Final, with Hired being the initial state. The only possible event at the Hired state is cancel(). The following control-loop process models the behaviour of the class of Car Hire objects:

$$
\begin{aligned}
\mathit{CarHireSM} &\mathrel{\widehat{=}} C(\emptyset, \emptyset) \\
C(h, \phi) &\mathrel{\widehat{=}} \mathit{CARHIRESET} \neq h \cup \phi \ \& \\
&\qquad\quad \textstyle\bigsqcap_{i \mid i \in \epsilon} \mathit{hire}!i\{i \in \epsilon\} \rightarrow F(h \cup \{i\}, \phi) \\
&\qquad\qquad \text{(where } \epsilon = \mathit{CARHIRESET} - (h \cup \phi)) \\
&\qquad\ \square\ \mathit{cancel}?i\langle i \in h\rangle \rightarrow C(h - \{i\}, \phi \cup \{i\})
\end{aligned}
$$

We have described the state-machine behaviour of two classes of objects separately by two CSP control-loop processes. We can take a further step to use a single control-loop process to describe the state-machine behaviour of both classes, ie. the entire system. Such a process can be obtained by “merging” the two that we have already defined as follows:

$$
\begin{aligned}
\mathit{SystemSM} &\mathrel{\widehat{=}} S(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \\
S(b, c, f, \phi_f, h, \phi_c) &\mathrel{\widehat{=}} \mathit{FLIGHTSET} \neq b \cup c \cup f \cup \phi_f \ \& \\
&\qquad\quad \textstyle\bigsqcap_{i \mid i \in \delta} \mathit{new}_f!i\{i \in \delta\} \rightarrow S(b \cup \{i\}, c, f, \phi_f, h, \phi_c) \\
&\qquad\qquad \text{(where } \delta = \mathit{FLIGHTSET} - (b \cup c \cup f \cup \phi_f)) \\
&\quad \square\ \mathit{connect}Ⓒ?(i, x)\langle i \in b \wedge x \in \mathit{FLIGHTSET}\rangle \rightarrow \\
&\quad\ (\mathit{connect}Ⓡ!\text{“no”} \rightarrow S(b, c, f, \phi_f, h, \phi_c) \\
&\qquad\quad \sqcap\ \mathit{connect}Ⓡ!\text{“ok”} \rightarrow S(b - \{i\}, c \cup \{i\}, f, \phi_f, h, \phi_c)) \\
&\quad \square\ \mathit{flydrive}Ⓒ?(i, y)\langle i \in b \wedge y \in \mathit{CARHIRESET}\rangle \rightarrow \\
&\quad\ (\mathit{flydrive}Ⓡ!\text{“no”} \rightarrow S(b, c, f, \phi_f, h, \phi_c) \\
&\qquad\quad \sqcap\ \mathit{flydrive}Ⓡ!\text{“ok”} \rightarrow S(b - \{i\}, c, f \cup \{i\}, \phi_f, h, \phi_c))
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad\quad\quad\quad\quad\quad \Box\ \mathit{cancel_f}?i\langle i \in b \cup c \cup f\rangle \to \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{if } i \in c \text{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S'(b, c - \{i\}, f, \phi_f \cup \{i\}, h, \phi_c) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{else if } i \in f \text{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \sqcap_{j|j\in h}\ \mathit{cancel_c}.j \to \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S(b, c, f - \{i\}, \phi_f \cup \{i\}, h - \{j\}, \phi_c \cup \{j\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{else} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S(b - \{i\}, c, f, \phi_f \cup \{i\}, h, \phi_c) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \Box\ \mathit{CARHIRESET} \neq h \cup \phi_c\ \& \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \sqcap_{i|i\in\epsilon}\ \mathit{new_c}!i\{i \in \epsilon\} \to S(b, c, f, \phi_f, h \cup \{i\}, \phi_c) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\text{where } \epsilon = \mathit{CARHIRESET} - (h \cup \phi_c)) \\
S'(b, c, f, \phi_f, h, \phi_c) \mathrel{\widehat=} \sqcap_{j|j\in b\cup c\cup f}\ \mathit{cancel_f}.j \to \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{if } j \in c \text{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S'(b, c - \{j\}, f, \phi_f \cup \{j\}, h, \phi_c) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{else if } j \in f \text{ then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \sqcap_{j|j\in h}\ \mathit{cancel_c}.j \to \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S(b, c, f - \{j\}, \phi_f \cup \{j\}, h - \{j\}, \phi_c \cup \{j\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{else} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S(b - \{j\}, c, \phi_f \cup \{j\}, h, \phi_c)
\end{array}
$$

The sets of object identities carried by *FlightSM* and *CarHireSM* have been brought together as parameters of *SystemSM*. The bodies of the two former processes have been merged to give the main body of *SystemSM*, which offers a choice of all call events for the system except the internal $\mathit{cancel_c}$ event for Car Hire objects. Note that we have used subscripts f and c (for Flight and Car Hire, respectively) to resolve the name clashes among parameters and events as in $\mathit{cancel_f}$ and $\mathit{cancel_c}$. Recursive $\mathit{cancel_f}$ calls for connected Flight objects are handled by $S'(b, c, f, \phi_f, h, \phi_c)$ in the same way as $F'(b, c, f, \phi)$ did. On the other hand, cancelling a "flydriven" flight results in a $\mathit{cancel_c}$ which models the sending of a call from the flydriven Flight object to the associated Car Hire object, whose identity is nondeterministic because the control-loop process does not maintain information about associations between the two classes of objects.

5 Abstract Machine Operations

The information for resolving the nondeterministic choices in *SystemSM* is actually available from the abstract machines defined earlier in Section 3. For instance, the nondeterministic choice following the *connect* event depends on the validity of the input parameter—x is valid if it identifies an object which is in either the Booked, Connected, or Flydriven states and connecting flight i and flight x will *not* lead to circular flight connections. While the machine-state of Flight object x can be determined within the control-loop process itself, information about flight connections can only be obtained from *FlightAM* through its abstract machine operations such as the following one:

$\mathit{reply} \leftarrow \mathbf{connected}(x, i) \mathrel{\widehat{=}}$
  PRE $x \in \mathit{FLIGHTSET} \wedge i \in \mathit{FLIGHTSET} \wedge x \neq i$ **THEN**
    IF $(x \mapsto i) \in \mathit{next}^+$ **THEN** $\mathit{reply} := \mathit{yes}$
    ELSE $\mathit{reply} := \mathit{no}$ **END**
  END

On the other hand, whenever a Flight object successfully changes its state from Booked to Connected, the following operation should be executed to update the *FlightAM* abstract machine:

$\mathbf{connect}(\mathit{ff}, \mathit{gg}) \mathrel{\widehat{=}}$
  PRE $\mathit{ff} \in \mathit{flight} \wedge \mathit{gg} \in \mathit{flight} \wedge \mathit{ff} \neq \mathit{gg}$
      $\wedge\ \mathit{ff} \notin \mathit{dom}(\mathit{next}) \cup \mathit{dom}(\mathit{car}) \wedge (\mathit{gg} \mapsto \mathit{ff}) \notin \mathit{next}^+$ **THEN**
    $\mathit{next}(\mathit{ff}) := \mathit{gg}$
  END

While the above abstract machine operations can be used to maintain and provide class-structure information for the control-loop process, their preconditions constrain the ways in which they are called. In other words, the control-loop process is consistent with the abstract machines if and only if the former only calls the operations of the later within their preconditions.

On the other hand, instead of checking the consistency of the control-loop process against each abstract machine individually, we can make use of the structuring mechanisms in B AMN to combine the two abstract machines (*FlightAM* and *CarHireAM*) into one as follows:

MACHINE *SystemAM*
EXTENDS *FlightAM*, *CarHireAM*
OPERATIONS
$\mathbf{cancel}_s(\mathit{ff}) \mathrel{\widehat{=}}$
  PRE $\mathit{ff} \in \mathit{dom}(\mathit{car})$ **THEN**
    $\mathit{cancel}_f(\mathit{ff}) \parallel \mathit{carhire} := \mathit{carhire} - \{\mathit{car}(\mathit{ff})\}$
  END
  END
END

The operation $\mathbf{cancel}_s$ deletes a "flydriven" Flight object and the associated Car Hire object at the same time. It has to be defined at the system level because it updates the state variables of both included machines. The operation also preserves the invariants of both included machines.

6 Checking Consistency

To check the consistency between a CSP control-loop process and its corresponding B abstract machine, we can make use of Treharne and Schneider's coupling between CSP and B [Treharne and Schneider 1999; Schneider and Treharne 2002]. Here we only illustrate the application of their coupling with our example. For a detailed account of the coupling itself, the reader is referred to [Treharne and Schneider 1999; Schneider and Treharne 2002].

We first need to incorporate the calling of appropriate abstract machine operations into the control-loop process. This requires an extended version of CSP as given in [Schneider and Treharne 2002] as follows:

$$P ::= a \to P \mid \ldots \mid S(p) \mid e\boldsymbol{?}v \to P \mid e\boldsymbol{!}x\{E(x)\} \to P \mid e\boldsymbol{?}v\boldsymbol{!}x\{E(x)\} \to P$$

The three additional options at the end are used for "calling" abstract machine operations with either input ($\boldsymbol{?}$) parameters, output ($\boldsymbol{!}$) parameters, or both, respectively, where e is a communication channel corresponding to an abstract machine operation. Notice that the emphasised symbols "$\boldsymbol{?}$" and "$\boldsymbol{!}$" are reserved for abstract machine operation calls. We can now elaborate the control-loop process *SystemSM* with appropriate abstract machine operation calls as shown partially below:

$$
\begin{array}{l}
\mathit{SystemSM} \mathrel{\widehat{=}} S(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)\\
S(b,c,f,\phi_f,h,\phi_c) \mathrel{\widehat{=}}\\
\quad \ldots\\
\quad \Box\ \mathit{connect}©?(i,x)\langle i \in b \wedge x \in \mathit{FLIGHTSET}\rangle \to\\
\qquad \text{if } x = i \vee x \notin b \cup c \cup f \text{ then } \mathit{connect}®!\text{“no”} \to S(b,c,f,\phi_f,h,\phi_c)\\
\qquad \text{else } \mathit{connected}\boldsymbol{?}(x,i)\boldsymbol{!}\mathit{rr}\{\mathit{rr} \in \{\mathit{yes},\mathit{no}\}\} \to\\
\qquad\qquad \text{if } \mathit{rr} = \mathit{yes} \text{ then } \mathit{connect}®!\text{“no”} \to S(b,c,f,\phi_f,h,\phi_c)\\
\qquad\qquad \text{else } \mathit{connect}\boldsymbol{?}(i,x) \to \mathit{connect}®!\text{“ok”} \to S(b-\{i\},c\cup\{i\},f,\phi_f,h,\phi_c)\\
\quad \ldots
\end{array}
$$

Based on Treharne and Schneider's coupling between B and CSP [Treharne and Schneider 1999], we can ascertain that *SystemSM* only calls those abstract machine operations within their preconditions if we can find a control loop invariant (CLI) which holds at each recursive call within the body of *SystemSM*, i.e.:

$$CLI \wedge I \Rightarrow [BBODY_{S(b,c,f,\phi_f,h,\phi_c)}]CLI \tag{1}$$

$$CLI \wedge I \Rightarrow [BBODY_{S'(b,c,f,\phi_f,h,\phi_c,n)}]CLI \tag{2}$$

where I is the invariant of abstract machine $SystemAM$; $BBODY_{S(b,c,f,\phi_f,h,\phi_c)}$, and $BBODY_{S'(b,c,f,\phi_f,h,\phi_c,n)}$ are the translation of the CSP expressions used in defining the parameterised process $S(b,c,f,\phi_f,h,\phi_c)$, and $S'(b,c,f,\phi_f,h,\phi_c,n)$, respectively, into B AMN operations. The notation $[S]P$ denotes the weakest precondition for operation S to achieve P. Figure 5 shows the translation function $\{S(p)\} = BBODY_{S(p)}$ which is defined inductively based on the extended syntax of CSP as given in [Treharne and Schneider 1999; Schneider and Treharne 2002].

An appropriate CLI for *SystemSM* is:

$$\begin{aligned}&\mathit{flight} = b \cup c \cup f \wedge \mathit{flight} \cap \phi_f = \emptyset \wedge b \cup c \cup f \cup \phi_f \subseteq FLIGHTSET\\&\wedge\ c = dom(next) \wedge f = dom(car) \wedge b \cap c \cap f = \emptyset \wedge \mathit{carhire} = h\\&\wedge\ \mathit{carhire} \cap \phi_c = \emptyset \wedge h \cup \phi_c \subseteq CARHIRESET \wedge n \in \mathit{flight} \cup \phi_f\end{aligned}$$

The proof of (1) and (2) can be found in [Yeung 2003].

7 Evaluation

The consistency-checking approach illustrated in this paper has been applied in the following situations:

1. A typical class (e.g. Student) is modelled by a B machine with some invariant properties (e.g. all student IDs must be unique) and operations for creating and destroying its objects and modifying object attributes. Operations have to be properly pre-conditioned to preserve the invariant of the machine. A corresponding state diagram specifying the acceptable ordering of calls to these operations with *informal* guards (e.g. object exists) can be *formalised* in CSP and verified as always respecting the operations' preconditions.

2. Similarly, a single class of objects with unary associations among themselves (e.g. The Flight class) can be modelled by a B machine with operations properly pre-conditioned to preserve any invariant constraints upon the unary associations (e.g. no loops of connecting flights). A corresponding state diagram can be similarly formalised and verified.

3. A call event may trigger other call events within the same class. Modelling this in B alone (e.g. [Ledang and Souquières 2001]) involves elaborate machine structuring. Modelling dynamic behaviour in CSP is more natural and

$$
\begin{aligned}
\{P \,\Box\, Q\}_\rho \mathrel{\widehat{=}}\; & \textbf{SELECT}\ \mathit{true}\ \textbf{THEN}\ \{P\}_\rho \\
& \textbf{WHEN}\ \mathit{true}\ \textbf{THEN}\ \{Q\}_\rho \\
& \textbf{END} \\
\{P \sqcap Q\}_\rho \mathrel{\widehat{=}}\; & \textbf{CHOICE}\ \{P\}_\rho\ \textbf{OR}\ \{Q\}_\rho\ \textbf{END} \\
\{\textstyle\bigsqcap_{x|E(x)} P\} \mathrel{\widehat{=}}\; & \textbf{ANY}\ x_b\ \textbf{WHERE}\ \rho[E(x_c)]\ \textbf{THEN} \\
& \quad \{P\}_{\rho \oplus \{x_c \mapsto x_b\}} \\
& \textbf{END} \\
\{\text{if } x_c \text{ then } P \text{ else } Q \,\text{end}\}_\rho \mathrel{\widehat{=}}\; & \textbf{IF}\ \rho[x_c]\ \textbf{THEN}\{P\}_\rho \textbf{ELSE}\ \{Q\}_\rho\ \textbf{END} \\
\{a \rightarrow P\}_\rho \mathrel{\widehat{=}}\; & \{P\}_\rho \\
\{c?x_c\langle E(x_c)\rangle \rightarrow P(x_c)\}_\rho \mathrel{\widehat{=}}\; & \textbf{ANY}\ x_b\ \textbf{WHERE}\ \rho[E(x_c)]\ \textbf{THEN} \\
& \quad \{P\}_{\rho \oplus \{x_c \mapsto x_b\}} \\
& \textbf{END} \\
\{d!w_c\{E(w_c)\} \rightarrow P\}_\rho \mathrel{\widehat{=}}\; & \textbf{PRE}\ \rho[E(w_c)]\ \textbf{THEN}\ \{P\}_\rho\ \textbf{END} \\
\{E_{name}\mathbf{?}x_c \rightarrow P\}_\rho \mathrel{\widehat{=}}\; & \mathit{name}(\rho[x_c]);\ \{P\}_\rho \\
\{E_{name}\mathbf{!}w_c\{T(w_c)\} \rightarrow P(w_c)\}_\rho \mathrel{\widehat{=}}\; & w_b \leftarrow \mathit{name};\ \{P\}_{\rho \oplus \{w_c \mapsto w_b\}} \\
\{E_{name}\mathbf{?}x_c\mathbf{!}w_c\{T(w_c)\} \rightarrow P(w_c)\}_\rho \mathrel{\widehat{=}}\; & w_b \leftarrow \mathit{name}(\rho[x_c]);\ \{P\}_{\rho \oplus \{w_c \mapsto w_b\}} \\
\{S(p')\}_\rho \mathrel{\widehat{=}}\; & c_b := \rho[p']
\end{aligned}
$$

Note: ρ is an environment binding for mapping variables and expressions into their values. c_b is a collection of AMN control variables corresponding to the parameters of the CSP process.

Figure 5: Translation function for the body of a CSP process $\{S(p)\}$

keep the underlying B machine structure relatively simple. CSP also lends itself conveniently to modelling recursive calls.

4. Two classes of objects, with one or more binary associations between the two classes, can first be modelled by two separate B machines with their own invariants and operations. They are then combined together to form a composite machine with invariant constraints due to the binary associations. On the other hand, the two corresponding state diagrams are separately formalised in CSP and then merged together to form a single process governing the ordering of operation calls to and between the two classes of objects.

Since B machines and CSP processes are compositional, the approach can, in principle, be generalised to three or more classes, although future work is needed to see whether they are any practical limitations.

The above situations are relevant to any system with non-trivial associations among classes and operations that update the associations. Such systems include data-intensive enterprise information systems that often have elaborate associations.

Previous attempts to model classes as B machines combine the static class-structure view and the dynamic state-machine view in a single formalism. The present approach adopts a clear-cut separation of concerns: B machines are used to model the static (invariant) properties in a class diagram whereas a CSP process is used to formalise the dynamic behaviour in a state diagram. Advantages of this approach include clarity and better traceability between UML and the formal descriptions. Another advantage is that a recursive CSP process lends itself to the modelling of recursive calls among a class of objects. The modelling of recursive calls in B alone has not yet been achieved (see, e.g. [Lano 1996; Meyer and Souquières 1999; Ledang and Souquières 2001]).

On the other hand, the use of a single CSP control-loop process to capture the dynamic behaviour of a system of interacting objects rules out any concurrency among the objects. While the lack of concurrency is reasonable for data-intensive enterprise information systems, it becomes a major weakness when trying to extend the present approach to real-time systems. Another weakness is the lack of support tools for directly formalising state diagrams in CSP (although some prototype tools have been proposed [Ng and Butler 2002, 2003]) and for automating the translation from CSP into B as defined in Figure 5. Further work is needed to develop these tools which would make the present approach more scalable for large systems. On the other hand, proof assistance is available from support tools for the B-method, including the B-Toolkit [B-Core 1996] and AtelierB [STERIA 1998], which are useful for establishing the control loop invariant as discussed in section 6. Finally, the use of the FDR model checking tool [FDR 2003] for CSP is rather limited since the control-loop process does not involve any concurrency.

8 Related Work

Various formal approaches to formalising the UML class and state models have been proposed. Kim and Carrington [2000] provided translation rules for mapping the UML class and state models into Object-Z but did not address the consistency issue. DeLoach and Hartrum [2000] formalised the class and state models in O-Slang, an algebraic specification language, but again they did not address the consistency issue. Berry and Weber [1997] formalised state diagrams in Z [Spivey 1992] with an emphasis on embedded control systems. They addressed the consistency between the functional aspect of the class model (i.e. data structure and I/O parameters) and the state model by computational induction. However, they did not address the consistency between the architectural aspect of the class model (i.e. constraints due to the associations and multiplicities) and the state model.

Davies and Crichton [2002] formalised the various UML diagrams using the notation in CSP but they did not address the consistency between the class and state models. Ng and Butler [2002, 2003] formalised UML state diagrams in CSP and developed support tools for translating the former into the latter.

Lano [1996] illustrated the translation of OMT diagrams for the class and state models [Rumbaugh et al. 1991] into B [Abrial 1996]; Meyer and Souquières [1999] presented a comprehensive scheme for the translation. In order to model object interaction in B involving multi-layer (non-recursive) calls, Ledang and Souquières [2001] proposed an approach involving layers of implementation machines corresponding to the static calling structures (hierarchies) of class operations. A class operation that calls some other class operation(s) is defined at an appropriate layer below which the called operation(s) is/are defined. None of these approaches, however, supports the modelling of recursive calls, i.e. a call event leading to the sending of the same call to another object of the same class, eg. the cancel() call event in Figure 2.

Tenzer and Stevens [2003] proposed the modelling of objects that receive recursive calls as recursive state machines [Alur et al. 2001]. They did not address the consistency between the class and state models.

A single formalism is used in each of the above approaches to formalise the UML class and state models. McUmber and Cheng [2001] formalised the state model in Promela/SPIN [Holzmann 1997] and they drew on the work of Bourdean and Cheng [1995] on formalising the class model using algebraic specification. However, they did not address the consistency between the two models.

9 Conclusion and Further Work

This paper has presented an example of applying of a pair of integrated formal methods, namely B and CSP, to the checking of consistency between the class model and state model of UML. The integrated approach allows the two formal methods to be applied separately and efficiently, with the help of support tools, to the two UML models. Consistency between the two models can simply be established by Treharne and Schneider's coupling between CSP and B.

Further work is needed to generalise the integrated approach to handle more complex state machines as well as more elaborate class structures involving generalisation and specialisation. These can be achieved by developing more realistic case studies. Support tools for translating state diagrams into CSP and for translating CSP into B can be further developed to facilitate the integrated approach.

Acknowledgements

This work is supported by a study leave from Lingnan University, Hong Kong. The author would like to thank Helen Treharne, Steve Schneider, and Damien Karkinsky for their discussion and help during his visit at Royal Holloway, University of London.

References

[Abrial 1996] Abrial, J. R.: "The B-Book"; Cambridge University Press (1996).

[Alur et al. 2001] Alur, R., Etessami, K., and Yannakakis, M.: "Analysis of recursive state machines." Proceedings of Computer Aided Verification 2001; Lect. Notes Comp. Sci. 2102, Springer (2001), 207–220.

[STERIA 1998] "Atelier B, Manuel Utilisateur, Version 3.5" (1998).

[B-Core 1996] "B-Toolkit User's Manual, Release 3.2" (1996).

[Berry and Weber 1997] Berry, D. M., and Weber, M.: "A Pragmatic, Rigorous Integration of Structural and Behavioral Modeling Notations."; Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97), IEEE (1997).

[Bolognesi and Brinksma 1988] Bolognesi, T., and Brinksma, E.: "Introduction to the ISO specification language LOTOS."; Computer Networks and ISDN Systems, 14, 1 (1988) 25–59.

[Bourdeau and Cheng 1995] Bourdeau, R. H., and Cheng, B. H. C.: "A Formal Semantics for Object Model Diagrams."; IEEE Transactions on Software Engineering, 21, 10 (1995) 799-821.

[Bowman and Derrick 1999] Bowman, H., and Derrick, J.: "A junction between state based and behavioural specification."; Proceedings of 4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Kluwer (1999) 213–237.

[Davies and Crichton 2002] Davies, J., and Crichton, C.: "Concurrency and Refinement in the Unified Modeling Language"; Electronic Notes in Theoretical Computer Science, 70, 3 (2002).

[DeLoach and Hartrum 2000] DeLoach, S. A., and Hartrum, T. C.: "A Theory-Based Representation for Object-Oriented Domain Models"; IEEE Transactions on Software Engineering, 26, 6 (2000) 500-517.

[Engels et al. 2001] Engels, G., Küster, J. M., Heckel, R., and Groenewegen, L.: "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models." Proceedings of the 8th European Software Engineering Conference, ACM Press (2001) 186–195.

[FDR 2003] "Failures-Divergence Refinement: FDR2 User Manual" (2003).

[Harel 1987] Harel, D.: "Statecharts: A visual formalism for complex systems"; Science of Computer Programming, 8, 3 (1987) 231–274.

[Hoare 1985] Hoare, C. A. R.: "Communicating Sequential Processes"; Prentice Hall (1985).

[Holzmann 1997] Holzmann, G.: "The model check SPIN"; IEEE Transactions on Software Engineering, 23, 5 (1997).

[Jacobson et al. 1992] Jacobson, I., Griss, M., Jonsson, P., and Oevergaard, G.: "Object-Oriented Software Engineering: A Use Case Driven Approach"; Addison-Wesley (1992).

[Kim and Carrington 2000] Kim, S.-K., and Carrington, D.: "An Integrated Framework with UML and Object-Z for Developing a Precise and Understandable Specification: The Light Control Case Study"; Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00), IEEE (2000) 240–248.

[Koskimies et al. 1998] Koskimies, K., Systä, T., Tuomi, J., and Männistö, T.: "Automated Support for Modeling OO Software"; IEEE Software (Jan–Feb 1998) 87–94.

[Lano 1996] Lano, K.: "The B Language and Method: A Guide to Practical Formal Development"; Springer-Verlag (1996).

[Ledang and Souquières 2001] Ledang, H., and Souquières, J.: "Integrating UML and B Specification Techniques"; GI2001 Workshop: Integrating Diagrammatic and Formal Specification Techniques, Universität Wien, Österreich (2001) .

[McUmber and Cheng 2001] McUmber, W. E. and Cheng, B. H. C.: "A general framework for formalizing UML with formal languages"; Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society (2001) 433–442.

[Meyer and Souquières 1999] Meyer, E., and Souquières, J.: "A Systematic Approach to Transform OMT Diagrams to a B Specification"; FM'99 – B users group meeting – applying B in an industrial context : Tools, lessons and techniques, Lect. Notes Comp. Sci. 1708, Springer (1999) 875–895.

[Ng and Butler 2002] Ng, M. Y., and Butler, M.: "Tool Support for Visualizing CSP in UML"; Proc. ICFEM 2002, Lect. Notes Comp. Sci. 2495, Springer (2002) 287–298.

[Ng and Butler 2003] Ng, M. Y., and Butler, M.: "Towards Formalizing UML State Diagrams in CSP"; Proc. 1st IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society (2003) 138–147

[OMG 2001] "OMG Unified Modeling Language Specification Version 1.4" (September 2001).

[Rumbaugh et al. 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.: "Object-Oriented Modeling and Design"; Prentice Hall (1991).

[Schneider 2001] Schneider, S.: "The B-Method: An Introduction"; Palgrave (2001).

[Schneider and Treharne 2002] Schneider, S., and Treharne, H.: . "Communicating B Machines"; ZB'2002 – formal specification and development in Z and B, Lect. Notes Comp. Sci. 2272, Springer (2002) 416–435.

[Spivey 1992] Spivey, J. M.: "The Z Notation: A Reference Manual (Second ed.)"; Prentice Hall (1992).

[Tenzer and Stevens 2003] Tenzer, J., and Stevens, P.: . "Modelling recursive calls with UML state diagrams"; Proceedings of FASE 2003, Lect. Notes Comp. Sci. 2621, Springer (2003) 135–149.

[Treharne and Schneider 1999] Treharne, H., and Schneider, S.: "Using a process algebra to control B OPERATIONS"; IFM'99 1st International Conference on Integrated Formal Methods, Springer (1999) 437–457.

[Yeung 2003] Yeung, W. L.: "Checking Consistency Between Class Structure and State Machines Based on CSP and B"; Tech. Rep., Department of Information Systems, Lingnan University, Hong Kong (2003) `http://cptra.ln.edu.hk/~wlyeung`