Semantics and Transformations for UML Models

K. Lano and J. Bicarregui

Dept. of Computing, Imperial College
180 Queens Gate, London SW7 2BZ
kcl@doc.ic.ac.uk

K. Lano J. Bicarregui

Abstract. This paper presents a semantic framework for a large part of UML, and gives a set of transformations on UML models based on this semantics. These transformations can be used to enhance, rationalise, refine or abstract UML models.

1 Introduction

A semantically-based transformation calculus for UML [19] and related OO notations is useful in a number of ways:

1. it provides a set of correct transformations which are equivalences or enhancements of models, and can be used to support forward or reverse engineering [12];
2. the transformations clarify the meaning of the modelling notations, without the developer needing to manipulate the mathematical formalisms underpinning the transformations.

A more rigorous development approach is essential for applications in critical areas, such as medical database and robotic systems [16], defence [17] and chemical process control [13].

Although our semantic model is not a complete semantics for UML, it provides a sufficient basis to justify transformations which are expected to be model enhancements or refinements. It is a step towards a full semantics.

The transformational approach is consistent with the presentation of UML in [19] (which includes, for example, equivalences on notations for composition aggregation), and lends itself to CASE tool support. The transformations could themselves be expressed in UML as refinements (typically with subdependencies) in which the new model is the client and the old model the supplier.

In this paper we present extracts from the proposed semantic framework and show how it can be used to justify some example transformations on the main modelling notations of UML.

2 Basic Semantic Elements

A mathematical semantic representation of UML models can be given in terms of theories in a suitable logic, as in the semantics presented for Syntropy in [3] and VDM^{++} in [15]. In order to reason about real-time specifications we will use the more general version of this formal framework, termed Real-time Action Logic (RAL), presented in [15].

A RAL theory has the form:

theory *Name*
types *local type symbols*
attributes *time-varying data, representing instance or class variables*
actions *actions which may affect the data, such as operations, statechart transitions and methods*
axioms *logical properties and constraints between the theory elements.*

The logical notation which can be used in theories is first order predicate logic using Z notations such as $\mathbb{F}(T)$, the set of finite subsets of T, together with temporal operators $\bigcirc$ (next), $\Box$ (henceforth), $\Diamond$ (eventually). There are also terms $\leftarrow(\alpha, i)$, $\rightarrow(\alpha, i)$, $\uparrow(\alpha, i)$ and $\downarrow(\alpha, i)$ denoting the request send, request arrival, initiation and termination times respectively of an action invocation (α, i) for action α and $i : \mathbb{N}_1$.

Theories can be used to represent classes, instances, associations and general submodels of a UML model.

2.1 Example Semantic Representation

An example UML class diagram is shown in Figure 1. The corresponding theory

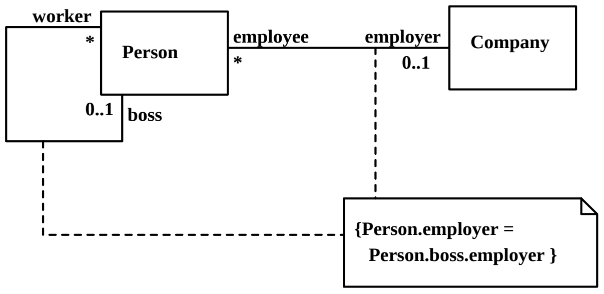

Fig. 1. UML Class Diagram

is:

theory *Employment*
types *Person*, *Company*
attributes
$\overline{Person} : \mathbb{F}\, Person$
$\overline{Company} : \mathbb{F}\, Company$
$employee\_employer : Person \leftrightarrow Company$

$employee : Company \rightarrow \mathbb{F}(Person)$
$employer : Person \rightarrow \mathbb{F}(Company)$
$\overline{worker\_boss} : Person \leftrightarrow Person$
$worker : Person \rightarrow \mathbb{F}(Person)$
$boss : Person \rightarrow \mathbb{F}(Person)$

$\overline{Person}$ represents the finite set of existing objects of class $Person$ – the extension $ext(Person)$ of $Person$ in the terms of [18]. Instance variables of class C are modelled as attributes of a function type $C \rightarrow T$. Associations between classes are modelled as relations between their types.

actions *Standard predefined actions to modify classes and associations:*
$create_{Person}(p : Person) \quad \{\overline{Person}\}$
$kill_{Person}(p : Person) \quad \{\overline{Person}\}$
$create_{Company}(c : Company) \quad \{\overline{Company}\}$
$kill_{Company}(c : Company) \quad \{\overline{Company}\}$
$add\_link_{employee\_employer}(p : Person, c : Company)$
$\quad \{\overline{employee\_employer}, employer, employee\}$
$delete\_link_{employee\_employer}(p : Person, c : Company)$
$\quad \{\overline{employee\_employer}, employer, employee\}$
$add\_link_{worker\_boss}(p : Person, q : Person) \quad \{\overline{worker\_boss}, worker, boss\}$
$delete\_link_{worker\_boss}(p : Person, q : Person) \quad \{\overline{worker\_boss}, worker, boss\}$

We present the write frame of each action as a set after the action declaration. This is the set of attributes which it may change. Query operations in the sense of UML are therefore represented by actions with an empty write frame.

axioms
The association links only existing persons and companies:

$$\overline{employee\_employer} \in \overline{Person} \leftrightarrow \overline{Company}$$

The two directions of the association are derived from the set of pairs in its relation:

$$\begin{aligned} &\forall p : \overline{Person};\ c : \overline{Company} \cdot \\ &\quad c \in employer(p) \equiv (p, c) \in \overline{employee\_employer} \ \wedge \\ &\quad p \in employee(c) \equiv (p, c) \in \overline{employee\_employer} \end{aligned}$$

Cardinality constraints:

$$\begin{aligned} &\forall p : \overline{Person} \cdot card(employer(p)) \leq 1 \\ &\forall p : \overline{Person} \cdot card(boss(p)) \leq 1 \end{aligned}$$

There are similar axioms for $worker\_boss$. The constraint of the model is expressed by the formula:

$$\forall p : \overline{Person} \cdot \ employer(p) = employer(\!| boss(p) |\!)$$

$f(\!| X |\!)$ denotes the set of values $f(x)$ for $x \in X$. OCL notation could be used for the axioms, but would be more prolix in general.

Theories can be linked by *theory morphisms* [9,7], which enable the theory of a complete model to be assembled from theories of submodels and eventually from the theories of specific elements, classes, states, associations, etc.

Generalisation of class C by class D in UML is directly represented by the theory $T(D)$ of D being the source of a signature morphism into $T(C)$ which is the identity (each symbol of $T(D)$ is interpreted by itself in $T(C)$). Dashed generalisation of C by D is directly represented by an interface morphism (a signature morphism which only maps action symbols of the first theory to action symbols of the second theory) from $T(D)$ to $T(C)$ which is the identity on the action symbols of D and their signature types.

A theory morphism is a signature morphism s from $T1$ to $T2$ which preserves all the axioms of the source theory. That is, $T2$ proves $s(P)$ for each axiom P of $T1$. The simplest form of theory morphism is the inclusion of one theory (all its symbols and axioms) in another. This is denoted by writing **includes** $T1$ after the header of theory $T2$.

Using this we can re-express theory *Employment* above as:

theory *Employment*
includes *WorkerBoss*, *EmployeeEmployer*
axioms $\forall\, p : \overline{Person} \cdot\ employer(p) \ = \ employer(\!|\, boss(p)\, |\!)$

where *WorkerBoss*, etc are theories of the associations which themselves *include* the theories of *Person* and *Company* (Figure 1).

3 Static Structure Diagrams

A UML class C is semantically represented by a theory $T(C)$ of the form:

theory $T(C)$
types C
attributes
  $\overline{C} : \mathbb{F}(C)$
  $self : C \to C$
  $att_1 : C \to T_1$
  ...
actions
  $create_C(c : C) \quad \{\overline{C}\}$
  $kill_C(c : C) \quad \{\overline{C}\}$
  $op_1(c : C, x : X_1) : Y_1$
  ...
axioms

$$\forall\, c : C \cdot \\ self(c) = c \ \wedge \ [create_C(c)](c \in \overline{C}) \ \wedge \ [kill_C(c)](c \notin \overline{C})$$

The notation $[action]P$ denotes that every execution of *action* terminates with the predicate P being true. Thus $create_C(c)$ always adds c to the set of existing C objects, and $kill_C(c)$ removes it.

Each instance attribute $att_i : T_i$ of C gains an additional parameter of type C in the class theory $T(C)$ and similarly for operations. The class theory can be generated from a theory of a typical C instance by means of an A-morphism

[3]. Class attributes and actions do not gain the additional C parameter as they are independent of any particular instance. We denote $att(a)$ for attribute att of instance a by the standard OO notation $a.att$, and similarly denote actions $act(a, x)$ by $a!act(x)$.

We will refer to the conjunction of all the properties of the attributes of C as the invariant Inv_C of the class. We include the axiom $\forall\, a : \overline{C} \cdot a.Inv_C$ in $T(C)$ to express this, where $a.P$ is P with a added as the first parameter of all instance attributes and actions of C in P.

Similarly each association lr can be interpreted by a theory which contains an attribute $\overline{lr}$ representing the current extent of the association (the set of pairs in it) and actions *add_link* and *delete_link* to add and remove pairs (links) from this. Axioms define the cardinality of the association ends and other properties of the association.

If D inherits from C then $T(D)$ is constructed by *include*ing $T(C)$, adding symbols and axioms for the new features of D, and adjoining the axioms $D \subseteq C \wedge \overline{D} \subseteq \overline{C}$ which ensure that attributes and operations of C can be applied to instances of D.

If class C has subclasses $S_1, \ldots, S_n$, we can assert that objects cannot migrate from one subclass to another by axioms:

$$\forall\, x : \overline{S_i} \cdot x \notin \overline{S_j} \;\Rightarrow\; \Box(x \notin \overline{S_j})$$

for $j \neq i$. However, if S_i and S_j arise as states in a statechart, then such subtype migration is permitted.

That two subclasses S_1 and S_2 are disjoint is expressed by axioms $S_1 \cap S_2 = \varnothing$ in a theory which contains both class theories. If a class C is abstract with a complete set of subclasses $S_1, \ldots, S_n$ then we can assert that $\overline{C} = \overline{S_1} \cup \ldots \cup \overline{S_n}$ in a theory containing all of these class theories. A complete set of subclasses for C prevents the application of any transformation to introduce new direct subclasses of C.

Likewise, if a class is asserted to be a *leaf*, then no transformation can introduce subclasses of this class, and no superclasses can be introduced for a *root* class.

3.1 Rationalising Inheritance Hierarchies

If two classes A and B are both subclasses of another class D, then it is valid to introduce a subclass C of D which acts as an abstract superclass of both A and B (Figure 2). This transformation is valid because $\overline{A} \subseteq \overline{D} \wedge \overline{B} \subseteq \overline{D}$ imply that $\overline{C} = \overline{A} \cup \overline{B}$ is a subset of $\overline{D}$.

3.2 Rationalising Disjoint Associations

The following transformation (Figure 3) can be applied to object models to eliminate some cases of optional association ends. This transformation is logically

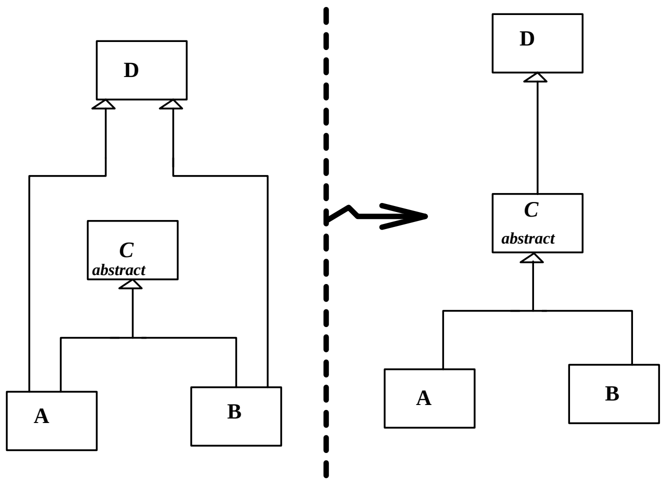

Fig. 2. Minimal Superclass Transformation

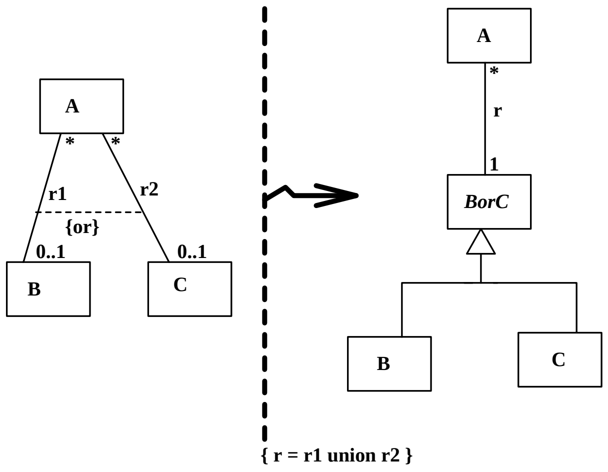

Fig. 3. Rationalising Disjoint Associations

valid as r_1 and r_2 are disjoint and function-like by definition of the "or" constraint [19]:

$$\begin{aligned}&\forall\, a \in \overline{A} \;\cdot\; (\exists\, b \in \overline{B} \;\cdot\; (a, b) \in \overline{r_1}) \;\vee\; (\exists\, c \in \overline{C} \;\cdot\; (a, c) \in \overline{r_2}) \;\;\wedge\\ &\forall\, a \in \overline{A} \;\cdot\; \neg\,((\exists\, b \in \overline{B} \;\cdot\; (a, b) \in \overline{r_1}) \;\wedge\; (\exists\, c \in \overline{C} \;\cdot\; (a, c) \in \overline{r_2}))\end{aligned}$$

and B and C are disjoint.

Thus the abstract generalisation class $BorC$ which has $\overline{BorC} = \overline{B} \cup \overline{C}$ can be constructed, and $\overline{r} = \overline{r_1} \cup \overline{r_2}$ has the specified cardinality at the $BorC$ end. A similar transformation works for any cardinality combination at the A end: the resulting association has cardinality the generalisation of the separate r_1 and r_2 cardinalities at this end.

3.3 Refining Class Invariants

Logically strengthening a class invariant is a refinement transformation. If class C has invariant Inv_C, then adding extra constraints or restating Inv_C in a logically stronger manner to produce a predicate Inv'_C yields a refined class. The theory interpretation is the identity.

3.4 Transitivity of Composition Aggregation

One proposed meaning [5] of composition aggregation of B instances into A via an association ab is that the B instances are *frozen* in their relationship with a particular A instance: the inverse image $\overline{ab}^{-1}(\!|\ \{b\}\ |\!)$ is constant for each $b : \overline{B}$ for the duration of its membership in ab.

If ab is a one-many association this means that b cannot move from one container to another $(*)$:

$$\forall\, a : A;\; b : B \cdot\; (a, b) \in \overline{ab} \wedge \diamond((a', b) \in \overline{ab}) \;\Rightarrow\; a = a'$$

$\diamond P$ denotes that P holds at the current or some future time.

The relational composition of two one-many composition aggregations is then itself a composition aggregation because:

$$\begin{aligned}&(a, c) \in \overline{ab};\ \overline{bc} \;\Rightarrow\\ &\qquad\qquad \exists\, b : \overline{B} \cdot (a, b) \in \overline{ab} \wedge (b, c) \in \overline{bc}\end{aligned}$$

$((1) \Rightarrow (2))$ and

$$\begin{aligned}&\diamond((a', c) \in \overline{ab};\ \overline{bc}) \;\Rightarrow\\ &\qquad\qquad \diamond(\exists\, b' : \overline{B} \cdot (a', b') \in \overline{ab} \wedge (b', c) \in \overline{bc})\end{aligned}$$

$((3) \Rightarrow (4))$. But then $(1) \wedge (3)$ implies $(2) \wedge (4)$, so by $(*)$ applied to b, c we have $b' = b$. Therefore, applying $(*)$ to a, b we have $a' = a$ as required.

3.5 Deduction Transformations

If we know that a diagram M_1 ensures that the properties of an enhanced diagram M_2 also hold, then we say that M_2 can be deduced from M_1: $M_1 \vdash M_2$. This is just the same as asserting that there is a refinement transformation from M_2 to M_1.

A particular example is that the composition of 'selector' associations remains a selector of the composed association. In other words, if we know that $\overline{r1} \subseteq \overline{R1}$, $\overline{r2} \subseteq \overline{R2}$, then also the composition $\overline{r1}$; $\overline{r2}$ is a subset of $\overline{R1}$; $\overline{R2}$.

4 Sequence and Collaboration Diagrams

A sequence diagram defines constraints on the timing of method requests, activations and terminations. For example, a timing mark a at the source point of a message m sent from object s to object t represents the time $a = \leftarrow(t!m, i)$ of some request send of m. If this arrow is horizontal this is also the time $a' = \rightarrow(t!m, i)$ of arrival of this request at t.

A timing mark at the destination of a signal arrow represents a request arrival time $\rightarrow(t!m, i)$, or the termination time $\downarrow(t!m, i)$ of an invocation in the case that the arrow represents the return of a procedural call $t!m$ (ie, the arrow is dashed with source t).

For example, Figure 4 translates to the following assertions, where each message execution lifeline is interpreted by a particular message instance:

$$\begin{aligned}
&\forall\, i : \mathbb{N}_1 \cdot \exists\, j, k, l, l' : \mathbb{N}_1 \cdot \\
&\qquad \rightarrow(Op, i) \;=\; \uparrow(create_{C1}(ob1), l) \\
&\qquad \downarrow(create_{C1}(ob1), l) \;\le\; \leftarrow(ob3!bar(x), j) \;=\; \rightarrow(ob3!bar(x), j) \\
&\qquad\qquad\qquad \le\; \leftarrow(ob4!do(w), k) \;=\; \rightarrow(ob4!do(w), k) \\
&\qquad \downarrow(ob4!do(w), k) \;\le\; \downarrow(ob3!bar(x), j) \\
&\qquad\qquad\qquad \le\; \downarrow(kill_{C1}(ob1), l') \;=\; \downarrow(Op, i)
\end{aligned}$$

These assertions can then be checked for consistency against detailed implementation level statecharts.

Replacing such constraints by logically stronger formulae (eg, reducing the range of possible time delays between a request arrival and a result signal) is therefore a refining transformation. It is also valid to introduce new objects and calls on these provided that the existing model elements are preserved.

The structural elements of a collaboration diagram simply represent particular instances of classes and their links, and so may be expressed in suitable extensions of class or submodel theories.

The interaction aspects can be modelled using composite actions [15] such as ; (sequential composition), := (assignment); || (concurrent composition); *for all* (iteration over a set); *if* (conditional execution); $\sqcap$ (binary choice of actions), *create* and *kill*, etc.

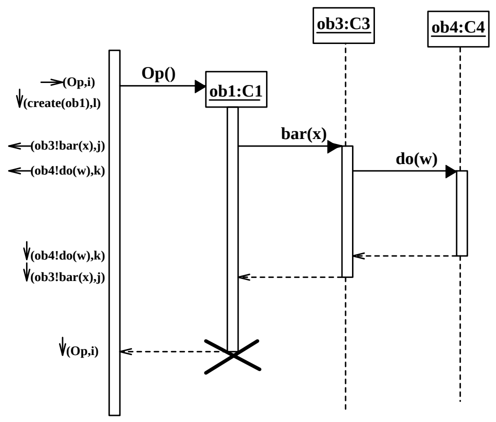

Fig. 4. Example Sequence Diagram with Annotations

5 Statecharts

A statechart specification of the behaviour of instances of a class C can be formalised as an extension of the class theory $T(C)$ of C, as follows. We use the relationship $\alpha \supset \beta$ "α calls β" for action symbols α and β to denote that every occurrence of α coincides with an occurrence of β:

$$\begin{aligned}
&\alpha \supset \beta \quad \equiv \\
&\quad \forall\, i : \mathbb{N}_1 \cdot \exists\, j : \mathbb{N}_1 \cdot \\
&\qquad \uparrow(\alpha, i) = \uparrow(\beta, j) \;\wedge\; \downarrow(\alpha, i) = \downarrow(\beta, j)
\end{aligned}$$

Then the extended theory of C has the additional axioms:

1. Each state S is represented in the same manner as a subclass of C, and in general, nesting of state S_1 in state S_2 is expressed by axioms $S_1 \subseteq S_2$ and $\overline{S_1} \subseteq \overline{S_2}$ as for class generalisation.
2. Each transition in the statechart and each event for which the statechart defines a response yields a distinct action symbol. The occurrence of an event e is equivalent to the occurrence of one of its transitions t_i (it is the abstract generalisation of the transition actions):

$$t_1 \supset e \;\wedge\; \ldots \;\wedge\; t_n \supset e$$

3. The axiom for the effect of a transition t from state S_1 to state S_2 with label $e(x)[G]/Post \frown Act$ where $Post$ is some postcondition constraint on the resulting state, is

$$\forall\, a : C \cdot a.G \wedge a \in \overline{S_1} \;\Rightarrow\; [a!t(x)](a.Post \wedge a \in \overline{S_2})$$

4. The transition only occurs if the trigger event occurs whilst the object is in the correct state:

$$\forall\, a : C \cdot a \in \overline{S_1} \wedge a.G \;\Rightarrow\; (a!e(x) \supset a!t(x))$$

5. The generated actions must occur at some future time (after t has occurred):

$$a!t(x) \;\Rightarrow\; \bigcirc \diamond\, a.Act$$

Transitions g with labels of the form $after(t)$ from source state S have an alternative axiom 4 defining their triggering which asserts that they are triggered t time units after the most recent entry time to state S [14].

Axiom 5 adopts the semantics given in Syntropy [5] for generated actions: the new state must be established before generated actions can be executed. In contrast to the statemate semantics of statecharts [10], these actions can be executed in steps other than the immediately following step. This appears to be the correct interpretation of asynchronously generated signals in UML [19]. Synchronously invoked actions have the alternative axiom

$$a!t(x) \;\supset\; a.Act$$

If state S is a concurrent composition of substates, we require that each occurrence of an event α results in an occurrence of one transition t_i for α in each distinct concurrent sub-region of S which has a transition for this event. For example, if there are transitions t_2 and t_3 for α in region 1, and transition t_1 in region 2 of a state S, then we have the axioms:

$$\begin{array}{l} a \in \overline{S} \;\Rightarrow\; (a!t_1 \supset a!t_2 \sqcap a!t_3) \\ a \in \overline{S} \;\Rightarrow\; (a!t_2 \supset a!t_1) \\ a \in \overline{S} \;\Rightarrow\; (a!t_3 \supset a!t_1) \end{array}$$

Thus changing the *isConcurrent* attribute of a composite state from *false* to *true* represents a theory extension and therefore a refinement.

Some typical transformations on statecharts are then as follows:

5.1 Source and Target Splitting

These transformations [5] can be shown to be valid for UML given the above semantics. Similarly, adding a nested state machine to a simple state S is generally a refinement provided that existing transitions from S are not overridden by transitions from substates of S which go to new destination states partly or fully disjoint from the original destinations.

5.2 Abstracting Events

In UML signal events can be arranged in a generalisation hierarchy. For example, an event $g(x)$ can be represented as a generalisation of events $h(x, y)$ and $f(x, z)$

on a class diagram (x, z are the *attributes* of event f, etc). The semantic meaning is that every occurrence of a specialised event is also an occurrence of every event it generalises (1):

$$h(x, y) \supset g(x) \qquad f(x, z) \supset g(x)$$

This means that transitions for h and f can be replaced by transitions for g, if g is an abstract generalisation of these two actions, since each axiom $a \in \overline{S} \wedge a.G \Rightarrow (a!g(x) \supset a!t(x))$ for a transition t of g yields the corresponding axiom for h or f.

This transformation is useful to reduce the number of events which a control system must respond to, eg, to replace separate events "switch on" and "switch off" by "toggle" [2].

5.3 Strengthening Transition Guards

The guard G of a transition from state S to state T may be strengthened by the invariant of S, since this invariant inevitably holds in the source state at points where the system is waiting for input events.

5.4 Eliminating Transitions

A transition t with a logically *false* guard can be eliminated, since it can never be taken. Its effect axiom has the form

$$a \in \overline{S} \wedge a.G \Rightarrow [t]Post$$

but this is trivially always true if $a.G \equiv \mathit{false}$.

Such transitions may arise as the result of source and target splitting, for example, in Figure 5, we target split the *Finished* state and transition *finish*, and then source split the *Filling* state and the two transitions for *finish*, yielding 6 separate transitions for *finish*. However, all but two of the resulting transitions are now impossible, so can be eliminated: the first transition for *finish*, with guard $level \geq min \wedge level < norm$ cannot occur from either the $F1$ or $F3$ states, and the second transition cannot occur from either the $F1$ or $F2$ states.

A similar step is carried out in the first refinement of Abrial's development of a distributed protocol [1].

Conclusions

This paper has illustrated the use of transformations on UML models as a means of rigorous development and re-engineering; based on a detailed semantics of these models. Real-time extensions of these models and corresponding transformations are currently under development. An international collaborative project on the UML semantics is underway to combine other related approaches, such as [6, 4] into a common framework. Tool support for transformations as part of a

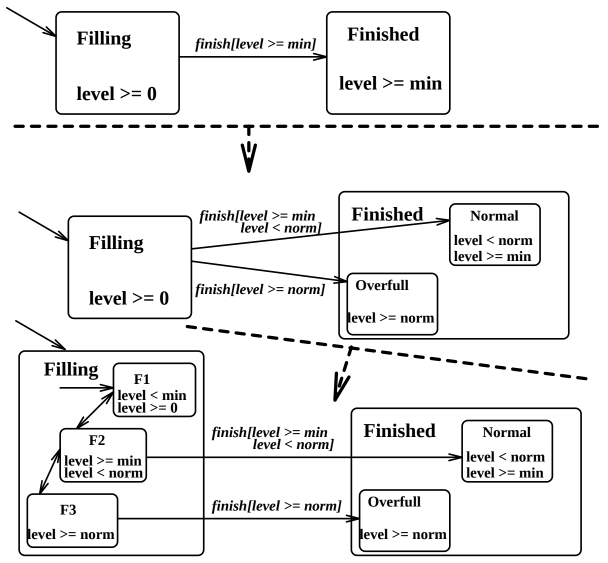

Fig. 5. Successive Splitting and Elimination Transformations

general CASE tool for UML will also be developed. A library of proved transformations will be provided, eliminating the need for developers to reason directly in RAL when applying transformations as development steps.

Suggestions for improvement of UML which have come from this work are:

1. Consider statechart states as classifiers, whose instances are those objects currently in the state. This unifies similar concepts in the same metamodel entity.
2. Attach constraints to packages or subsystems which enclose the submodel on which the constraint applies, in preference to attaching the constraint to a possibly large number of elements in this submodel.

References

1. J Abrial, L Mussat. *Specification and Design of a Transmission Protocol by Successive Refinements using B*, 1997.
2. M Awad, J Kuusela, and Jurgen Ziegler. *Object-oriented Technology for Real-time Systems.* Prentice Hall, 1996.
3. J C Bicarregui, K C Lano, T S E Maibaum, *Objects, Associations and Subsystems: a hierarchical approach to encapsulation*, ECOOP 97, LNCS, 1997.
4. R Breu, U Hinkel, C Hofmann, C Klein, B Paech, B Rumpe, V Thurner, *Towards a Formalization of the Unified Modeling Language*, ECOOP 97 proceedings, LNCS 1241, Springer-Verlag, 1997.
5. S Cook and J Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Sept 1994.

6. A Clark and A Evans, *Foundations of the Unified Modeling Language.* In D Duke and A Evans, editors, *BCS FACS – 2nd Northern Formal Methods Workshop*, Workshops in Computing, Springer Verlag, 1997.
7. J Fiadeiro and T Maibaum. *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing 4(3), pp. 239–272, 1992
8. R France, A Evans, K Lano, *The UML as a Formal Modelling Notation*, OOPSLA 97 Workshop on Object-Oriented Behavioral Semantics, 1997.
9. J Goguen and R Burstall. *Introducing Institutions.* In Clarke and Kozen, eds. Logics of Programs, pp. 221-256, Springer-Verlag, 1984.
10. D Harel and A Naamad, *The Statemate Semantics of Statecharts*, technical report, i-Logix, Inc, 1995.
11. K Lano, S Goldsack, J Bicarregui and S Kent. *Integrating VDM^{++} and Real-Time System Design*, Z User Meeting, 1997.
12. K. Lano, N. Malik, *Reengineering Legacy Applications using Design Patterns*, STEP '97, IEEE Computer Society Press, 1997.
13. K Lano, A Sanchez, *Design of Reactive Control Systems for Event-driven Operations*, FME '97, LNCS, Springer-Verlag, 1997.
14. K. Lano, *Transformations on Syntropy and UML Models*, Technical Report, "Formal Underpinnings for Object Technology" project, Dept. of Computing, Imperial College, 1997.
15. K Lano, *Logical Specification of Reactive and Real-Time Systems*, to appear in *Journal of Logic and Computation*, 1998.
16. N Leveson, *Safeware: system safety and computers*, Addison-Wesley, 1995. ISBN 0-201-11972-2.
17. Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment*, DEF-STAN 00-55, Issue 1, Part 2. Room 5150, Kentigern House, 65 Brown St., Glasgow G2 8EX, 1997.
18. R Wieringa, W. de Jonge, P. Spruit, *Roles and Dynamic Subclasses: A Modal Logic Approach*, IS-CORE report, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.
19. The UML Notation version 1.1, UML resource center, `http://www.rational.com`, 1997.