Formalising Inter-model Consistency of the UML

K. Lano, D. Clark, K. Androutsopoulos

King's College London, Strand, London, WC2R 2LS

Abstract. In this paper we define consistency rules between class diagrams, object diagrams and statecharts in a subset of UML, based on the UML metamodel and a formal semantics for these models.

1 Introduction

Many errors in a system specification can be detected at an early stage if adequate specification consistency checks are available. Errors at the early development stages are the most expensive to correct if they are undetected, so checks can have a major benefit. The UML [6] consists of several related notations, including class diagrams, object diagrams and statemachines. These models all present different aspects or views of a system specification or design:

1. Class diagrams – define the static structural elements of the system, the classes and their attributes, operations and invariants, and relationships between classes.
2. Object diagrams – define particular arrangements of objects and links between them, showing a specific state (snapshot) of the system.
3. Statecharts – define possible modes of classes, and the response of objects to the methods of their classes.

There are already internal consistency rules for these models defined in UML, such as the restriction that there can be no cycles in the inheritance relation [7]. However, semantic consistency constraints between different models of the same system are generally not specified.

In the following we will define a precise semantics for the RSDS [5] subset of UML, and use this to define internal and inter-model consistency rules.

2 RSDS

RSDS (Reactive System Design Support) specifies reactive systems using statemachines, class diagrams, object diagrams and invariants. An RSDS system Sys consists of a single class diagram CD_{Sys}, multiple object diagrams, and one statemachine for each class of CD_{Sys}.

Classes are stereotyped to be of one of three kinds: *sensors* or *interface* components, which receive input events from the outside world; *controllers*, which compute the system response to these events, and *actuator* components, which carry out the response.

2.1 RSDS Metamodel

The RSDS metamodel is a strict subset of that of UML. Most of the content of an RSDS specification in this metamodel occurs in instances of the *Constraint* metaclass, which express RSDS invariants. In particular, pre-post specifications of operations are expressed as action invariants. The *ownerScope* of all attributes is *instance*.

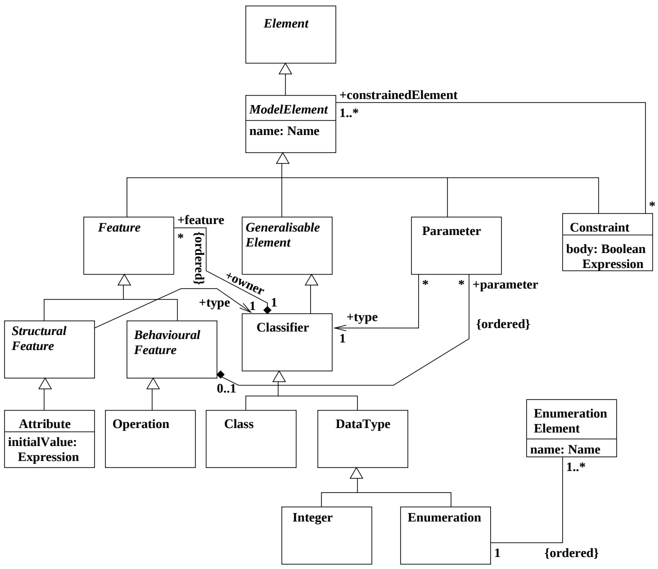

Fig. 1. RSDS Core Metamodel: Elements

3 Class Diagrams

A class diagram describes a set $\{C_1, \ldots, C_n\}$ of classes, and associations $\{r_1, \ldots, r_m\}$, each association has a set of association ends, each end has a corresponding class and multiplicity (Figure 2).

For a class diagram CD we define a 1st order CTL theory Δ_{CD} [3]. This consists of:

1. Type symbols $@C$ for each class C, and for standard primitive types *bool*, *int*, etc;
2. Attributes $\overline{C} : \mathbb{F}(@C)$ for each class C, and $\overline{r} : \mathbb{F}(@D_1 \times \ldots \times @D_k)$ for each association r, with end points at classes D_1 to D_k;
3. Attributes

$$att : @C \to T$$

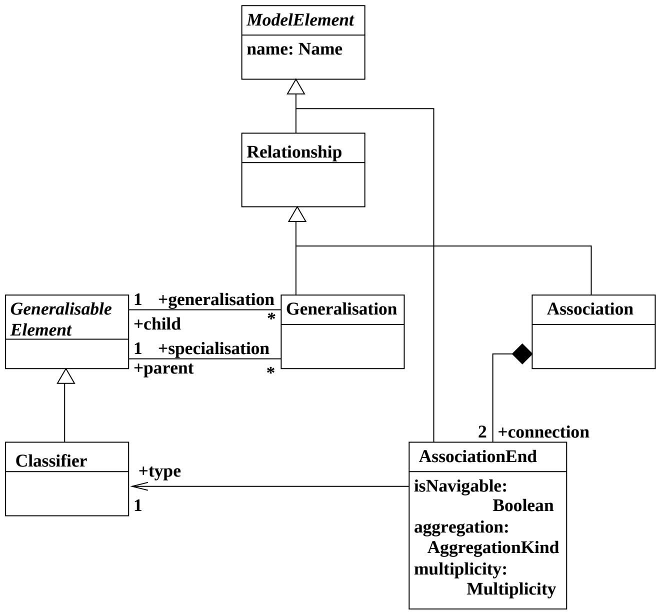

Fig. 2. RSDS Core Metamodel: Relationships

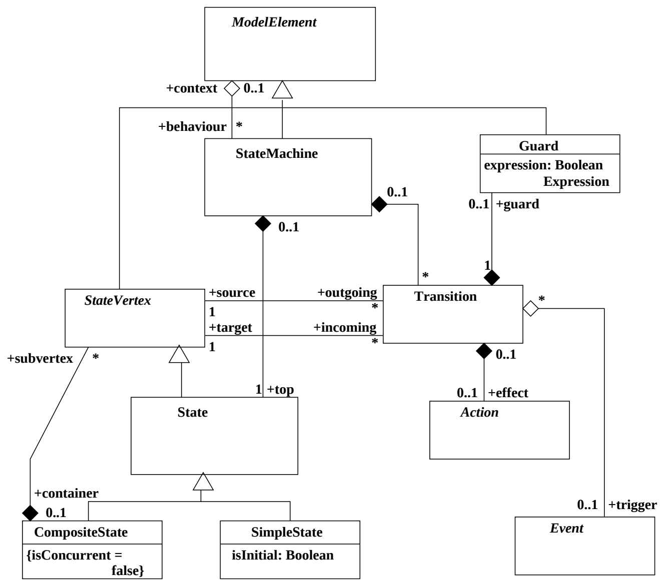

Fig. 3. RSDS Core Metamodel: Statemachines

for each attribute $att : T$ of class C. We may write $x.att$ for $att(x)$ to emphasise that these represent instance-level variables. In addition the mode of C – the current state of its corresponding statemachine Sm (Section 5) – is included as an attribute $sm : @C \to States_{Sm}$;

4. Action symbols

$$op : @C \times T_1 \times \ldots \times T_p \to T$$

   for each operation $op(T_1, \ldots, T_p) : T$ of class C. The syntax $x.op(v_1, \ldots, v_p)$ can be used for $op(x, v_1, \ldots, v_p)$;

5. Axioms $AG(@D \subseteq @C)$ and $AG(\overline{D} \subseteq \overline{C})$ if D inherits C;
6. Axioms

$$AG(\overline{r} \subseteq \overline{D_1} \times \ldots \times \overline{D_k})$$

   for each association r, with end points at classes D_1 to D_k;

7. Axioms defining the allowed cardinality of an association: if c_k is the cardinality of association r at end k, expressed as a set c_k of elements of $\mathbb{N}$, then for each particular tuple $(x_1, \ldots, x_{k-1}) \in \overline{D_1} \times \ldots \times \overline{D_{k-1}}$ at the other ends, the number of elements $x \in \overline{D_k}$ such that $(x_1, \ldots, x_{k-1}, x) \in \overline{r}$ must be in c_k. Similarly for each of the other indexes;
8. Actions for creating and deleting instances:

$$new_C : @C \to$$
$$kill_C : @C \to$$

   and axioms defining their effect:

$$AG(\forall X : \mathbb{F}(@C) \cdot \overline{C} = X \ \wedge \ new_C(x) \\ \Rightarrow \ AX(\overline{C} = X \cup \{x\}))$$

$$AG(\forall X : \mathbb{F}(@C) \cdot \overline{C} = X \ \wedge \ kill_C(x) \\ \Rightarrow \ AX(\overline{C} = X - \{x\}))$$

9. Actions for linking and unlinking tuples in relationships:

$$link_r : @D_1 \times \ldots \times @D_k \to$$
$$unlink_r : @D_1 \times \ldots \times @D_k \to$$

   for each association between D_1 to D_k. The axioms for these actions are:

$$AG(\forall X : \mathbb{F}(@D_1 \times \ldots \times @D_k) \cdot \overline{r} = X \ \wedge \ link_r(x_1, \ldots, x_k) \\ \Rightarrow \ AX(\overline{r} = X \cup \{(x_1, \ldots, x_k)\}))$$

$$AG(\forall X : \mathbb{F}(@D_1 \times \ldots \times @D_k) \cdot \overline{r} = X \ \wedge \ unlink_r(x_1, \ldots, x_k) \\ \Rightarrow \ AX(\overline{r} = X - \{(x_1, \ldots, x_k)\}))$$

10. Pseudoattributes

$$bs : @A \rightarrow \mathbb{F}(@B)$$
$$as : @B \rightarrow \mathbb{F}(@A)$$

for each binary relation r between classes A and B. The notations $y \in bs(x)$ and $x \in as(y)$ abbreviate $(x, y) \in r$.
11. Axioms defining the initial state of attributes:

$$new_C(x) \;\Rightarrow\; AX(att(x) = val)$$

where *val* is the initial value of *att* specified in the class diagram;
12. Locality axioms, for each class C and attribute $att : T$ of C:

$$AG(\forall\, v : T \cdot\; att = v \;\wedge\; \forall\, x_1 \cdot \neg\, op_1(x_1) \;\wedge\; \ldots \;\wedge\; \forall\, x_h \cdot \neg\, op_h(x_h) \\ \Rightarrow\; AX(att = v))$$

where the op_h are all the operations of C.

3.1 Invariants

Invariants may be used in the class diagram to specify how the states of attributes within an object, and the states of different objects, should be related to each other. These invariants can be of the forms:

1. Static (or single state) invariants: $P \;\Rightarrow\; Q$ where both P and Q consist only of constraints on current states of objects in the system. No temporal operators or event names are used. For example the constraint, attached to class C, that

   $$att \leq 10$$

   for an integer attribute *att* of C.
   The other case of static constraints are those attached to associations, for example, the constraint, attached to a relation r between classes Sw, $S2$ and Bm in the feed belt of a simple production cell (Figure 4), where $s2$ is the state of the blank sensor at the end of the belt, *sw* the belt switch and *bm* the belt motor:

   ```
   sw = On & s2 = Off  =>  bm = On
   ```

   "if the switch is on and the belt end sensor is off, the belt motor is on".
2. Operational invariants $\alpha\ \&\ P \;\Rightarrow\; AX(Q)$ where P and Q are state constraints as in 1, and α is an event of some sensor. Eg:

   ```
   swon & sw = Off & s2 = Off  =>  AX(bm = On)
   ```

   "if the switch is switched on and the end of the belt is clear, then in the next state the motor is on."

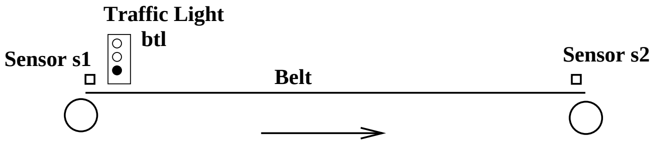

Fig. 4. Feed Belt of Simple Production Cell

3. Temporal invariants $P \Rightarrow AF(Q)$ or $P \Rightarrow AG(Q)$. AF is "in some future state" and AG is "in all future states". Eg:

   ```
   s2 = Off & bm = On  =>  AF(s2 = On)
   ```

   "If the belt end is clear and the belt motor on, then eventually the belt end will become occupied."

A constraint φ on a class C has the semantic interpretation

$$AG(\forall\, x : \overline{C} \cdot \varphi(x))$$

Attributes *att* of C are parameterised by x in $\varphi(x)$. So the first static invariant example above has the semantics

$$AG(\forall\, x : \overline{C} \cdot att(x) \leq 10)$$

A constraint on an association r between classes D_1 to D_k is interpreted by

$$AG(\forall\, x_1 : \overline{D_1};\ \ldots;\ x_k : \overline{D_k} \cdot (x_1, \ldots, x_k) \in \overline{r} \ \Rightarrow\ \varphi(x_1, \ldots, x_k))$$

where each attribute *att* of D_i occurring in φ is replaced by $att(x_i)$ in $\varphi(x_1, \ldots, x_k)$.

For example the second static invariant above is interpreted as:

$$\begin{aligned} AG(\forall\, x : \overline{Sw};\ y : \overline{S2};\ z : \overline{Bm} \cdot \\ (x, y, z) \in \overline{r} \ \Rightarrow \\ (sw(x) = On \wedge s2(y) = Off \ \Rightarrow\ bm(z) = On)) \end{aligned}$$

Internal consistency of a class diagram requires that the invariants are consistent with each other. This is checked by the RSDS tools [1].

4 Object Diagrams

An object diagram defines a set of objects of classes that occur in the class diagram, and a set of links between the objects, which are instances of the associations in the class diagram. It may also define specific values for attributes of the objects.

For an object diagram OD we define a 1st order CTL theory Ω_{OD}. This consists of:

1. Type symbols $@C$ for each class C and for the primitive types;

2. Attributes $\overline{C} : \mathbb{F}(@C)$ for each class C, and $\overline{r} : \mathbb{F}(@D_1 \times \ldots \times @D_k)$ for each association r, with end points at classes D_1 to D_k;
3. Attributes

$$att : @C \to T$$

for each attribute $att : T$ of class C;
4. Axioms

$$AG(\overline{C} \;=\; \{x_1, \ldots, x_p\})$$

where the x_i are all objects in the diagram declared to be of class C or of a subclass of C;
5. Axioms

$$AG(\overline{r} \;=\; \{t_1, \ldots, t_q\})$$

where the t_j are all tuples of objects corresponding to the links in the diagram declared to be instances of r;
6. Axioms $AG(att(obj) = val)$ for each attribute equation $att = val$ listed in object obj.

The key consistency requirement between an object diagram OD and its corresponding class diagram CD is that, for the model elements of CD that appear in OD, each of the static data axioms 5, 6, 7 of Δ_{CD} for these elements must be valid in Ω_{OD}, and that each of the static invariants that involve only these elements in CD must be valid in Ω_{OD}.

5 Statecharts

A component statemachine in RSDS is a tuple $A = (init_A, States_A, Trans_A, Events_A)$ where $init_A \in States_A$ is the initial state, $States_A$ is the set of states of A, $Events_A$ the set of events of A, and $Trans_A$ is the set of transitions of A. There are functions

$$source_A : Trans_A \to States_A$$
$$target_A : Trans_A \to States_A$$
$$event_A : Trans_A \to Events_A$$

which give the source, target and triggering event of each transition – these correspond to the meta-associations *source*, *target* and *trigger* in Figure 3. In addition

$$guard_A : Trans_A \to BooleanExpression$$

gives the guard of the transition (default expression *true*).

A controller statemachine C has the same structure but additionally has a set of receivers:

$$receivers_{Sys}(C) \subseteq Conts \cup Acts$$

where $Conts$ is the set of controller statemachines and $Acts$ the actuator statemachines, and it has generations on its transitions:

$$generations_C : Trans_C \rightarrow$$
$$(@CC \rightarrow \mathrm{seq}(Events_{R_1} \times @CR_1 \cup \ldots \cup Events_{R_p} \times @CR_p))$$

where $receivers_{Sys}(C) = \{R_1, \ldots, R_p\}$ and the CR_i are the classes corresponding to the R_i statemachines, CC is the class corresponding to C. The generations correspond to *effect*; *action* in the UML metamodel. No event of any sensor component can occur in any generation.

Each RSDS statechart is associated to a class in the system class diagram (by the *context* association of Figure 3), and describes the modes that objects of this class may pass through in their life history. The receiver statemachines of Sm must correspond to (actuator or controller) classes which are connected to the class of Sm by associations in the class diagram.

The action invariants

$$\alpha \;\&\; P \;\Rightarrow\; AX(Q)$$

of the class diagram must be consistent with the statemachines, in other words, if the condition P is true when event α occurs, then Q must be true in the state resulting from the completed execution (including execution of all generated events) of any transition triggered by α. Likewise, the static invariants and temporal invariants should be valid in the initial state of the system (consisting of the initial states of every statemachine) and in each subsequent *stable state* (states where there is no processing of an event in progress).

There are two distinct RSDS semantics for statecharts:

1. "Coarse grain" semantics describes "computations" of the system as sequences of steps, each step consists of a single sensor event and a finite number of actuator events, comprising the externally visible reaction to this sensor event. The steps lead from one stable state to the next.
2. "Fine grain" semantics steps consist of a sensor event followed by the sequence of actuator events which are the response to the sensor event – ie, ordering of actuator commands is defined within a reaction cycle, and intermediate states between stable states.

5.1 Coarse Grain Semantics

The *external events* of an RSDS system Sys are the sensor events:

$$Ext_{Sys} \;=\; \bigcup_{i:1..n} Events_{S_i}$$

where the set of sensors of *Sys* is $Sen = \{S_1, \ldots, S_n\}$. All other events of *Sys* are *internal*.

For each component statemachine *Sm* for class C a theory Γ_{Sm} is defined. This has a single attribute

$$sm : @C \rightarrow States_{Sm}$$

and it has action symbols

$$act : @C \rightarrow$$

for each element *act* of $Trans_{Sm} \cup Events_{Sm}$.

The axioms of Γ_{Sm} include:

1. The state-transition behaviour of *Sm*, for all $x : @C$:

$$\begin{array}{l} AG(sm(x) = s \;\wedge\; G(x) \;\wedge\; \alpha(x) \;\Rightarrow\; tr(x)) \\ AG(tr(x) \;\Rightarrow\; \alpha(x)) \\ AG(sm(x) = s \;\wedge\; tr(x) \;\Rightarrow\; AX(sm(x) = t)) \end{array}$$

   for each transition *tr* of *Sm* with source s, target t, guard G, and trigger event α.
2. That at most one transition of *Sm* can occur in a step:

$$AG(\forall\, x, y : \overline{C} \cdot \neg\, (tr(x) \wedge tr'(y)))$$

   for each pair of distinct transitions of *Sm*. Similarly for events.
3. That a transition can only occur if *Sm* is in its source state:

$$AG(tr(x) \;\Rightarrow\; sm(x) = s)$$

   where $s = source_{Sm}(tr)$.
4. Locality:

$$\begin{array}{l} AG(\forall\, v : State_{Sm} \cdot sm(x) = v \;\wedge \\ \qquad\qquad \neg\, tr_1(x) \;\wedge\; \ldots \;\wedge\; \neg\, tr_h(x) \;\Rightarrow\; AX(sm(x) = v)) \end{array}$$

   where the tr_i are all the transitions of *Sm*.

Notice that events may happen in a state from which there is no transition for the event. The state remains unchanged in this case.

At the system level, we define the theory Th_{Sys} of the complete RSDS specification *Sys* as the union of the class diagram theory and each of the statechart theories, together with the following global axiom:

1. For a controller transition *tr* with generations ρ:

$$AG(tr(x) \;\Rightarrow\; \rho(1)(y_1) \wedge \ldots \wedge \rho(p)(y_p))$$

   where p is $size(generations(tr)(x))$, $\rho(i)$ is the action symbol corresponding to event $first((generations(tr)(x))(i))$, and $y_i = second((generations(tr)(x))(i))$.

The consistency rules for a statemachine Sm are that its *context* must be a class C in the class diagram of the system, that its events must correspond to the operations of C, that the generated events from its transitions must correspond to invocations of operations of classes D which C is connected to by an association in the class diagram, and that any state within Sm which has a *context* must have that this context is a subclass of C (and substates of a state have contexts which are subclasses of its state, if both have contexts).

In terms of the metamodels of Figures 1, 2 and 3, these constraints are:

1. $Sm.context \in Class$
2. For each transition t of Sm:

$$\begin{array}{l} t.trigger \in CallEvent \\ t.trigger.operation \in C.feature \end{array}$$

3. For each transition t of Sm: $t.generation \in ActionSequence$ and

$$\forall\, i : 1\,..\, size(t.generation.action) \cdot t.generation.action(i) \in CallAction$$

and

$$\begin{array}{l} \forall\, i : 1\,..\, size(t.generation.action) \cdot \\ \qquad\qquad \exists\, D : C.allOppositeAssociationEnds.type \cdot \\ \qquad\qquad\qquad\qquad t.generation.action(i).operation \in D.feature \end{array}$$

where C is the class of the statemachine of t.

4. For states s and $s1$ of Sm:

$$\begin{array}{l} s1.container = s \;\wedge \\ s.stateMachine.context \neq null \;\wedge\; s1.stateMachine.context \neq null \;\Rightarrow \\ \qquad\qquad \exists\, g : Generalisation \cdot \\ \qquad\qquad\qquad\qquad s1.stateMachine.context = g.child \;\wedge \\ \qquad\qquad\qquad\qquad s.stateMachine.context = g.parent \end{array}$$

These constraints can also be expressed in OCL syntax, for example constraint 3 is:

$$\begin{array}{l} t.generation.oclIsTypeOf(ActionSequence) \\ forAll(i : 1\,..\, t.generation.action \rightarrow size \mid \\ \qquad\quad t.generation.action \rightarrow at(i).oclIsKindOf(CallAction)) \\ forAll(i : 1\,..\, t.generation.action \rightarrow size \mid \\ \qquad\quad exists(D : C.allOppositeAssociationEnds.type \;\mid \\ \qquad\qquad\quad D.feature \;\rightarrow \\ \qquad\qquad\qquad\quad includes(t.generation.action \rightarrow at(i).operation))) \end{array}$$

Internal consistency of a statemachine in the coarse grain semantics requires that the generations of a transition do not include conflicting actions, ie, events $\beta(x)$ and $\gamma(x)$ on the same object x which trigger different transitions of x's statemachine with the same source state but with different targets.

5.2 Fine Grain Semantics

The fine-grain semantics of Sys consisting of sensors $S_1, \ldots, S_n$, controllers $C_1, \ldots, C_m$ and actuators $A_1, \ldots, A_p$ can be expressed in terms of statemachines of the components together with axioms on an attribute

$$event\_queue : \mathrm{seq}(EventToken)$$

of Th_{Sys} which holds a sequence of tokens representing pending events.

$EventToken$ is isomorphic to the set $Events_{Sys}$ of all events of the system:

$$\begin{array}{rcl} Events_{Sys} & = & \bigcup_{i:1..n} Events_{S_i} \times @X_i \ \cup \ \bigcup_{i:1..m} Events_{C_i} \times @Y_i \\ & & \cup \ \bigcup_{i:1..p} Events_{A_i} \times @Z_i \end{array}$$

where X_i is the class corresponding to S_i, etc. Let $name : Events_{Sys} \to EventToken$ be this semantic isomorphism.

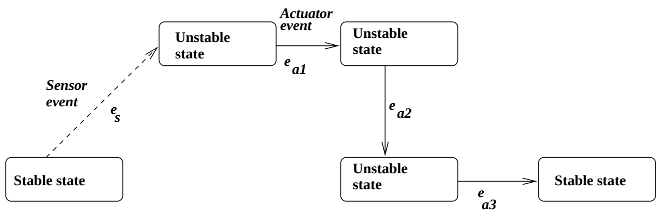

Fig. 5. Fine-grain Semantics

The global axiom 1 of the coarse-grain semantics is replaced by the following axioms on the event queue:

The event queue is initially empty:

$$event\_queue = []$$

An external event α can only be responded to if the queue is empty:

$$AG(t(x) \ \Rightarrow \ event\_queue = [])$$

for each transition t with trigger event some $\alpha \in Ext_{Sys}$.

When an external event is accepted by the system, generations of the triggered transition become the new event queue – all these events should be internal:

$$AG(tr(x) \ \Rightarrow \ AX(event\_queue = \rho))$$

where $tr(x)$ is the transition triggered by $\alpha(x)$, and

$$\rho_1 = generations(tr)(x)$$

and $\rho(i) = name(\rho_1(i))$ for each i. If several transitions in different components are triggered by $\alpha(x)$ then some interleaving of their generations becomes the new event queue.

An internal event can only be processed if it is at the head of the queue:

$$AG(t(x) \;\Rightarrow\; event\_queue \neq [] \;\wedge\; name(\beta(x)) = event\_queue(1))$$

for each controller or actuator transition $t(x)$ with trigger $\beta(x)$.

The head of the queue is removed when the corresponding event is processed, and replaced with some interleaving ρ of generations of all transitions triggered by this occurrence of β:

$$\forall\, q \cdot AG(t(x) \wedge q = event\_queue \;\Rightarrow\; AX(event\_queue = \rho \frown tail(q)))$$

This models the "run to completion" semantics of statecharts in UML [7].

A *stable state* is a configuration of *Sys* where $event\_queue = []$ (Figure 5), these correspond to the states of the coarse-grain models. Figure 6 shows an example system, where there is one object of each class. The dashed arrows indicate the *receivers* relation. The execution history of the event queue in this

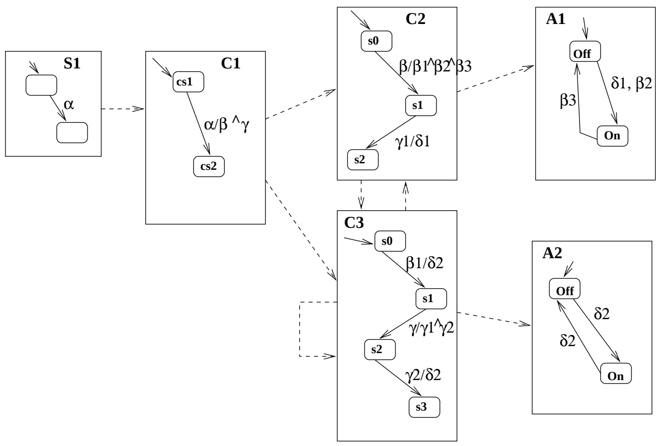

Fig. 6. Example of Fine-Grain Semantics

case is:

$[]$
$[\beta, \gamma]$
$[\beta_1, \beta_2, \beta_3, \gamma]$
$[\delta_2, \beta_2, \beta_3, \gamma]$
$[\beta_2, \beta_3, \gamma]$
$[\beta_3, \gamma]$
$[\gamma]$
$[\gamma_1, \gamma_2]$
$[\delta_1, \gamma_2]$
$[\gamma_2]$
$[\delta_2]$
$[]$

Deferral of events can also be modelled in this semantics: if $event\_queue(1)$ is deferred in the current state, then apply the above rules to the modified queue

$$[rest(1)] \frown def \frown tail(rest)$$

where $event\_queue = def \frown rest$ and def is the maximal initial subsequence of events which are all deferred in the current state. A consistency rule would be that there must be at least one non-deferred event in a non-empty event queue.

5.3 Extension to Full UML Statecharts

In general UML state models, generated actions may either be synchronous or asynchronous. The former must be executed to completion before their caller can progress to generate the next action, whilst the latter immediately release the caller to progress to the next action.

To model this, we add an attribute

$$event\_set : \mathbb{F}(EventToken)$$

which holds the set of asynchronous events generated but not yet consumed. These are the *CallAction* instances with *isAsynchronous* being *true*, in the generated actions of transitions.

For example, if a transition for event α generates events $\beta;\ \gamma$ and β triggers a transition with generations $\beta_1;\ \beta_2$ with β_2 asynchronous, and γ triggers a transition with generations $\gamma_1;\ \gamma_2$, then the change of the event queue and event set with each step after α is accepted could be as follows:

event_queue	*event_set*
$[\beta, \gamma]$	$\{\}$
$[\beta_1, \beta_2, \gamma]$	$\{\}$
$[\gamma]$	$\{\beta_2\}$
$[\gamma_1, \gamma_2]$	$\{\beta_2\}$
$[\gamma_2]$	$\{\beta_2\}$
$[\gamma_2]$	$\{\}$
$[]$	$\{\}$

A system can select either the event at the head of the event queue, or any event in the event set to respond to at any step:

$$AG(tr(x) \;\Rightarrow\; name(\alpha(x)) \in event\_set \;\vee \\ (event\_queue \neq [] \wedge name(\alpha(x)) = event\_queue(1)))$$

where $tr(x)$ is triggered by $\alpha(x)$.

The first element in *event_queue* must always be a synchronous event:

$$AG(tr(x) \wedge event\_queue \neq [] \wedge name(\alpha(x)) = event\_queue(1) \;\Rightarrow \\ AX(event\_queue = q' \wedge event\_set = s'))$$

where $tr(x)$ is triggered by $\alpha(x)$ in the current step, q is the value of the event queue at start of the step, s is the value of the event set at start of the step, ρ is the sequence of tokens of events generated by $tr(x)$. $s' = s \cup \mathrm{ran}(a)$ and $\rho \frown tail(q) \;=\; a \frown q'$ where a is the maximal sequence of asynchronous events at the front of $\rho \frown tail(q)$.

Likewise, if an event from the event set is selected:

$$AG(tr(x) \wedge name(\alpha(x)) \in event\_set \;\Rightarrow \\ AX(event\_queue = q' \wedge event\_set = s'))$$

$s' = s - \{name(\alpha(x))\} \cup \mathrm{ran}(a)$ and $\rho \frown q \;=\; a \frown q'$ where a is the maximal sequence of asynchronous events at the front of $\rho \frown q$.

6 Verification of RSDS Models

RSDS models may be verified by translation to B AMN and use of the B Toolkit [4] to check that required invariants are maintained by the implemented method algorithms. However temporal properties cannot be checked using B, for such properties we can use model checkers such as SMV.

Translations have been defined from the coarse and fine-grain models of RSDS into SMV to support such verification [5]. For the fine-grain semantics, an important semantic property is that each reaction to an external event terminates:

$$AG(event\_queue \neq [] \;\Rightarrow\; AF(event\_queue = []))$$

This can be checked in the SMV translation of each particular system.

7 Case Study

An example of RSDS specification is a simple internet based system which consists of a database on a server, containing a set of entries, records with two string fields, *name* and *data*. Via a web browser a user may add such records to the database, delete records, and list all records in the database (Figure 7). The constraints between the interface (sensor) classes representing the add and

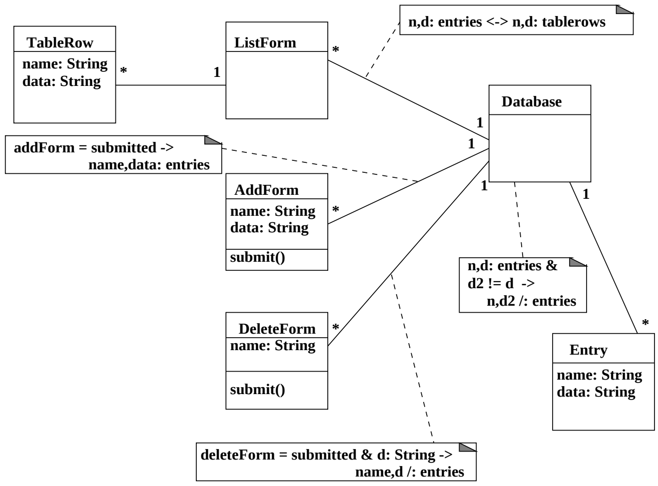

Fig. 7. Class Diagram of Internet System

delete HTML forms and the database represent the requirements that as soon as the commit button on each instance of these forms is pressed, the corresponding entry is added or removed from the database. Each form has a simple two-state statemachine (Figure 8). The other component state models are trivial (single state statemachines). In the implementation of this system, suitable actions are needed in the generations of the interface statemachines (or in suitable controller statemachines) to ensure the preservation of the invariants. For example the *submit* transition in the delete form statemachine must remove using *unlink* and *kill* all entries with the name in the form.

Summary

We have shown that a useful subset of UML notation can be formally defined in a coherent manner, so that statechart and class diagram models have semantic representations in the same logical language. Consistency rules within and between models have been defined. A large number of case studies using RSDS have been carried out and tool support for RSDS is under development.

References

1. K. Androutsopoulos, *The RSDS Tool*, Department of Computer Science, King's College, 2001. `http://www.dcs.kcl.ac.uk/pg/kelly/`
2. B. Berard et al, *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer-Verlag, 1999.
3. J. Bicarregui, K. Lano, T. Maibaum. *Objects, Associations and Subsystems: A hierarchical approach to encapsulation*, ECOOP 97, LNCS, 1997.

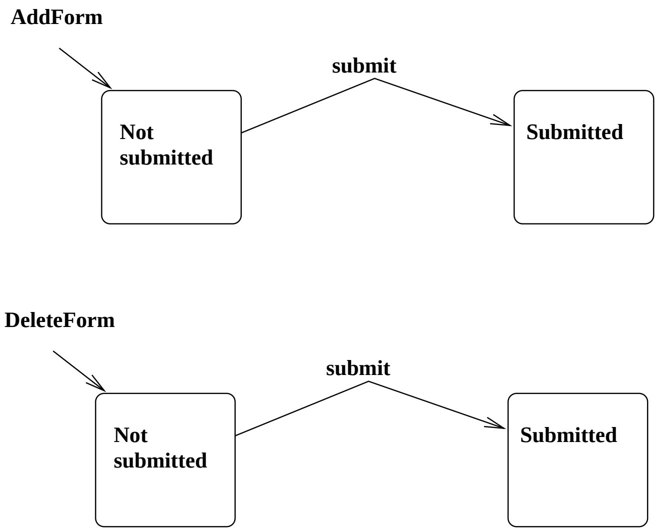

Fig. 8. State Diagrams of Internet System

4. The B Toolkit, B Core UK Ltd, http://www.b-core.co.uk/.
5. K. Lano, D. Clark, K. Androutsopoulos, P. Kan, *Invariant-based Synthesis of Fault-tolerant Systems*, FTRTFT 2000.
6. OMG unified modelling language specification 1.4. OMG, http://www.omg.org/technology/documents/formal/uml.htm.
7. UML Semantics, at `http://www.rational.com/uml`.