Refinement through Pictures: Formalising Syntropy Refinement Concepts

K. Lano, J. Bicarregui
Dept. of Computing, Imperial College, 180 Queens Gate, London, SW7 2BZ

October 31, 1997

Abstract

This paper provides techniques for formally justifying the refinement steps given for the Syntropy method of Cook and Daniels.

It uses a temporal logic formalism, the Object Calculus, to formalise the models of Syntropy in modules which are theories, linked by theory-preserving morphisms. Refinements are also characterised by particular forms of interpretations between theories.

The intention is to provide support for a system which allows refinement steps to be carried out via diagrammatic descriptions, rather than mathematical formulae, and hence to enhance the usability of a formal approach to object-oriented software development.

1 Introduction

Syntropy [4] is a methodology for object-oriented analysis and design similar to OMT [17] with additional formal specification elements derived from Z [19]. It represents a significant advance over previous object-oriented methods in giving mathematical specifications of data models and dynamic behaviour. Concepts from Syntropy have been used in the UML [18], particularly in the development of its object constraint language.

Three distinct levels of modelling are used in Syntropy. At each of these three levels, type view diagrams depict the structure of object classes. Objects have attributes of non-object types. Associations between classes are depicted by connecting lines. Statecharts [9] are also used at each of the three levels. However, different models of communication are used at each level of abstraction.

- *Essential models* describe the problem domain of the application. They describe the system as a whole including the proposed software solution and its environment. They use events and broadcast communications to abstract from the localisation of methods in classes.
- *Specification models* abstractly model the requirements of the software application, hence defining the software/environment boundary. They decompose a reaction to an external event into a series of event generations and internal reactions by specific classes.
- *implementation models* model the required software in detail. In addition, object interaction graphs (termed *mechanisms* in Syntropy) are used at this level, with object to object message passing.

Syntropy adopts a number of mathematical notations, however, a semantics is only indicated for data models. In addition, there is no formal definition of refinement between models.

There are two main forms of refinement which arise in the Syntropy method:

1. Refinement which involves a change in granularity of the actions of the system, such as factoring a transition which establishes a complex postcondition into a sequence of internal transitions which each establish part of the postcondition.

2. Refinement which involves no change in granularity – essentially a form of subtyping as enumerated in [4, Chapter 8].

We will show how these forms of refinement can be formally justified in terms of theory interpretation and extension, using the semantics of Syntropy given in [2].

An important point is that a direct refinement from an essential to a specification model will not in general exist: not all entities in the essential model will have interpretations in the specification, if they lie outside the software system to be constructed. In addition, permission guards in the essential model will only appear as preconditions in the specification model.

2 Syntropy Semantics

2.1 The Object Calculus

The Object Calculus [7] is a formalism based on structured first order theories composed by morphisms between them.

An object calculus theory models a component of a system. It consists of a set $\mathcal{S}$ of constant symbols, a set $\mathcal{A}$ of *attribute symbols* (denoting time-varying data) and a set $\mathcal{G}$ of *action symbols* (denoting atomic operations). Axioms describe the types of the attributes and dynamic properties of the actions.

A global, discrete linear model of time is adopted (eg. [11]) and axioms are specified using temporal logic operators including: $\bigcirc$ (in the next state), $\mathcal{U}$ (until), $\Box$ (always in the future) and $\diamond$ (sometime in the future). The predicate BEG is true exactly at the first moment.

The temporal operators are also expression constructors. If e is an expression, $\bigcirc e$ denotes the value of e in the next time interval, etc.

In the style of [8], theories are composed by morphisms to yield a modular definition of a whole system. The Object Calculus defines a notion of locality which ensures that only actions local to a particular theory can effect the value of the local attributes. For each theory we have a logical axiom

$$\bigvee_{g_i \in \mathcal{G}} g_i \;\vee\; \bigwedge_{a \in \mathcal{A}} a = \bigcirc a$$

"Either some action g_i of the theory executes in the current interval, or every attribute a of the theory remains unchanged in value over the interval."

2.2 Interpreting Object Types

Figure 1 depicts a fragment of a Syntropy type view diagram. A single class, A, is defined with two attributes, f and g, of (non-object) types T_1 and T_2 respectively[1].

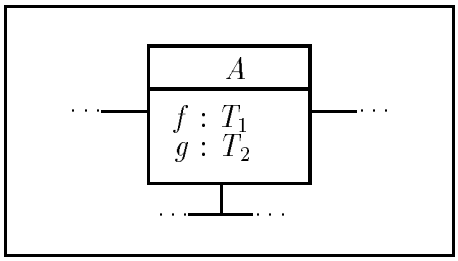

Figure 1: Part of a type view diagram

Such a diagram can be understood as a view of a typical object of the type, or it can be interpreted as depicting the entire class of such objects. To interpret this diagram, we define two Object Calculus theories.

[1] Note object-typed attributes are given via associations, see Section 2.3.

The first, A_i, gives the theory of a single instance of the type, the second, M_A, manages the collection of currently existing instances. A number of the former are then combined with the latter to form the theory Γ_A of the class.

The signature of a generic instance We define a theory, A_i for a typical object of this class. The theory of the instance introduces a sort for the type of each attribute, there are no constant (or function) symbols and, for each attribute, there is an attribute symbol for each attribute. For the present, there are no actions, we will later use information in the statechart to define the actions.

$\mathcal{S} = \{T_1, T_2\}$
$\mathcal{A} = \{f : T_1, g : T_2\}$
$\mathcal{G} = \{\ldots\}$

self A key technique used in OO notations is that an individual object can refer to itself as self whilst it's external identity (its object identifier) is given by the class. As in [6], we interpret self using A-morphisms which add the object identifier as an extra parameter when attributes and actions are globalised.

The signature of the class The creation and deletion of instances is accomplished through a class manager. Class manager and class instances are then combined to form the theory of the class. The definition of the class manager is independent of the structure of A and so is defined in terms of a general class type X.

The class manager theory, M, introduces a sort for identifiers of objects, $@X$ and no constant symbols. It is convenient to define an attribute, $\overline{X}$, to record the finite set of currently existing instances. In terms of [20], $@C$ is $ext(C)$ and the value of $\overline{C}$ at time π is $ext_\pi(C)$. There are actions of M to create and kill objects of X.

$\mathcal{S} = \{\ @X\ \}$
$\mathcal{A} = \{\ \overline{X} : \mathbb{F}\,@X\ \}$
$\mathcal{G} = \{\ create : @X, kill : @X\ \}$

Note that creating an instance does not initialise it. Creation and initialisation can be brought together via an action *new* which synchronises them.

We cannot create an existing object nor delete a non-existent one[2] (*pre-create* and *pre-kill*). Creation adds an object to the set of existing objects and deletion removes it (*post-create* and *post-kill*). We require that objects are only added or removed from the set of existing objects by creation and deletion. These six conditions can be condensed to:

$create(x) \ \Leftrightarrow\ x \notin \overline{X} \wedge x \in \bigcirc\overline{X}$
$kill(x) \ \Leftrightarrow\ x \in \overline{X} \wedge x \notin \bigcirc\overline{X}$

which concisely characterise the two actions.

We may wish to give an initialisation stating, for example, that the set of existing objects is initially empty (*initialisation*):

$\mathrm{BEG} \ \Rightarrow\ \overline{X} = \varnothing$

Embedding instances in the class At any point in time, there are a finite number of living instances. The theories of these are combined with the theory of the class manager via morphisms which name each instance according to the identifier given when it is created (Figure 2).

We combine the theory of each instance with the theory of the class via an $@A$-morphism which adds an extra parameter of type $@A$ to each attribute and action symbol [6]. This is equivalent to defining self as a constant in the instance theory which acts as a (dummy) placeholder for later identification with the object identifiers in the class theory.

[2] In a deontic setting one could use the notion of "permitted" here.

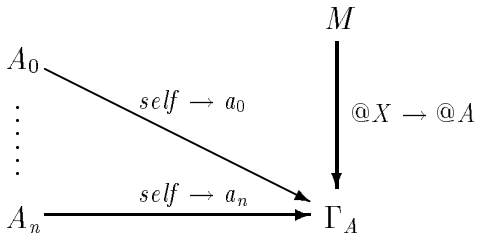

Figure 2: Instance and class manager theories are embedded in the theory of the class

The resultant theory, Γ_A, has an attribute $att(a)$ for each attribute att of each existing instance a. For example, for instance a_i and attribute f, there is an attribute $\sigma_i(f)$ in the class theory. In effect f is a (finite) partial function from $@A$ to T_1. We define a syntactic sugar which names the $\sigma_i(f)$ conveniently:

$f : @A \rightarrow T_1$

$a_i.f = \sigma_i(f)$

So, in A, f is a partial function from $@A$ to T_1 which is written in the right. A similar approach is taken to the naming of instance actions.

2.3 Interpreting Associations

We now formalise the notion of an association as depicted in Figure 3. We will interpret the association without any knowledge of the structure of the objects it associates[3]. Thus we have a generic theory of associations. We then use a renamed copy of this theory for each particular association in the model.

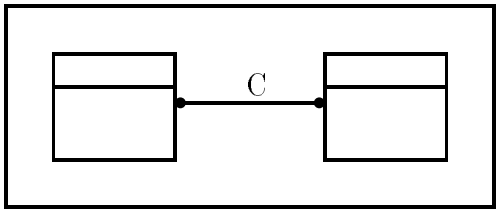

Figure 3: A simple association

We begin with the most general case, a many-many association depicted by the black "blobs" at each end of the connecting line. The same approach will also work for other cardinalities of association by requiring further axioms for the constrained cases. For this section we consider only how to interpret associations at the level of the classes. In some circumstances, such as when the association has attributes of its own, it may be desirable to make a two level construction as was done for object classes.

The association is interpreted as a many-many relation lr between object identifiers for the class on the left, $@L$ and the class on the right, $@R$[4]. Note that lr plays the same role as $\overline{X}$, it is the set of existing links in the association. The theory signature is:

$\mathcal{S} = \{ @L, @R \}$

$\mathcal{A} = \{ lr : \mathbb{F}(@L \times @R) \}$

$\mathcal{G} = \{ link : @L \times @R, unlink : @L \times @R \}$

As for object classes, we require axioms for adding and removing pairs from the relation and again have an "instance-by-instance" locality requirement which yields a characterisation of the two actions

[3] We do however assume each class theory has been constructed from instance theories and class manager theory as defined above.

[4] This turns out to be considerably more convenient than having a pair of primitive functions $r : @L \rightarrow \mathbb{F}@R$ and $l : @R \rightarrow \mathbb{F}@L$, such functions can be defined from the relation if required.

$link(l, r) \Leftrightarrow (l, r) \notin lr \wedge (l, r) \in \bigcirc lr$

$unlink(l, r) \Leftrightarrow (l, r) \in lr \wedge (l, r) \notin \bigcirc lr$

Again, it may be appropriate to add an axiom concerning the initialisation such as

BEG $\Rightarrow lr = \varnothing$

There is no axiomatic constraint between *link* for the association and *create* for the object classes here. Such constraints are given when the theories of objects and association are brought together. In keeping with encapsulation, there are no actions to update or inspect the associated object instances directly.

Bringing association and objects together Now assume that A and B are associated by C in a diagram D. D is interpreted as the co-limit of the theories for A, B and C. The class manager theories for A and B provide the "glue" which brings theories of objects and associations together. C is "glued" to each of A and B by identifying $@L$ and $@R$ with $@A$ and $@B$ respectively. Where names would otherwise clash, they are subscripted by the name of the theory from which they emanate. Purely for convenience, lr is renamed to ab in D.

Figure 4 shows the hierarchical construction of the theories involved, and how these relate (dashed arrows) to the object model. Notice that D corresponds to a theory of a "subsystem" which includes all of the items in the object model.

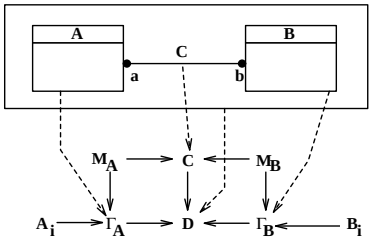

Figure 4: The type view diagram is interpreted as the colimit of the object and association theories.

We can now add axioms to D or to C which interpret the particular kind of association or aggregation required. Whatever kind of association is required, it can only link existing objects. This can be formalised by stating that the relation only relates the existing objects:

$ab \subseteq \overline{A} \times \overline{B}$

This property relates the symbols of the association theory with those of the two class manager theories (but not those of the instance theories). These symbols are all available in the colimit of the association theory and manager theories and it is therefore meaningful to give it as an axiom of the theory "C" in the above diagram.

The axiom can be written as an extra, trans-theory, postcondition for *link*:

$link(a, b) \Rightarrow a \in \bigcirc \overline{A} \wedge b \in \bigcirc \overline{B}$

which, due to locality, yields a synchronisation between *link* and *create*:

$link(a, b) \Rightarrow a \in \overline{A} \vee a.create_A$

$link(a, b) \Rightarrow b \in \overline{B} \vee b.create_B$

Optional unary associations ( ). If the "blob" on the right is white, that is each $\overline{A}$ is associated with at most one $\overline{B}$, then the relation is a (partial) map from $@A$ to $@B$:

$\forall a, b_1, b_2 \cdot (a, b_1) \in ab \wedge (a, b_2) \in ab \Rightarrow b_1 = b_2$

This can be interpreted purely in the theory of the association by strengthening the constraints on *link*:

$link(a, b) \Rightarrow a \notin \mathrm{dom}\, ab$

$link(a, b) \wedge link(a, b') \Rightarrow b = b'$

Thus this constraint is truly a specialisation of the concept of association, independent of all other constructions.

Compulsory unary associations ( ). If the blob on the right is missing altogether, that is each $\overline{A}$ is associated with exactly one $\overline{B}$, then the map is total on $\overline{A}$. Again this is a constraint between association and class manager theories:

$$\mathrm{dom}\, ab = \overline{A}$$

Note that we do not require the map to be surjective since an $@B$ can be associated with an empty set of $@A$s.

Aggregation can be formalised in a similar way [3, 12].

2.4 Interpreting Subclassing

Subclassing is represented by a subset relation between the respective sets of object identities. If D is a subclass of C then we have

$$\begin{array}{l} \overline{D} \subseteq \overline{C} \\ @D \subseteq @C \end{array}$$

Notice that therefore any attribute of C automatically becomes one of D. Similarly for associations. More precise constraints on the values of attributes $att(d, x)$ for $d \in \overline{D}$ can be defined for subtype classes without affecting the overall properties of att as an attribute of C. But we cannot change the type of this attribute to be disjoint from its type as an attribute of C.

Exclusive subclasses are represented by asserting

$$\overline{D_1} \cap \overline{D_2} = \varnothing$$

A number of subtle distinctions regarding migration between subclasses can be formalised using this basis [14]. It is reasonable to assume that non-state types cannot experience subclass migration, ie, if at any time in the lifetime of $a \in \overline{C}$ it is a member of the non-state subclass A of C, then its lifetime in A and C are the same:

$$\begin{array}{l} \forall\, a : \overline{C} \cdot a \in \overline{A} \Rightarrow \\ \qquad \Box(a \in \overline{C} \equiv a \in \overline{A}) \end{array}$$

An abstract supertype C of subclasses $D_1, \ldots, D_n$ has $\overline{C} = \overline{D_1} \cup \ldots \cup \overline{D_n}$.

2.5 Interpreting Statecharts

Statecharts are the most complex and semantically rich notation employed by Syntropy. Based on [9], they depict the state space of an object, partitioned according to "those states which distinguish the possible orderings of events" ([4], p.91).

State classes, depicted by boxes with a diagonal line in their top left hand corner, represent varying subsets of the objects of the superclass where an individual instance can move between the subtypes. Statecharts define the transitions which take instances from one state class to another (Figure 5 gives a simple example).

In Syntropy, the effect of transitions is specified by preconditions and postconditions similar to those used in Z or VDM. For example, $e_1[P]/Q$, indicates that transition e_1 can only occur if the predicate P holds and that the two-state predicate Q must then hold between the before and after states of each occurrence of e_1.

Further semantics is given by *Events* listed in the textual part at the bottom of the statechart. Events are system-wide, but can be targeted at particular objects by the use of parameters and filters. Typically, events effect a state transition in a single object of the class and have the same name as a state transition in the diagrammatic part of the statechart.

We make a syntactic distinction between the event and its associated transitions by capitalising the event name and indexing the transition names.

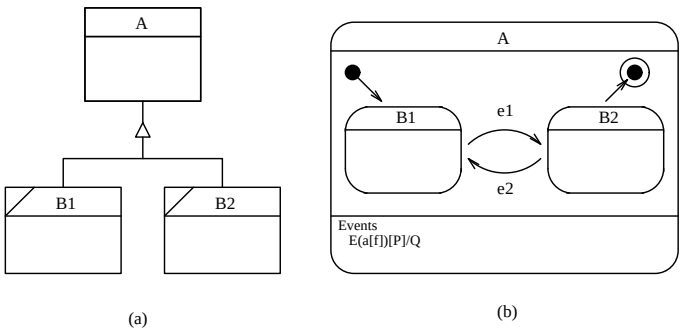

Figure 5: Typical Syntropy Statechart

2.6 Interpreting State Types and Events

The information in the class diagram is interpreted as above. The statechart defines the actions of the classes which replace the white box actions of the instance theory. Each arrow in the statechart represents a class of possible state transitions and is interpreted as an action of the instance theory.

Several arrows can be used to describe different cases of a particular system event. The event is interpreted as an action in the theory of the subtype/supertype subsystem and is synchronised with the instance actions that correspond to the required state changes. For example, in Figure 5, if e_1 and e_2 are different cases of the same event $E(a)$ then e_1 and e_2 are interpreted as separate actions in the instance theory of A whereas $E(a)$ is interpreted in the theory of the $\{ A, B_1, B_2 \}$ subsystem and then synchronised with e_1 and e_2 via an axiom of the form:

$$E(a) \Rightarrow a.e_1 \vee a.e_2$$

Filters More generally, events are of the form $E(p[F])$, where the parameter p is a list of object or value parameters and the filter F is a predicate involving the parameters, self and the class constants. Object instances that satisfy the filter will undergo the corresponding transition (depending on their state and precondition), whereas objects for which a filter fails to hold ignore the associated event.

Interpreting Preconditions Preconditions in the Syntropy essential model are intended to specify that certain transitions "cannot occur" in given circumstances. Thus we interpret preconditions as (blocking) guards which prevent execution of the transition they annotate. Consider a transition $e_1[P]/Q$ from state B_1 to state B_2. We define a *permission axiom* in the instance theory:

$$e_1 \Rightarrow P$$

which expresses that e_1 can only occur when P holds.

Permission axioms are not included in specification or implementation models. Instead guards are used as additional assumptions in the state-transition axioms.

The interpretation of preconditions as permission guards prevents preconditions from being weakened in refinement, that is, such transformations do not yield theory extensions. Thus subtyping form 5 of Chapter 8 of [4] (weakening preconditions) is not valid in essential models[5].

At the class level, each transition is also guarded by the state from which it occurs, for example, we have $a.e_1 \Rightarrow a \in \overline{B_1}$ where $\overline{C}$ denotes the set of currently existing objects of class C.

Postconditions Postconditions are expressed in terms of the change between attribute values of the current state and those after the transition. Modifications to associations which result from postconditions defining a change to one end only are assumed to be made explicit in the postcondition.

[5] In specification models, on the other hand, preconditions are to be interpreted as assumptions: any behaviour is valid if a transition is executed when its precondition is false. So preconditions *can* be weakened in specification models.

For the above transition with postcondition, Q, we have the *state-transition* axiom

$e_1 \Rightarrow Q^{\bigcirc}$

where Q is a predicate in attribute symbols f_i and f_i' and we replace f_i' with $\bigcirc f_i$ in $Q^{\bigcirc}$.

At the class level, the event additionally moves the targeted instances to state B_2:

$a.e_1 \Rightarrow a \in \bigcirc\overline{B_2}$

In specification and implementation models we have:

$$e_1 \wedge P \;\Rightarrow\; Q^{\bigcirc}$$

and

$$a.e_1 \wedge a \in \overline{B_1} \;\Rightarrow\; a \in \bigcirc\overline{B_2}$$

In implementation models there are also un-named transitions τ which are intended to activate as soon as their source state is entered (*automatic* transitions).

We can specify this behaviour by asserting that no other transition of the object can occur until this automatic transition has occurred, once its source state S is entered (there can only be at most one automatic transition executable from a particular source state at any time):

$$a \in \overline{S} \;\Rightarrow\; \neg\,(a.\alpha_1 \vee \ldots \vee a.\alpha_n)\;\mathcal{U}\;a.\tau$$

S is referred to as an *unstable* state. States without automatic transitions from them are known as *stable* states.

Generated actions Generated actions act of e_1 (in specification model statecharts) are required to eventually occur:

$$a.P \wedge a.e_1 \wedge a \in \overline{B_1} \Rightarrow \bigcirc(\neg\,(ext_1 \vee \ldots \vee ext_k)\;\mathcal{U}\;act(a))$$

where the ext_i are all external actions of the model: this asserts that $act(a)$ must occur before any other external event is detected, that is, it must occur in the same *macro* step of the execution of the state machine model as $a.e_1$. Additional axioms are used to enforce that the order in which events generated from a single object occur is the same as that specified in the generation clause.

3 Subtyping and Refinement

The relationship between the models of Syntropy is not formalised in [4], however a number of examples of techniques for transformation between models are given, and systematic transformations for *subtyping* within a model level are provided.

The theory Γ_M of a model M is the co-limit (effectively union) of the Γ_A theories of its classes.

We can then formalise refinement between models by theory morphisms: model D represents a refinement (or subtype) of model C if there is an interpretation σ of symbols of C into symbols of D which preserves the theorems of C:

$$\Gamma_C \vdash \varphi \quad \text{implies} \quad \Gamma_D \vdash \sigma(\varphi)$$

where Γ_C is the class theory of C, etc.

This definition works between essential models, specification models and implementation models, and between specification and implementation models. When comparing essential and specification models we do not expect all elements of the essential model to have an interpretation in the specification model, so σ may be partial. In addition, the permission axioms for statechart transitions are not preserved.

This is because guards in essential models describe when events can physically occur in the complete system, whilst guards in the specification and implementation models are assertions that the software *only* needs to have a defined response when the guard condition holds. Clearly the software only needs to define a response to an event in the situations where it can feasibly occur.

Using this definition, all the forms of subtyping described in [4] can be shown to be refinement steps in addition, with the exceptions that arbitrary redefinition of generated actions on transitions is not valid, and that *target splitting* of transitions is only valid if the target state is unstructured in the abstract model.

3.1 Subtyping and Refinement of Class Models

It is clear that placing more constraints on the attributes and associations of a class will yield a stronger theory as the meaning of the more constrained model. In particular, replacing associations by aggregations ('adding more diamonds') will yield a stronger theory.

It is also acceptable to add new classes, attributes and associations to the existing model. However all existing inheritance relationships must be preserved, and disjoint inheritances cannot be modified to become independent inheritances. If C is an abstract superclass partitioned by a set $D_1, \ldots, D_n$ of subclasses, then we cannot add a new element to this family.

Constraining attributes and associations also serves to produce a subtype of an individual class. However, it may also render a class inconsistent because it may become impossible for a method of the class to preserve the strengthened constraints. This indicates that additional logical checks would be a useful part of a more formal version of the Syntropy method. These checks could be based on those used in VDM^{++} or Z^{++} [14].

An example of a refinement transformation is a factorisation of one class into two or more superclasses via inheritance (Figure 6).

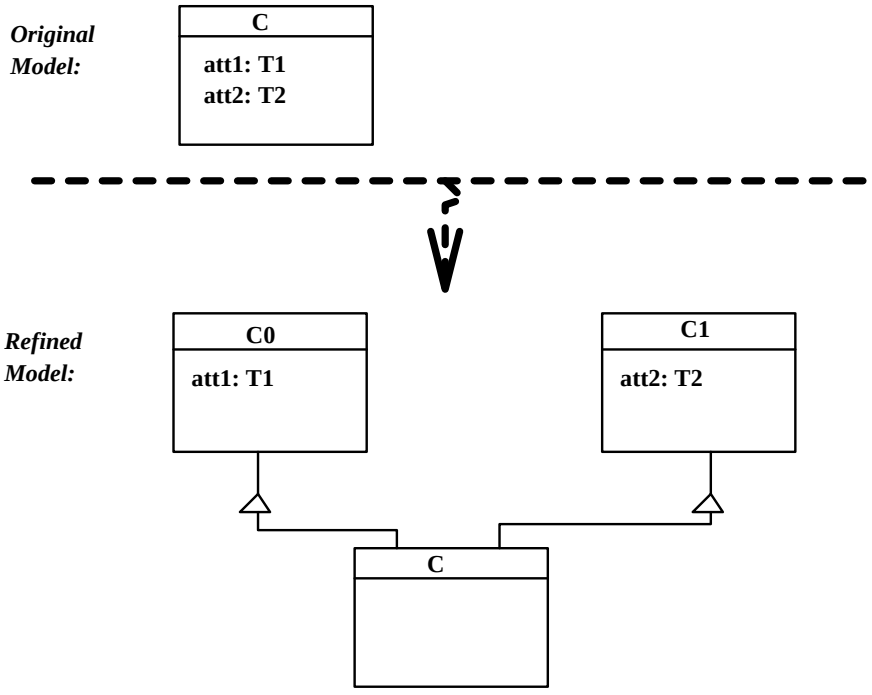

Figure 6: Factorisation of Class

This transformation is a refinement because the axioms for $att1$ and $att2$ in the initial model will be

satisfied by $att1'$ and $att2'$ defined from the second model, where:

$$\begin{aligned} att1' &= att1 \upharpoonright \overline{C} \\ att2' &= att2 \upharpoonright \overline{C} \end{aligned}$$

These definitions are valid because $\overline{C} \subseteq \overline{C0}$ and $\overline{C} \subseteq \overline{C1}$.

A further useful transformation is the elimination of an optional association between A and B (optional at the B end) by defining subclasses of A, one of which represents the domain of this association (Figure 7).

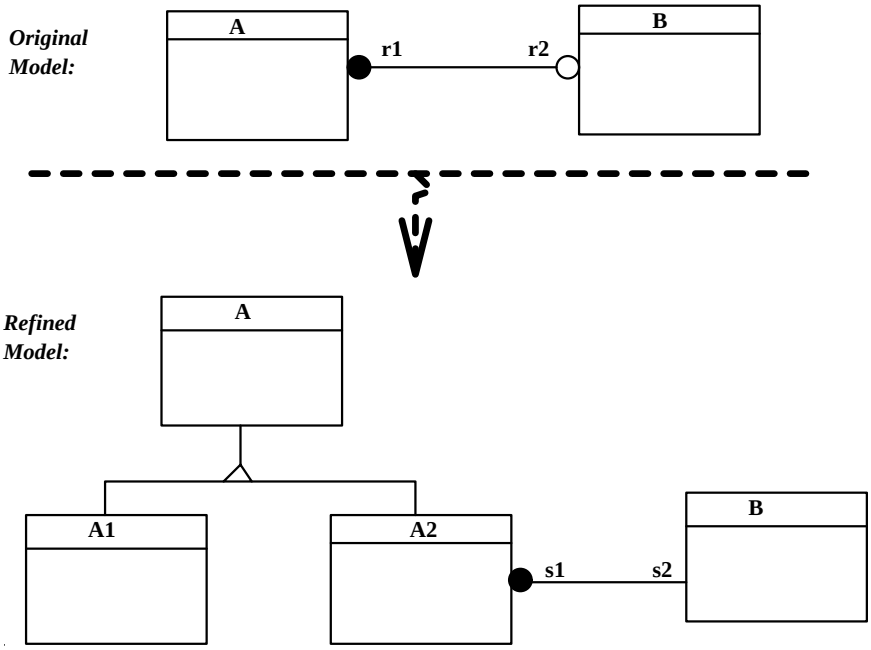

Figure 7: Elimination of Optional Association

This transformation is a refinement because the axiom

$$\begin{aligned} &\forall\, a : \overline{A};\ b : \overline{B} \cdot \\ &\quad b = r2(a) \wedge r2(a) \neq nil_B \ \equiv\ a \in r1(b) \end{aligned}$$

of the first model is also valid in the second, if we interpret $r1$ by $s1$ and $r2$ by

$$\begin{aligned} r2(a) \ = \ & \mathit{if}\ \ a \ \in \ \overline{A2} \\ & \mathit{then}\ \ s2(a) \\ & \mathit{else}\ \ nil_B \end{aligned}$$

because

$$\begin{aligned} b = r2(a) \wedge r2(a) \neq nil_B \ \Rightarrow \\ a \in \overline{A2} \ \Rightarrow \\ b = s2(a) \ \Rightarrow \\ a \in s1(b) \end{aligned}$$

by the axiom linking $s1$ and $s2$ in the refined model. Similarly for the other direction.

4 Subtyping of Statecharts

Only certain transformations on a statechart yield an extension of its theory. Chapter 8 of [4] gives a number of transformations on statecharts which the authors intend to represent subtypings of the corresponding classes. We discuss each of these in the following sections.

4.1 Expanding a State

Transforming a single state in the abstract model into a state with an enclosed statechart in the concrete model will produce a theory extension and hence a refinement.

This is the case since introducing nested states does not affect existing axioms, and adds new axioms for each new substate. State-transition axioms for transitions to the state are strengthened because they are now redirected to the designated initial substate of this state.

4.2 Weakening Transition Guards and Preconditions

This transformation (item 5 in the list of Chapter 8 of [4]) clearly does not yield a theory extension for essential models, since permission axioms would be weakened by it.

Intuitively the permission axioms do formalise the assertion that events cannot occur unless there is some transition available to record their occurrence. This, according to [4], is the essential model interpretation of a statechart. However the specification and implementation models interpret guards simply as preconditions in the usual sense of VDM [10] and B [13]: if the precondition fails to hold then the event concerned can still occur, but the response of the software to the event is not defined. In this case weakening transition guards and preconditions leads to stronger state-transition axioms and hence a refinement.

For concurrent systems Syntropy re-introduces the idea of guards as synchronisation constraints (Chapter 10 of [4]), but no graphical representation of these guards is given on the statechart models, only on class models. We can interpret these guards as permission constraints, and then require that subtyping of classes yields stronger constraints: so that any behaviour of a subtype object is the possible behaviour of some supertype object [15].

4.3 Splitting and Re-targeting Transitions

A transition t (but not the initialisation) can be *re-targeted* by changing its target from a state tt to a substate tt_i of tt in the new subtype model. tt must be an unstructured state in the original model, ie, we re-target t in combination with dividing tt into new substates. This restriction is not made explicit in [4], and is necessary because otherwise t could be redirected to a substate of tt disjoint from its original target, the designated initial substate of tt. Re-targeting results in a stronger theory for the new model, because the consequent of the state-transition axiom for t will become stronger. Initialisation, nesting and disjointness axioms are unaffected. Permission axioms are also unaffected, so that this transformation also yields a theory extension of class theories.

Re-targeting to a state which is not a substate of the original target is clearly not a subtyping under any of the definitions: it violates the state-transition axioms for the transition concerned.

A transition can be split either at its source or its target. Figure 8 shows a simple example of a source splitting of transition t. In a source splitting, the original guards, generations and postconditions must be repeated on each of the new transitions, and these can additionally define new generations and postconditions for particular cases. The guards cannot be changed. Every substate of ss must become a source for a split case of t.

The permission axioms for t are preserved by a source-splitting (and indeed, this is the case even if not every substate of the source state is maintained as a source of a split transition). Initialisation, nesting and disjointness axioms are unaffected, whilst state-transition axioms for t are strengthened because $obj \in \overline{ss}$ implies that $obj \in \overline{ss_i}$ for some substate; but then the new model implies that t from ss_i establishes all the

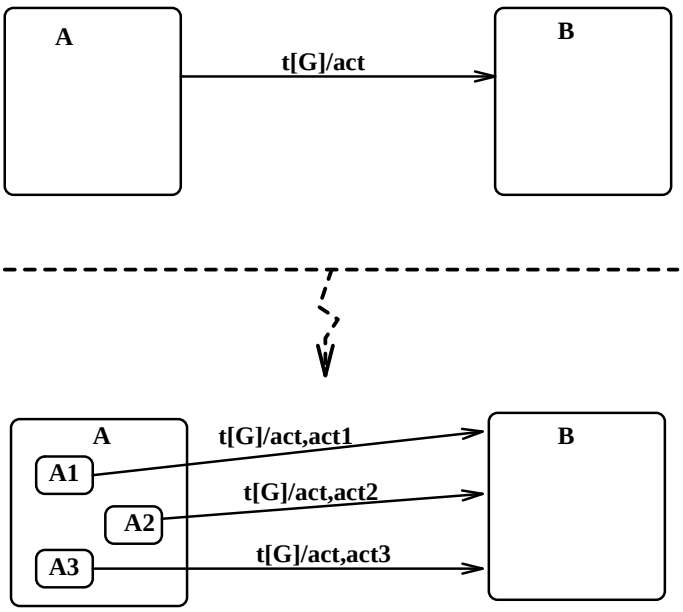

Figure 8: Source-splitting a Transition

generations and postconditions that t from ss did in the previous model. Obviously state-transition axioms would not be preserved if we allowed t to not be sourced from every substate of ss. Thus this transformation leads to a theory extension in each of the models.

A (non-initial) transition t can be split at its target state tt by breaking the original guard into several (disjoint and exhaustive) cases, assigning these to new transitions for the same event, and giving these targets within or at the original target. New generations and post-conditions can be added to each case (Figure 9). Again, tt must be unstructured in the original model.

Permission axioms will be preserved in this transformation because each individual split transition will have a stronger permission axiom than the original. State-transition axioms are preserved because the new target states and post-conditions are at least as constrained as for the original transition. In addition, if the precondition and guard of the original transition hold, then there must be some valid new transition whose guard and precondition hold, because these new conditions partition the old condition. Thus this transformation leads to a theory extension.

4.4 Strengthening Postconditions

This transformation clearly results in a theory extension in each of the semantic interpretations. Again, inconsistency may be introduced by this transformation, and so should be checked, as indicated in Chapter 8 of [4].

If new transitions for the same event are added in the subtype statechart, then the old postconditions for this event do not need to apply to these new transitions, provided that the new transitions have firing conditions disjoint from those of any previous case for this event (an example would be the introduction of transitions for exception cases). The filter predicate for the transition cannot be weakened.

4.5 Redefining Generations

Contrary to item 9 in Chapter 8 of [4], we cannot allow generated events to be arbitrarily redefined in a subtype. In terms of the subtyping definitions of [15] such a redefinition is invalid because then the behaviour of a subtype object may not be simulatable by any supertype object. Instead, new generations can be added

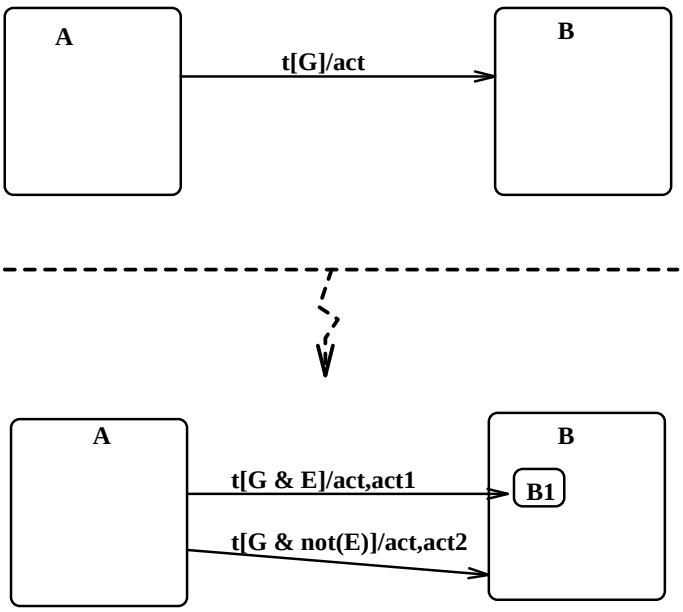

Figure 9: Target-splitting a Transition

to any transition, including the initialisation, provided they do not introduce inconsistency due to multiple event sends to the same receiver.

Likewise in the case of entry and exit generations (item 10): new generations can be added but old generations must be maintained, as must their order.

5 Transformations Involving a Change of Granularity

5.1 Introducing Internal Actions

A typical form of transformation from an essential to a specification model is to replace the simple assertion that an event α is followed by an event β (α being an input event to the software, β an output, although this distinction is not documented in the essential model), by a model of the software which shows how this reaction is produced (Figure 10). The transitions will typically be in three different state models.

The appropriate theory interpretation σ for this refinement is:

Symbol of abstract model	Symbol of refined model
α	α
β	β
$\overline{S}$	$\overline{S_1} \times \overline{C_1} \times \overline{T_1}$
$\overline{A}$	$\overline{S_2} \times \overline{C_1} \times \overline{T_1} \quad \cup \quad \overline{S_2} \times \overline{C_2} \times \overline{T_1}$
$\overline{T}$	$\overline{S_2} \times \overline{C_2} \times \overline{T_2}$

The disjointness axioms for the states of the abstract statechart therefore hold. Notice that membership of A is interpreted to mean "α has happened more recently than β". It covers the period of time when the internal reaction g to α is in progress.

The state-transition axioms for α and β are thus interpreted as:

$$self \in \overline{S_1} \times \overline{C_1} \times \overline{T_1} \ \wedge \ \alpha \ \Rightarrow \bigcirc(self \in \sigma(\overline{A}))$$

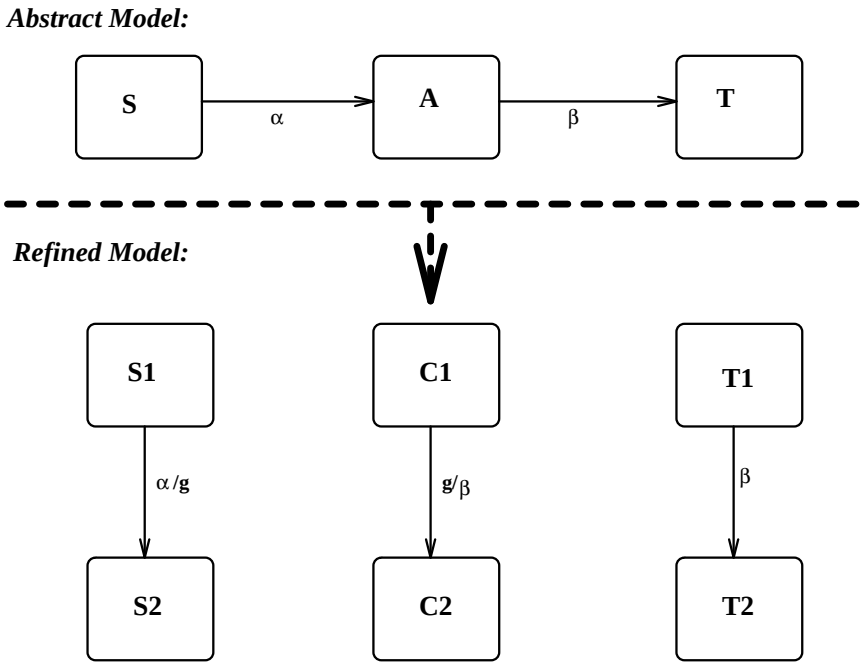

Figure 10: Transformation from Essential to Specification Model

and

$$self \in \sigma(\overline{A}) \ \wedge \ \beta \ \Rightarrow \ \bigcirc(self \in \overline{S_2} \times \overline{C_2} \times \overline{T_2})$$

The first holds because (if x is the first factor of *self*):

$$\alpha \wedge x \in \overline{S_1} \ \Rightarrow \ \bigcirc(x \in \overline{S_2})$$

in the concrete model, and no other change of state occurs in the statecharts for the other factors of *self*. Thus *self* enters the first subset of elements of $\sigma(\overline{A})$.

The second holds because β can only occur if g has more recently occurred than α or any other transition for an external event, by the axiom

$$\alpha \ \Rightarrow \ \bigcirc(\neg\,(\alpha \vee \beta)\ \mathcal{U}\ g)$$

of the generation part of α.

Thus if $self \in \sigma(\overline{A})$, it must be in the second subset of elements of this set, ie, the third factor of *self* is in $\overline{T_1}$ and the second factor is in $\overline{C_2}$, so that by the effect axiom for β in the refined model, the result follows.

Permission axioms

$$\alpha \Rightarrow (self \in \overline{S})$$

and axioms asserting that an event A happens iff one of its transitions happen:

$$A \ \equiv \ \alpha \ \vee \ \ldots$$

of the essential model should become preconditions in the specification model:

$$A \wedge self \in \sigma\overline{S} \ \Rightarrow \ \alpha$$

"If A occurs in state S, the transition labelled α must occur."

A more complex form of this transformation can be carried out within the specification model, by replacing a single internal reaction step by a sequence of two or more steps (Figure 11).

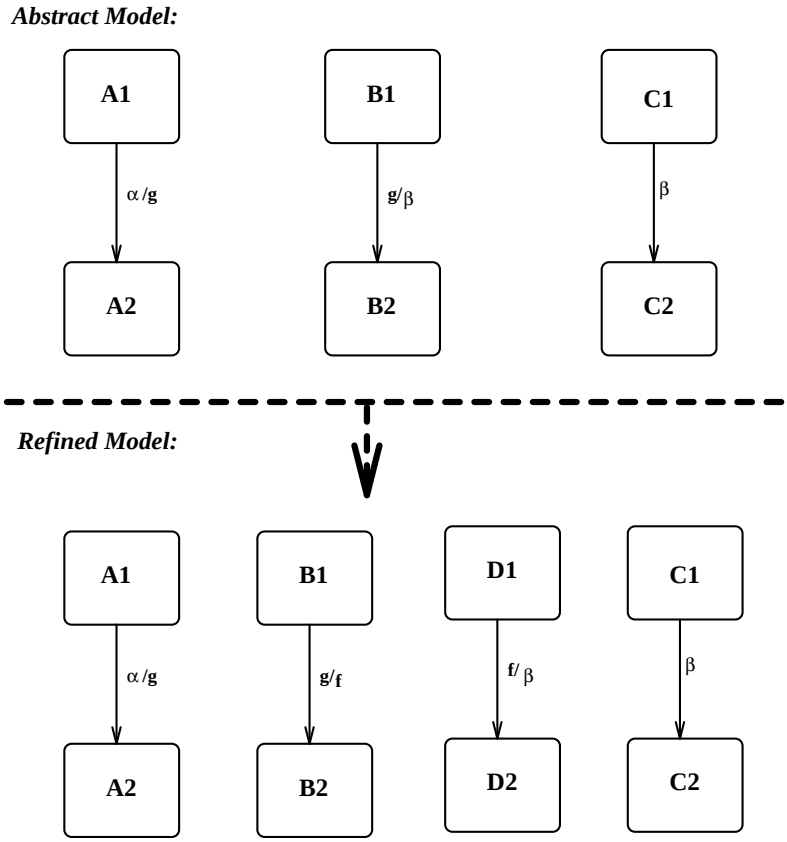

Figure 11: Factoring a Response in Specification Model

The refinement approach used in the first case can also be used here. We additionally need that the interpretation of the axiom

$$g \Rightarrow \bigcirc(\neg\, \alpha\, \mathcal{U}\ \beta)$$

of the abstract model is true in the refined. This follows because we have

$$g \Rightarrow \bigcirc(\neg\, (\alpha \vee \beta)\, \mathcal{U}\ f)$$

and

$$f \Rightarrow \bigcirc(\neg\, \alpha\, \mathcal{U}\ \beta)$$

in the refined model.

5.2 Introducing Control Flow

In this section we cover the replacement of an abstract transition by a procedural combination of concrete transitions. This is used for a number of purposes:

- Forwarding messages from one object to another, in order to make an event available to all the objects which are specified to respond to it in the abstract model;
- sending messages to other objects to ensure that specification model invariants which constrain these objects are preserved by the current transition and local effects;
- replacing complex navigation expressions by a sequence of method calls which compute their values;
- introducing intermediate results (eg, new temporary variables being used to swap the values of two other variables), etc.

Decomposition into conditionals does not change the granularity of actions and is covered by the target-splitting transformations of [12].

5.2.1 Forwarding

A standard form of refinement in Syntropy is to replace broadcast events (in a specification model) by point-point message sending in an implementation model (Figure 12).

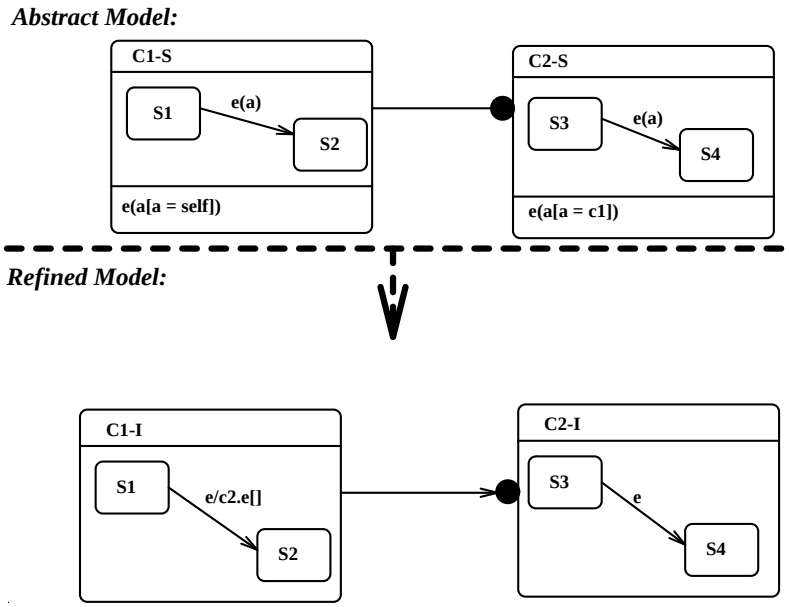

Figure 12: Forwarding Transformation

This transformation is valid *provided* that there is no interference during the execution of the forwarding process which corresponds to the abstract $e(a)$ transition.

The abstract specification gives the following semantics to an occurrence of $e(a)$:

$$a \in \overline{S_1} \wedge e(a) \;\Rightarrow\; \bigcirc(a \in \overline{S_2})$$
$$\forall\, y \in \overline{S_3} \cdot y.c_1 = a \;\Rightarrow\; \bigcirc(y \in \overline{S_4})$$

All of these changes occur in the same conceptual execution interval.

At the concrete level however we must explicitly require that the forwarded messages to objects of C_2 are executed within the "secured" part of the execution of $a.e$, where a is the target object in C_1.

Otherwise, a could respond to $a.e$, send e to an associated C_2 object b in the relaxed part of the transition for e, and enter a new state, different to S_2, by the time that this forwarded event is responded to by b.

If the messages are sent in the secured part, then the concrete e transition in C_1 completes with all the generated transitions of associated C_2 objects, resulting in a state which corresponds (as far as the pictured subsystem is concerned) with the post-state of the abstract e event.

5.2.2 Sequential Decomposition

Refinement steps from the specification to the implementation model, and within the implementation model, can involve the replacement of a single transition with a suitable composition of transitions in the same state model (in contrast with the transformations of the previous section). Automatic transitions (not triggered by an event) may be used, or a transition may generate its successor. The simplist case is that of sequential decomposition (Figure 13).

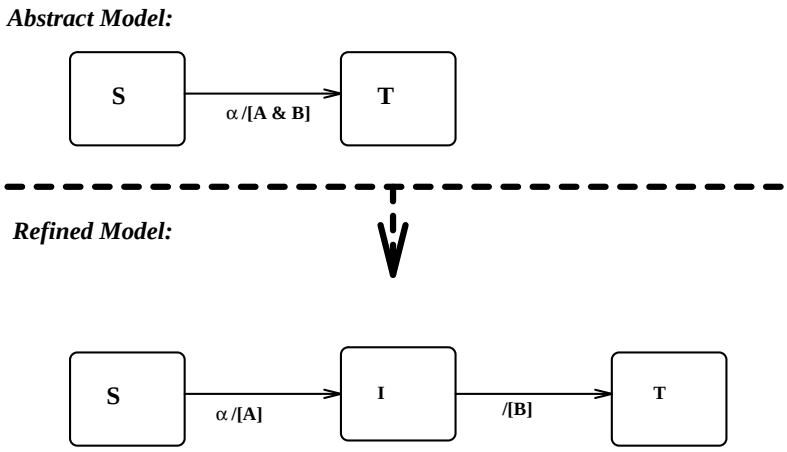

Figure 13: Sequential Decomposition

A new intermediate state I is introduced, and an automatic transition τ leading from this state to the target state. Part A of the original postcondition $A \wedge B$ is achieved by the refined transition for α, and part B is achieved by τ (which is enabled as soon as I is entered). The axiom for τ is:

$$(1) \quad self \in \overline{I} \;\Rightarrow\; \neg\,(\alpha \vee \ldots)\; \mathcal{U}\; \tau$$

where all other transitions of the statechart are excluded from occurring until τ has occurred. A stronger version of this axiom is

$$self \in \overline{I} \;\Rightarrow\; \tau$$

Both are sufficient to prove the refinement.

The interpretation of the effect axiom for α is:

$$\forall\, x : X \cdot x = v \wedge self \in \overline{S} \wedge \alpha \;\Rightarrow\; \bigcirc_{Abs}(self \in \overline{T} \wedge A[x/v, v/v'] \wedge B[x/v, v/v'])$$

where $v : X$ are the state variables (assumed to be interpreted by themselves in the refinement).

The interpretation $\bigcirc_{Abs} P$ for a predicate P means "at the next initiation time of an action m of Abs, P holds". But such an m can only occur from the source state T, because no transition of the original model has source I in the refined model. We could alternatively (and equivalently) interpret $\bigcirc_{Abs} P$ as "in the next stable state P holds".

At such a time point after an occurrence of α, with $self \in \overline{S}$, τ must have occurred, and no other transition can have occurred, because of (1) above. Thus, if the effects of α and τ in the refined model do not interfere, the result follows.

An alternative (valid for specification models) would be to introduce a new named transition g instead of τ, and to add the generation of g to the concrete transition α.

A more general version of this transformation breaks the original transition into two interfering transitions whose cumulative effect establishes the original postcondition. This requires that

$$\forall\, x : T \cdot P_1[x/v'] \wedge P_2[x/v] \;\Rightarrow\; P$$

where P_1 is the postcondition of the first transition, and P_2 the postcondition of τ.

The interpretation of states is that $\overline{S}$ maps to $\overline{S} \cup \overline{I}$, and $\overline{T}$ maps to itself. This is valid in the case of automatic transitions because the abstract state is assumed not to be changed by the concrete α transition or by membership of I [4].

In the case of the specification model, abstract states are interpreted by themselves, and states such as I correspond to elements of the class C of the statechart which are not in any of the abstract states.

5.2.3 Iterative Decomposition

Introducing a loop structure through automatic transitions (at the implementation model level) or generated transitions (for specification models) also changes the granularity of actions. Figure 14 shows the situation for implementation models. The proof is closely related to the usual verification process for loops using weakest preconditions. A predicate I must be found such that

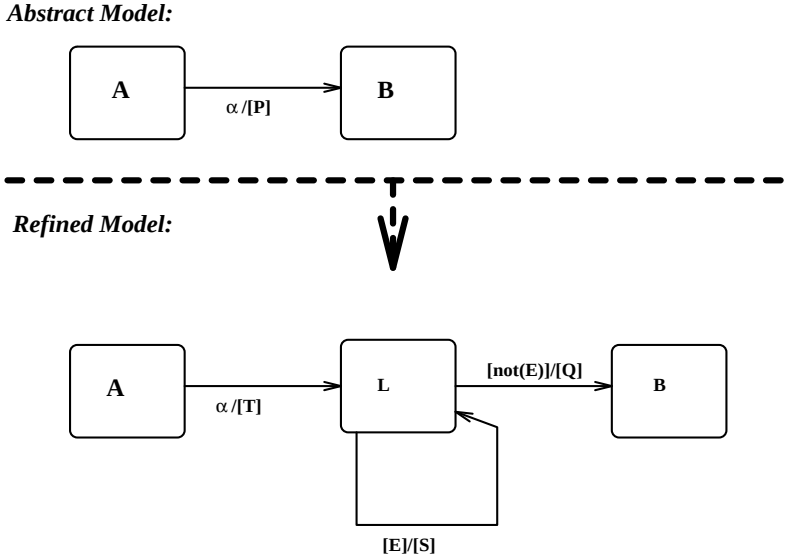

Figure 14: Iterative Decomposition

$$\alpha \wedge \mathit{self} \in \overline{A} \;\Rightarrow\; \bigcirc I$$

and

$$\tau_1 \wedge I \wedge E \wedge \mathit{self} \in \overline{L} \;\Rightarrow\; \bigcirc(I \wedge \mathit{self} \in \overline{L})$$

and

$$\tau_2 \wedge \neg\, E \wedge \mathit{self} \in \overline{L} \wedge I \;\Rightarrow\; \bigcirc(P \wedge \mathit{self} \in \overline{B})$$

hold in the refined model, where τ_1 and τ_2 are the two new automatic transitions introduced in the refined model.

In Syntropy the attributes of an object are assumed to be unchanged by automatic transitions or throughout membership of 'unstable' states (states with at least one automatic transition leading away from them). Thus we can simplify the requirements for correctness to be:

1. $T \Rightarrow I$
2. $E \wedge S \wedge I \;\Rightarrow\; I'$ where we assume that only local variables acquire a $'$ in I'
3. $not(E) \wedge I \wedge Q \;\Rightarrow\; P$

As for sequential decomposition, we interpret the abstract $\bigcirc$ by $\bigcirc_{Abs}$, so that P is only required to hold at the next initiation time of some transition m from the abstract model after α executes. Again, such a situation implies that a series of iterations of the loop has taken place followed by a single execution of τ_2. By the choice of I, this establishes the required postcondition P at this time.

An example is given in [4] for the $clearTo(m)$ operation of $AlarmStore$. In this case the abstract transition has postcondition P as

$$queue' \;=\; clear\_to(m, queue)$$

where this function removes all elements from the front of $queue$ before the first occurrence of m.

A local variable $tmpQ$ is introduced to carry out this iteration. The state $Searching$ plays the role of L, postcondition T is

$$tmpQ' = queue$$

the invariant I is

$$\exists\, t : seq(Minder) \cdot queue = t \frown tmpQ \;\wedge\; m \notin \mathrm{ran}(t)$$

Condition E is

$$\#tmpQ > 0 \;\wedge\; m \neq first\ tmpQ$$

S is

$$tmpQ' = tail\ tmpQ$$

and Q is

$$queue' = tmpQ$$

It can be easily checked that the conditions (1), (2) and (3) above hold in this case.

5.3 Replacing Broadcast Communication by Message Passing

In the transition to the implementation model, broadcast communication is replaced by point-to-point message passing. In the usual case that there is a single target object of a broadcast message, this is a trivial refinement: a generated event $m(x)$ where b is the only target object that satisfies the filter $F(b, x)$ for this event at a particular time, can be replaced by an invocation $b.m(x)$. Both have the same semantic representation in our formalisation of Syntropy.

In the case that a collection $\{c : @C \mid F(c, x)\}$ of objects passes the filter for m we can replace the generation by an unordered iteration of the individual executions:

```
forall c ∈ { c : @C | F(c, x) }
do
  c.m(x)
```

This unordered iteration executes over an interval $[t_1, t_2]$ iff exactly one execution of each action $c.m(x)$ for distinct choices of c executes within this interval.

The concrete interval $[t_1, t_2]$ therefore corresponds to the abstract interval which observes the execution of $m(x)$ ('simultaneously' on all the objects of C that pass the filter).

Conclusion

We have given a theoretical basis for showing which transformations of Syntropy models lead to refinements. Examples of refinement transformations have been given. We believe that such a set of transformations, if supported by suitable tools, could provide a rigorous development technique which could be used by software engineers in practice. A related project at Imperial College, Lisbon University and the University of Rio de Janeiro is currently undertaking such tool development.

References

[1] M Awad, J Kuusela, and J Ziegler. *Object-oriented Technology for Real-time Systems.* Prentice Hall, 1996.

[2] J C Bicarregui, K Lano and T Maibaum. *Objects, Associations and Subsystems: a hierarchical approach to encapsulation.* ECOOP 97, LNCS, 1997.

[3] Towards a Compositional Interpretation of Object Diagrams. J.C. Bicarregui, K.C. Lano and T.S.E. Maibaum. To appear: Proc. of IFIP TC2 Working Conference on Algorithmic Languages and Calculi, Strasbourg, February, 1997.

[4] Cook S., Daniels J., *Designing Object Systems*, Prentice Hall, 1994.

[5] Steve Cook and John Daniels. *Syntropy Case Study: The Petrol Station*, Object Designers Ltd., 1996.

[6] J Fiadeiro and T Maibaum. *Describing, Structuring and Implementing Objects*, in de Bakker *et al.*, *Foundations of Object Oriented languages*, LNCS 489, Springer-Verlag, 1991.

[7] J. Fiadeiro and T. Maibaum *Temporal Theories and Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing, Vol.4, No. 3, pp. 239-272, 1992. Springer Verlag.

[8] Goguen, J. and Burstall, R. Introducing Institutions. In Clarke and Kozen, eds. Logics of Programs, pp. 221-256, Springer-Verlag, 1984.

[9] D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Sci. Comput. Prog. **8** pp. 231-274 (1987).

[10] Jones C. B., *Systematic Software Construction using VDM (2nd Edition).* Prentice Hall, 1990.

[11] L. Lamport, The Temporal Logic of Actions, Digital Technical Report 79, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301. December 25th, 1991.

[12] Lano K., *Semantic Frameworks for Syntropy*, BIRO Project Document GR/K67311-1, Imperial College, Feb. 1996.

[13] Lano K., Haughton H., *Formal Development in B AMN*, Information and Software Technology, Vol. 37, No. 5–6, 1995, 303–316.

[14] Lano K., *Formal Object-oriented Development*, Springer-Verlag, 1995.

[15] Liskov B., Data abstraction and hierarchy. In *OOPSLA '87 Conference Proceedings*, 1987.

[16] Liskov B., Wing J., *Specifications and their use in Defining Subtypes*, ZUM '95 Proceedings, Springer-Verlag LNCS Vol. 967, 1995.

[17] Rumbaugh, J. et al. Object-Oriented Modelling and Design, Prentice-Hall, Englewoods Cliffs, New jersey, 1991.

[18] Rational Co., *UML Version 1.0*, `http://www.rational.com`, 1997.

[19] M. Spivey, The Z Notation: a reference manual, Prentice-Hall, 1992.

[20] Wieringa R., de Jonge W., Spruit P., *Roles and Dynamic Subclasses: A Model Logic Approach*, IS-CORE report, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.