A. Evans
Dept. of Computing, University of York
andye@cs.york.ac.uk

Rigorous Development in UML

Abstract. The Unified Modelling Language (UML) is becoming the de facto industry standard notation for object-oriented analysis and design. In this paper we propose a development process using UML and other notations which supports formal analysis and verification, so enabling the notation to be used for highly critical systems.

We will illustrate the development process using a small example of a traffic light control system.[1]

1 Introduction

The UML [12] combines and extends elements of previous OO notations such as OMT, Booch and Objectory. In contrast to these methods, its notations are precisely defined using the Object Constraint Language (OCL) and a metamodel to express the allowed forms of diagrams and their properties. In previous papers we have shown how the semantic meaning of some UML diagrams can also be precisely defined [8, 7, 3]. This semantics supports the use of *transformational development*: the refinement of abstract models towards concrete models, using design steps which are known to be correct with respect to the semantics (all properties of the abstract model remain valid in the refined model).

For highly critical applications (systems where the consequence of incorrect functioning may include loss of life or severe financial loss), it is important that the development process used can help detect and eliminate errors. The process should in particular support the *verification* of refined models against abstract models by comparing their semantics.

A number of problems have been recognised with the implicit method for using UML [13], for example:

1. Use cases have been extended from being simply a requirements elicitation tool, to being a notation which (via the **extends** and **uses** dependencies between use cases) can describe quite complex control flow. Premature and inappropriate design can therefore result.

2. Statecharts are a design-oriented notation not ideally suited for abstract behaviour specification, and are used to describe the behaviour of individual objects, instead of system-level modelling.

We attempt to remedy the first problem by not allowing dependencies between use cases, and by using Yourdon-style Data and Control-flow Diagrams

[1] This work was partly supported by a grant from the Laboraturio de Methodos Formais of the Departamento de Informatica of Pontificia Universidade Catolica do Rio de Janeiro.

(DCFD's) [14] to describe the overall context of data and control flows between the system and the external agents and devices it interacts with. We deal with the second problem by using *operation schemas* which describe in an abstract way the response of the system or an object to an input event or request.

Our proposed process can be summarised as follows:

1. Requirements – modelled using Yourdon context diagrams and/or UML use case diagrams (without dependencies between use cases).
2. Essential Specification – described using UML class diagrams, operation schemas (from Fusion and Octopus), statecharts and sequence diagrams.
3. Design – modelled using UML class diagrams, statecharts, sequence diagrams and collaboration diagrams.

In order to support verification, a number of well-defined relationships between these models can be given:

1. Each input event/message on the system context diagram should have a system response described by an operation schema.
2. The effect described by an operation schema for an event **e** must be established by the completed response sequence to **e** described in design level statecharts, that is, by the transitions specified for **e** and the set of their generated events and transitions.
3. Design level class diagrams should satisfy all the properties asserted in the specification level class diagrams.
4. Sequence diagrams should be consistent with collaboration diagrams: the structure of object inter-calling should be the same.
5. Collaboration diagrams should be consistent with statecharts: messages sent by an object in response to a message **m** should correspond to events generated from transitions for **m** in the statechart of the object.

Of these, 2 and 3 are formal verification steps, because class diagrams, operation schemas and statecharts have precise formal semantics in our formalisation. 1, 4 and 5 are syntactic checks which could be implemented in CASE tools.

2 Semantics and Verification Rules

A mathematical semantic representation of UML models can be given in terms of *theories* in extended first-order set theory as in the semantics presented for Syntropy in [2] and VDM^{++} in [10]. In order to reason about real-time specifications the more general version, Real-time Action Logic (RAL) [10] can be used.

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.*

Theories can be used to represent classes, instances, associations and general submodels of a UML model. These models are therefore taken as *specifications*: they describe the features and properties which should be supported by any implementation that satisfies the model. In terms of the semantics, theory **S** satisfies theory **T** if there is an interpretation σ of the symbols of **T** into those of **S** under which every property of **T** holds:

$$\mathbf{S} \vdash \sigma(\varphi)$$

for every theorem φ of **T**. A design model **D** with theory **S** is a correct refinement of abstract model **C** with theory **T** if **S** satisfies **T**.

In addition to standard mathematical notation such as $\mathbb{F}$ for "set of finite sets of", etc, RAL theories can use the following notations:

1. For each classifier or state **X** there is an attribute $\overline{\mathbf{X}} : \mathbb{F}(\mathbf{X})$ denoting the set of existing instances of **X**.

2. If α is an action symbol, and **P** a predicate, then $[\alpha]\mathbf{P}$ is a predicate which means "every execution of α establishes **P** on termination", that is, **P** is a *postcondition* of α.

3. For every action α there are functions $\uparrow(\alpha,\mathbf{i})$, $\downarrow(\alpha,\mathbf{i})$, $\leftarrow(\alpha,\mathbf{i})$ and $\rightarrow(\alpha,\mathbf{i})$ of $\mathbf{i} : \mathbb{N}_1$ which denote the activation, termination, request send and request arrival times, respectively, of the **i**-th invocation of α. These times are ordered as:

$$\leftarrow(\alpha,\mathbf{i}) \;\leq\; \rightarrow(\alpha,\mathbf{i}) \;\leq\; \uparrow(\alpha,\mathbf{i}) \;\leq\; \downarrow(\alpha,\mathbf{i})$$

   Also

$$\mathbf{i} \leq \mathbf{j} \;\Rightarrow\; \leftarrow(\alpha,\mathbf{i}) \;\leq\; \leftarrow(\alpha,\mathbf{j})$$

4. If α and β are actions, then $\alpha \supset \beta$ "α calls β" is defined to mean that

$$\forall\, \mathbf{i} : \mathbb{N}_1 \cdot \exists\, \mathbf{j} : \mathbb{N}_1 \cdot \uparrow(\alpha,\mathbf{i}) = \uparrow(\beta,\mathbf{j}) \;\wedge\; \downarrow(\alpha,\mathbf{i}) = \downarrow(\beta,\mathbf{j})$$

Either Z or OCL notation could be used for axioms in theories, representing the semantics or constraints of UML models. In [9] we define a translation from OCL into Z.

2.1 Object Models

A UML class **C** is represented as a theory of the form given in Figure 1. Each

[htbp]

theory $\mathit{\Gamma}_{\mathrm{C}}$

types $\mathbf{C}$

attributes $\overline{\mathbf{C}} : \mathbb{F}(\mathbf{C})$
$\quad \mathbf{self} : \mathbf{C} \to \mathbf{C}$
$\quad \mathbf{att}_1 : \mathbf{C} \to \mathbf{T}_1$
$\quad \dots$

actions $\mathbf{create}_{\mathrm{C}}(\mathbf{c} : \mathbf{C}) \quad \{\overline{\mathbf{C}}\}$
$\quad \mathbf{kill}_{\mathrm{C}}(\mathbf{c} : \mathbf{C}) \quad \{\overline{\mathbf{C}}\}$
$\quad \mathbf{op}_1(\mathbf{c} : \mathbf{C}, \mathbf{x} : \mathbf{X}_1) : \mathbf{Y}_1$
$\quad \dots$

axioms

$$\begin{array}{l} \forall\, \mathbf{c} : \mathbf{C} \cdot \\ \quad \mathbf{self}(\mathbf{c}) = \mathbf{c} \;\wedge \\ \quad [\mathbf{create}_{\mathrm{C}}(\mathbf{c})](\mathbf{c} \in \overline{\mathbf{C}}) \;\wedge \\ \quad [\mathbf{kill}_{\mathrm{C}}(\mathbf{c})](\mathbf{c} \notin \overline{\mathbf{C}}) \end{array}$$

Fig. 1. Theory of Class **C**

instance attribute $\mathbf{att_i} : \mathbf{T_i}$ of **C** gains an additional parameter of type **C** in the class theory $\mathit{\Gamma}_{\mathbf{C}}$ and similarly for operations[2]. Class attributes and actions do not gain the additional **C** parameter as they are independent of any particular instance. We can denote $\mathbf{att}(\mathbf{a})$ for attribute **att** of instance **a** by the standard OO notation **a.att**, and similarly denote actions $\mathbf{act}(\mathbf{a}, \mathbf{x})$ by $\mathbf{a}.\mathbf{act}(\mathbf{x})$.

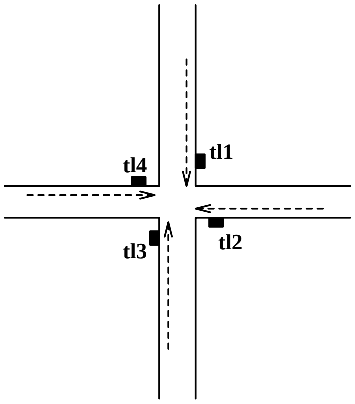
Similarly each association **lr** can be interpreted in a theory which contains an attribute $\overline{\mathbf{lr}}$ representing the current extent of the association (the set of pairs in it) and actions **add_link** and **delete_link** to add and remove pairs (links) from this set. Axioms define the cardinality of the association ends and other properties of the association. In particular, if **ab** is an association between classes **A** and **B**, then $\overline{\mathbf{ab}} \subseteq \overline{\mathbf{A}} \times \overline{\mathbf{B}}$, so membership of $\overline{\mathbf{ab}}$ implies existence for elements of a link.

3 Software Requirements

The system to be constructed in this case study is a controller for two pairs of traffic lights at a crossroads (Figure 2). Traffic lights 1 and 3 must always show the same indication, as must lights 2 and 4. Traffic lights cycle from Green to

[2] The class theory can be generated from a theory of a typical **C** instance by means of an **A**-morphism [2].

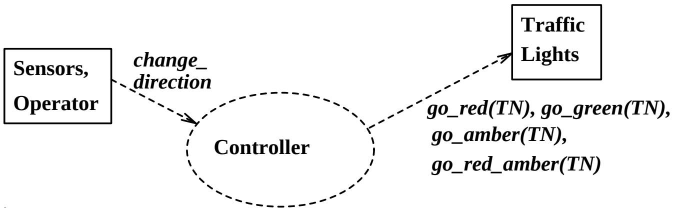
Amber to Red on the 'go red' cycle, and Red, Red and Amber, Green, on the 'go green' cycle. There is a delay of 3 seconds in the Red and Amber state, and 5 seconds in the Amber state. The safety requirement is that at least one pair of traffic lights must be red at any given time.



[htbp]

Fig. 2. Traffic Light Layout

The system responds to a signal 'change_direction'. The response should be to set the currently red signals to green, and the currently green signals to red. **TN** is the type $\{1, 2, 3, 4\}$ indexing traffic lights. A Use Case diagram similar to



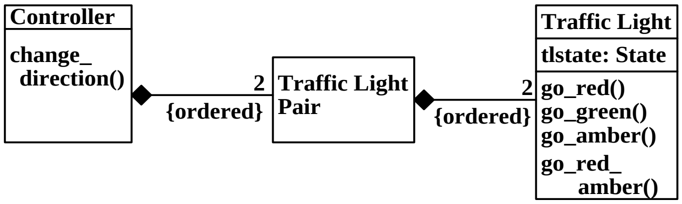
[htbp]

Fig. 3. Context Diagram of Traffic Light Control System

Figure 3 could also be defined, with agents being the signal generator (operator or sensors) and the traffic light actuators.

4 Essential Specification Level

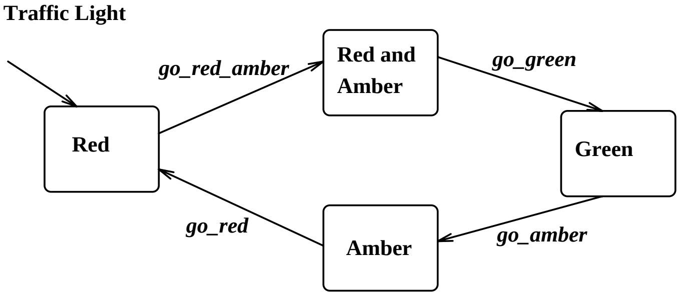
We could model the system as a collection of two traffic light pairs containing distinct traffic lights **tl**1 and **tl**3, and **tl**2 and **tl**4. The abstract object model is given in Figure 4. **State** is the enumerated type $\{\mathbf{green}, \mathbf{amber}, \mathbf{red}, \mathbf{red\_amber}\}$ for the illumination state of an individual traffic light.



[htbp]

Fig. 4. Abstract Object Model of Traffic Light Controller

The behaviour of individual traffic lights is given in Figure 5.



[htbp]

Fig. 5. Statechart of Traffic Light

A sequence diagram would show that the traffic light must be in the **red_amber** state for at least 3 seconds, and in the **amber** state for at least 5 seconds.

The invariant that the light pairs always illuminate the same lamps is expressed as an invariant of **TrafficLightPair**:

$$\begin{array}{l} \underline{\mathbf{TrafficLightPair}} \\ \mathbf{self}.\mathbf{traffic\_light}[1].\mathbf{tlstate} = \mathbf{self}.\mathbf{traffic\_light}[2].\mathbf{tlstate} \end{array}$$

where **composite**[**i**] denotes the **i**-th element in a composite list of objects.

The safety constraint is formalised as:

$$\begin{array}{l} \underline{\mathbf{Controller}} \\ \mathbf{self}.\mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[1].\mathbf{tlstate} = \mathbf{red} \quad \mathbf{or} \\ \mathbf{self}.\mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[1].\mathbf{tlstate} = \mathbf{red} \end{array}$$

We need to show that these invariants are maintained by operation schemas and their implementations. In implementations we may require that the invariants are also maintained at a finer level of granularity than the complete execution of the operation (ie, they are true at certain points during the operation execution) depending on the concurrency policy in force.

The operation schemas express the required effects of the operations listed in the use cases of the system, without any decomposition into methods of individual objects. As we discuss in [11], this style of essential model description is often clearer than the artificial localisation of such specifications used in Syntropy essential models [1].

operation change_direction
reads traffic_light_pair, traffic_light
writes tlstate

precondition

```
(tl1.tlstate = green and tl3.tlstate = green and
 tl2.tlstate = red and tl4.tlstate = red) or
(tl1.tlstate = red and tl3.tlstate = red and
 tl2.tlstate = green and tl4.tlstate = green)
```

postcondition

```
if tl1.tlstate@pre = green
then
  tl1.tlstate = red and tl2.tlstate = green and
  tl3.tlstate = red and tl4.tlstate = green
else
  if tl1.tlstate@pre = red
  then
    tl1.tlstate = green and tl2.tlstate = red and
    tl3.tlstate = green and tl4.tlstate = red
```

tl1, etc are abbreviations for OCL expressions:

tl1 = **traffic_light_pair**[1].**traffic_light**[1]
tl2 = **traffic_light_pair**[2].**traffic_light**[1]
tl3 = **traffic_light_pair**[1].**traffic_light**[2]
tl4 = **traffic_light_pair**[2].**traffic_light**[2]

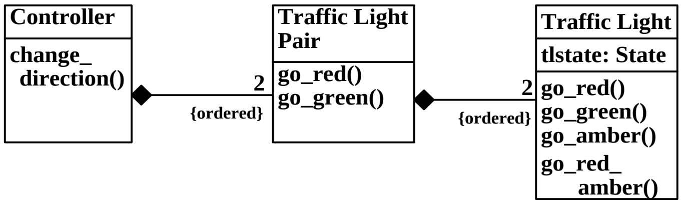
The notation **e**@**pre** denotes the value of **e** at activation of the operation. If an attribute **e** does not occur in the **writes** list, then **e** can be used instead of **e**@**pre** since the values of these expressions are then the same.

The precondition excludes the case that **change_direction** occurs if **tl1**.**tlstate**@**pre** $\in$ {**amber**, **red_amber**}, or other combinations other than a 'stable state'.

In this description, there is no detail concerning *how* these changes of state are brought about. This is a concern of later design stages.

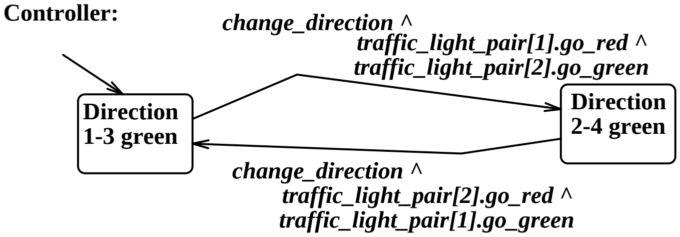
5 Design

We enhance the original object model to include additional operations for the **TrafficLightPair** class (Figure 6).



[htbp]

Fig. 6. Refined Object Model of Traffic Light Controller


Controller:
change_direction ^
traffic_light_pair[1].go_red ^
traffic_light_pair[2].go_green
Direction
1-3 green
Direction
2-4 green
change_direction ^
traffic_light_pair[2].go_red ^
traffic_light_pair[1].go_green

[htbp]

Fig. 7. Statechart of **Controller**

Statecharts for the controller and traffic light pair classes are given in Figures 7 and 8.

Verification that the reaction achieves the effect specified in the operation schema is direct. We have to check that each transition for **change_direction** in the controller statechart results in a poststate satisfying the postcondition of the operation schema, under the assumption of the guard on the transition (including the properties implied by membership of the source state) and the precondition of the operation schema.

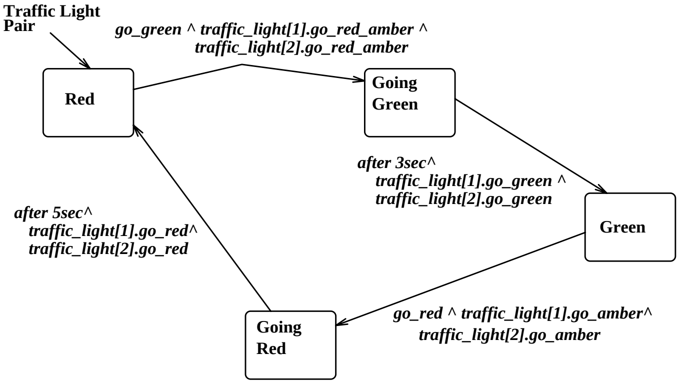
For example, in the case that direction 1-3 is initially green, the transition for **change_direction** in the **Controller** state machine terminates once the transition for **go_red** on the state machine for **traffic_light_pair**[1] and then the transition for **go_green** on the state machine for **traffic_light_pair**[2] terminate. The first of these transitions results in a state where

$$\begin{array}{lcl}
\mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[1].\mathbf{tlstate} & = & \mathbf{red} \\
\mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[2].\mathbf{tlstate} & = & \mathbf{red} \\
\mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[1].\mathbf{tlstate} & = & \mathbf{red} \\
\mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[2].\mathbf{tlstate} & = & \mathbf{red}
\end{array}$$

and the second in the specified state

$$\begin{array}{lcl}
\mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[1].\mathbf{tlstate} & = & \mathbf{red} \\
\mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[2].\mathbf{tlstate} & = & \mathbf{red} \\
\mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[1].\mathbf{tlstate} & = & \mathbf{green} \\
\mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[2].\mathbf{tlstate} & = & \mathbf{green}
\end{array}$$

as required. It can also be checked that each transition maintains the safety invariant.


Traffic Light Pair
go_green ^ traffic_light[1].go_red_amber ^ traffic_light[2].go_red_amber
Red
Going Green
after 3sec^ traffic_light[1].go_green ^ traffic_light[2].go_green
Green
after 5sec^ traffic_light[1].go_red^ traffic_light[2].go_red
go_red ^ traffic_light[1].go_amber^ traffic_light[2].go_amber
Going Red

[htbp]

Fig. 8. Statechart of **TrafficLightPair**

Finally, we need to check that the disjunction of the guards of all the transitions for **change_direction** is logically weaker than the operation schema precondition, which is also the case.

6 The Role of Transformations

In a complex development UML models may have hundreds of classes and associations. Any changes in the structure of this data from the abstract to refined models must be carried out in a way which ensures the correctness of the concrete system with respect to the abstract.

There are three kinds of transformation which have been developed:

1. Enhancement transformations, which simply extend a model with new model elements. For example, adding new classes, invariants, attributes, operations or associations to a class diagram, or introducing state nesting or new transitions to a statechart. Figure 6 represents an enhancement of Figure 4.
2. Reductive transformations, which allow a model expressed in the full UML notation to be reexpressed in a sublanguage of this notation. For example, 'flattening' of nested or concurrent states into equivalent sets of basic states, or replacing association qualifiers by association classes [8].

   These transformations can give a partial semantics of full UML models in terms of a sublanguage of UML [4].
3. Refinement transformations, which support re-writing models in ways which lead from analysis to design and implementation.

   The refinement transformations on class diagrams we have verified are: *refining class invariants*, *rationalising disjoint associations*, *eliminating many-many associations*, *relational composition of aggregations*, *specialising interfaces*, *strengthening association constraints*, *relational composition of selector associations* [9, 7], *moving associations into aggregations* [8].

The transformations on statecharts we have verified are: *source and target splitting of transitions*, *abstracting events*, *strengthening transition guards*, *eliminating transitions with false guards*, *collecting common transitions*, *restricting source of transitions*, *introducing sequencing and iteration* [9, 7].

These transformations also include the introduction of *design patterns* [6].

Enhancement transformations are *complete* in the sense that any UML class diagram or statechart can be constructed by iterating such transformations on an initially empty class diagram or statechart. They are also simple to verify, since they result in a logically stronger theory (although possibly an inconsistent theory).

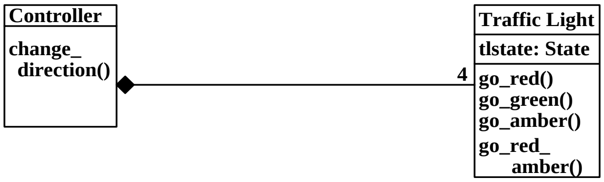
Reductive transformations are complete in the sense that any model in the full notation can be equivalently expressed in the subnotation by applying these transformations. These transformations apply to statecharts without history entry nodes or deferred events, and reduce them to state machines without nested or concurrent states: ie, in which all states are basic.

Refinement transformations are not complete, in that it is possible to devise refinements which are not expressible as a combination of the transformations given above. It should however be the case that these transformations cover a wide range of those used in practice by developers.

Examples of transformations are given in the following sections and in [7, 8].

6.1 Class Models and Transformations

Consider an alternative analysis model of the traffic light control system where the **TrafficLightPair** class is not defined during analysis, so that the controller is directly related to the four separate traffic light objects (Figure 9). For this version the operation schema is the same as that in Section 4 except that **tl**1, etc, abbreviate



[htbp]

Fig. 9. Initial Class Diagram of Traffic Light System

tl1 = **traffic_light**[1]
tl2 = **traffic_light**[2]
tl3 = **traffic_light**[3]
tl4 = **traffic_light**[4]

and **traffic_light_pair** is not in the **reads** list. This version of the system can be refined to that presented in Section 4, using the following transformations.

Composing Aggregations Composition associations in UML represent a strong 'part of' relationship between a 'whole' entity and several 'part' entities. Such a relationship should have several properties [12]:

1. One-many or one-one (a whole can have many parts, but a part cannot belong to different wholes at the same time): if whole objects $\mathbf{a}$ and $\mathbf{a}'$ are related to the same part $\mathbf{b}$, then $\mathbf{a} = \mathbf{a}'$.
2. Deletion propagating: deleting the whole deletes its parts.
3. Transitive.
4. Irreflexive

If we have a situation as in Figure 10, where two aggregations $\mathbf{ab}$ and $\mathbf{bc}$ exist between different classes $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, which have no common objects, then the relational composition $\mathbf{ac} = \mathbf{ab}; \mathbf{bc}$ of these two aggregations also satisfies the properties 1 to 4 above if $\mathbf{ab}$ and $\mathbf{bc}$ do:

1. If $(\mathbf{a}, \mathbf{c}), (\mathbf{a}', \mathbf{c}) \in \mathbf{ac}$, then there are $\mathbf{b}$, $\mathbf{b}'$ such that
$$(\mathbf{b}, \mathbf{c}), (\mathbf{b}', \mathbf{c}) \in \mathbf{bc} \;\wedge\; (\mathbf{a}, \mathbf{b}), (\mathbf{a}', \mathbf{b}') \in \mathbf{ab}$$
But then $\mathbf{b} = \mathbf{b}'$ by property 1 of $\mathbf{bc}$, and so $\mathbf{a} = \mathbf{a}'$ by property 1 of $\mathbf{ab}$.
2. $\mathbf{kill_A}(\mathbf{a}) \supset \mathbf{kill_B}(\mathbf{b})$ for each $(\mathbf{a}, \mathbf{b}) \in \mathbf{ab}$, and $\mathbf{kill_B}(\mathbf{b}) \supset \mathbf{kill_C}(\mathbf{c})$ for each $(\mathbf{b}, \mathbf{c}) \in \mathbf{bc}$, so $\mathbf{kill_A}(\mathbf{a}) \supset \mathbf{kill_C}(\mathbf{c})$ for each $(\mathbf{a}, \mathbf{c}) \in \mathbf{ac}$, as required.
3. $\mathbf{ab}$ and $\mathbf{bc}$ are trivially transitive since there can be no pairs $(\mathbf{x}, \mathbf{y}), (\mathbf{y}, \mathbf{z}) \in \mathbf{ab}$, etc. Likewise $\mathbf{ac}$ is trivially transitive since $\mathbf{C}$ is disjoint from $\mathbf{A}$.
4. Similarly $\mathbf{ac}$ is trivially irreflexive.

[htbp]

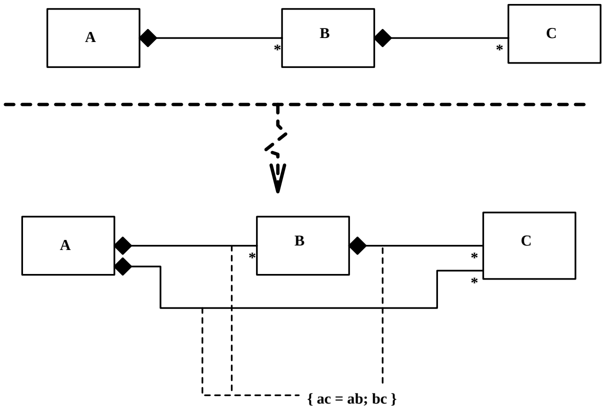


Fig. 10. Transitivity of Composition Aggregations

If there are specific cardinalities $1 : \mathbf{n}$, $1 : \mathbf{m}$ for **ab** and **bc** respectively, then **ac** has cardinality $1 : (\mathbf{n} * \mathbf{m})$.

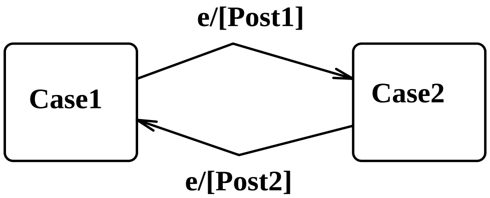
In the case of the traffic light system, we can use this transformation to deduce that the model of Figure 4 refines the model of Figure 9.

6.2 Refinement of Operation Schemas to Statecharts

If an operation schema definition for operation **e** has the form

```
precondition  Case1 or Case2
if Case1@pre
then Post1
else
  if Case2@pre
  then Post2
```

where $\mathbf{Post}2 \Rightarrow \mathbf{Case}1$, $\mathbf{Post}1 \Rightarrow \mathbf{Case}2$ and $\neg\,(\mathbf{Case}1 \wedge \mathbf{Case}2)$, then this is expressible as a binary state machine of the form of Figure 11. In general, if a system involves a fixed finite number of objects (eg, objects representing actuators or sensors in a reactive system) each of which has only finitely many states, then it is possible to mechanically produce a finite state machine from the set of its operation schemas. Such a finite state machine may be extremely large however, and need further abstraction before it can be used as a useful analysis model.



[htbp]

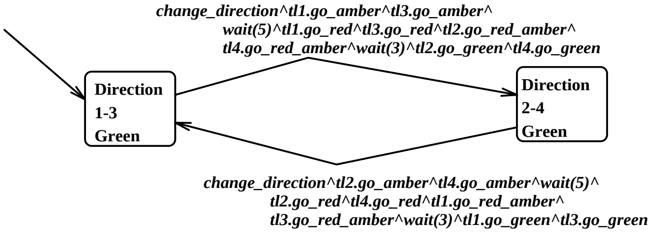
Fig. 11. Implementation of Operation Schema

The statechart of Figure 11 can then be further refined to replace postconditions [**Post**] by suitable sequencing of actions which ensure these postconditions.

In the case of the traffic light control system, we could apply this transformation to the operation schema of Section 6.1 to obtain Figure 12 (**tl**1 denotes **traffic_light**[1], etc).

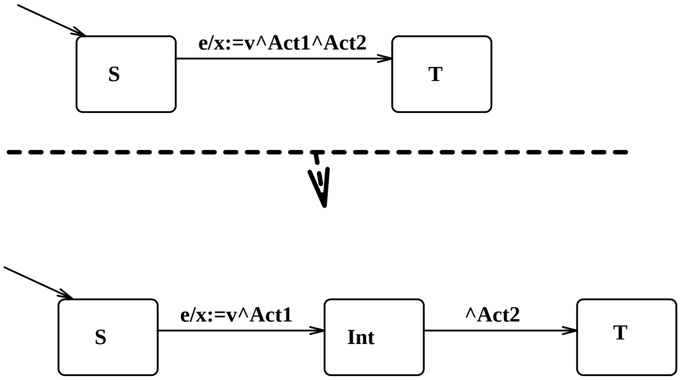
6.3 State Machine Models and Transformations

Sequential Decomposition Sequential decomposition allows the introduction of procedural control flow into a statechart description of a method. It can be used in the step from a statechart to an activity diagram. Figure 13 shows a typical


change_direction^tl1.go_amber^tl3.go_amber^
wait(5)^tl1.go_red^tl3.go_red^tl2.go_red_amber^
tl4.go_red_amber^wait(3)^tl2.go_green^tl4.go_green
Direction
1-3
Green
Direction
2-4
Green
change_direction^tl2.go_amber^tl4.go_amber^wait(5)^
tl2.go_red^tl4.go_red^tl1.go_red_amber^
tl3.go_red_amber^wait(3)^tl1.go_green^tl3.go_green

[htbp]

Fig. 12. Abstract Controller Statechart



[htbp]

Fig. 13. Sequential Decomposition Example

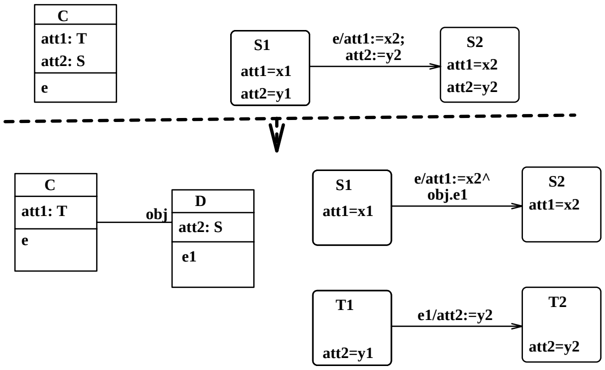
example. Here, **Int** is a new state, with no other incident transitions. The theory interpretation is that $\mathbf{t}_1$, the abstract transition for **e** from **S**, is mapped to $\mathbf{t}_2;\ \mathbf{t}_3$ where $\mathbf{t}_2$ is the concrete transition for **e** from **S**, and $\mathbf{t}_3$ is the automatic transition from **Int**.

If the original transition had labelling $\mathbf{e}/\mathbf{x} := \mathbf{v} \frown \mathbf{Act}1 \frown \mathbf{wait}(\mathbf{n}) \frown \mathbf{Act}2$ where $\mathbf{wait}(\mathbf{n})$ indicates a delay of at least $\mathbf{n}$ time units, then the decomposed version has instead the labelling $\mathbf{after\ n} \frown \mathbf{Act}2$ on transition $\mathbf{t}_3$.

Annealing A transformation involving refinement of both class diagrams and state machines is *annealing* [5]. This involves the replacement of a local attribute of a class with a reference to an object, or the addition of an intermediate reference between objects. In terms of dynamic models, a transition in a single statechart is replaced by a succession of two transitions in separate statecharts, one invoked by the other.

Figure 14 shows a typical case with two attributes.

In the traffic light case study, an annealing step from the models given in Section 6.1 and Figure 12 would introduce **TrafficLightPair** objects and define



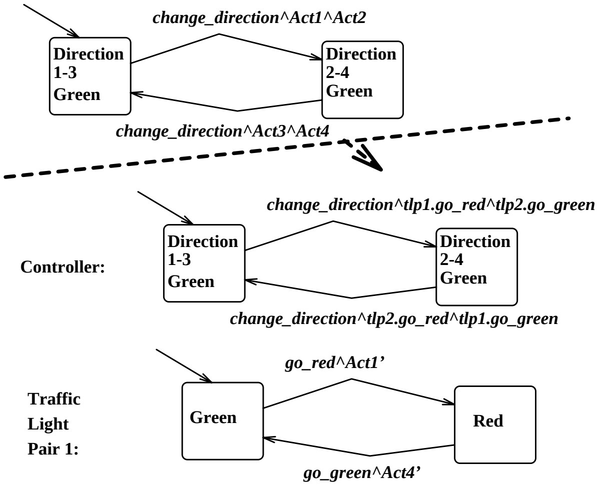
[htbp]

Fig. 14. Annealing

the theory interpretation σ:

$$\begin{array}{lcl}
\mathbf{traffic\_light}[1] & \longmapsto & \mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[1] \\
\mathbf{traffic\_light}[2] & \longmapsto & \mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[1] \\
\mathbf{traffic\_light}[3] & \longmapsto & \mathbf{traffic\_light\_pair}[1].\mathbf{traffic\_light}[2] \\
\mathbf{traffic\_light}[4] & \longmapsto & \mathbf{traffic\_light\_pair}[2].\mathbf{traffic\_light}[2]
\end{array}$$

This transformation is shown in Figure 15. $\mathbf{Act}1$ is $\mathbf{tl}1.\mathbf{go\_amber}\frown\mathbf{tl}3.\mathbf{go\_amber}\frown$ $\mathbf{wait}(5)\frown\mathbf{tl}1.\mathbf{go\_red}\frown\mathbf{tl}3.\mathbf{go\_red}$, where $\mathbf{tl}1 = \mathbf{traffic\_light}[1]$, $\mathbf{tlp}1 = \mathbf{traffic\_light\_pair}[1]$, etc., whilst $\mathbf{Act}1'$ is $\mathbf{Act}1$ with $\mathbf{tl}1 = \mathbf{traffic\_light}[1]$, $\mathbf{tl}3 = \mathbf{traffic\_light}[2]$.


change_direction^Act1^Act2
Direction
1-3
Green
Direction
2-4
Green
change_direction^Act3^Act4
change_direction^tlp1.go_red^tlp2.go_green
Controller:
Direction
1-3
Green
Direction
2-4
Green
change_direction^tlp2.go_red^tlp1.go_green
go_red^Act1'
Traffic
Light
Pair 1:
Green
Red
go_green^Act4'

[htbp]

Fig. 15. Annealing of Traffic Light System

Combining this with sequential decomposition of the traffic light pair transitions shows that the design of Section 5 refines the version given in Section 6.1.

Conclusions

We have proposed a systematic development process using UML notations. The steps of this process can be verified, in principle, and make use of formally correct transformations on UML models. We are currently working on tool support for such a process, which should enable it to be trialed for use in an industrial context.

References

1. S. Cook, J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy*, Prentice Hall, 1994.
2. J C Bicarregui, K C Lano, T S E Maibaum, *Objects, Associations and Subsystems: a hierarchical approach to encapsulation*, ECOOP 97, LNCS, 1997.
3. A. Evans, R. France, K. Lano, B. Rumpe, *Developing the UML as a formal modelling language*, UML 98 Conference, Mulhouse, France, 19 98.
4. M. Gogolla, M. Richters, *Equivalence Rules for UML Class Diagrams*, UML 98.
5. S. Goldsack, K. Lano, E. Durr, *Invariants as Design Templates in Object-based Systems*, Workshop on Foundations of Component-based Systems, ESEC 97.
6. K. Lano, N. Malik, *Reengineering Legacy Applications using Design Patterns*, STEP '97, IEEE Press, 1997.
7. K. Lano, J. Bicarregui, *Semantics and Transformations for UML Models*, UML 98 Conference, Mulhouse, France, 1998.
8. K. Lano, J. Bicarregui, *Formalising the UML in Structured Temporal Theories*, ECOOP 98 Workshop on Behavioural Semantics, Technical Report TUM-I9813, Technische Universitat Muchen, 1998.
9. K. Lano, J. Bicarregui, *UML Refinement and Abstraction Transformations*, ROOM 2 Workshop, University of Bradford, 1998.
10. K Lano, *Logical Specification of Reactive and Real-Time Systems*, *Journal of Logic and Computation*, Vol. 8, No. 5, pp 679–711, 1998.
11. K. Lano, R. France, J-M. Bruel, *A Semantic Comparison of Fusion and Syntropy*, to appear in *Object-oriented Systems*, 1998.
12. Rational Software et al, *UML Documentation*, Version 1.1, http://www.rational.com/uml, 1997.
13. A. Simons, I. Graham, *37 Things That Don't Work in Object-Oriented Modelling with UML*, ECOOP 98 Workshop on Behavioural Semantics, Technical Report TUM-I9813, Technische Universitat Muchen, 1998.
14. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, 1989.

This article was processed using the LaTeX macro package with LLNCS style