Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation

J.C. Bicarregui, K.C. Lano, T.S.E. Maibaum

Imperial College, London

Abstract. We describe a compositional approach to the formal interpretation of type view diagrams and statecharts. We define theories for object instances and classes, and theories for associations between them. These theories are combined with categorical constructions to yield a formalisation of the entire system.

We observe that some notations require the identification of theories intermediate between the theories of the constituent classes and associations and that of the entire system. This leads us to propose a notion of subsystem which generalises the concept of object and yields an approach to system specification employing object-like encapsulation in a nested hierarchy of components[1].

1 Introduction

The combination of data encapsulation through the aggregation of related attributes, and instance identity to distinguish separate occurrences of that encapsulated data is a powerful feature of the object oriented paradigm. Objects provide a convenient way to group related attributes (eg: a table has a length, a width and a height), Object Identifiers (OIds) give us the ability to model the distinct existence of two or more instances with exactly the same characteristics (two tables may have the same attributes but still be different tables).

The indirection implicit in the use of OIds is then exploited to distinguish attributes which are themselves objects from attributes which are (pure) values. We will use the terms "reference attributes" and "value attributes" for these respectively. An example of the use of reference attributes is in structural decomposition of an object. For example if a table has four legs and a top, then a particular table object, *table23*, may comprise of four particular leg objects, *leg12*, *leg13*, *leg14*, *leg15*, and a top, *top27*. Examples of pure attributes are the height, length and width of the table given above.

Properties of objects can relate both pure attributes and reference attributes indiscriminately. For example a long table might be one that is more than twice as long as it is wide, a level table must have the four legs of the same length, and the height of the table is the length of the legs plus the depth of the top, etc.

[1] This work is being undertaken by the UK EPSRC project "Formal Underpinnings of Object Technology".

In most Object Oriented Analysis and Design notations value attributes are considered to be part of the object itself, whereas reference attributes are given via associations between objects. Associations are therefore a separate modelling tool in the Object Oriented designers toolbox. Opinions differ as to whether associations should themselves be objects with their own object identities, whether associations are a separate primitive concept which should have identities (AIds), or whether associations should be *pure* constructions without identities.

Associations-as-objects yield an economy of form and a uniform approach to modelling but may be confusing two separate purposes. Associations with AIds enable the distinction of two copies of the same association between the same objects. Pure associations enable a value-based approach to equality and do not require the overhead of indirection in construction.

More generally, the same issues arise when we collect objects together in larger aggregations. The concept of "subsystem" can be adopted as a means to provide coarse grained modularity in OO design. Subsystems can be defined using the class-instance approach as for objects, they can be managed by coordinating objects and they can be units of encapsulation in the same way as objects. Subsystems can provide a structure in which properties of collections of objects can be defined at the appropriate level without complete globalisation (as in Fusion operation schemas [4]) or over localisation (as in Syntropy statecharts [5]). If subsystems are considered to be first class objects, then they yield the possibility of developing a nested hierarchy of levels of granularity and hence a compositional approach to vertical structuring.

An open question is whether there is merit in attributing identifiers to instances of subsystems (SIds) in the way that Oids are given to object instances. In this paper we adopt the position that a hierarchical approach to structuring designs is essential if large designs are not to be subject to an exponential increase in complexity. We therefore separate the concerns of aggregation (as embodied in objects and associations) and instance identity (as embodied in OIds) and formally define the concept of subsystem as a first class construction and argue that it is a generalisation of the familiar concept of both object and association.

The ideas presented here have arisen out of the formalisation of the "Syntropy" approach to object oriented analysis and design which has shown that many existing notations, although sometimes attributed to object classes, need to be formalised at a level in between objects and systems. In structuring the formal interpretation of system diagrams to formalise these notations, it becomes clear that their representation as part of an object class is inappropriate and that their interpretation at a higher level is more justified.

Two examples we will examine in this paper are the **RadioButton** example of [5], and an example of multiple constraints between associations. The radio button example of [5] highlights the lack of clarity which can arise as a result of enforced local (to objects) behaviour specification. The Syntropy statechart in this case is shown in Figure 1. The **true** filter in the event list indicates that *every* existing object of **RadioButton** may react to the **turn_on** event. The way

in which they react depends upon whether they are the button being pressed ($\mathbf{x} = \mathbf{self}$) and what their current state is. It is not immediately evident from

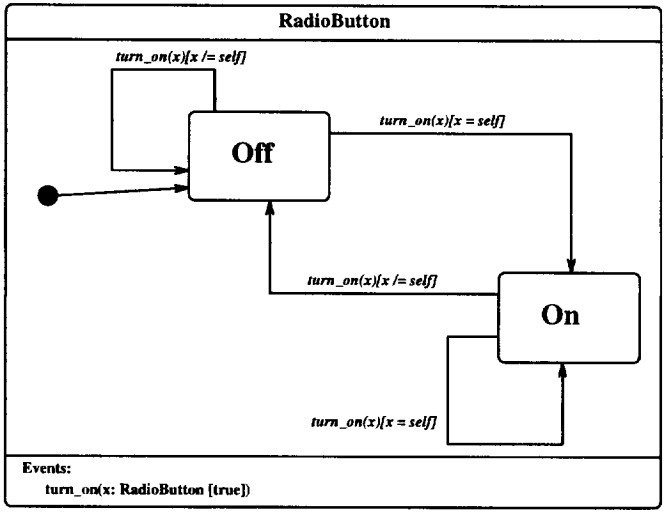


Fig. 1. Statechart of Radio Button

this diagram that in fact, after a **turn_on(x)** event, that *only* **x** will be in the **On** state, and all other existing radio buttons will be in the **Off** state. Nor is it clear that actually the final state of an object depends only on whether it is **x**, and not on its starting state. We examine more abstract and design-free versions of this specification in Section 3.

Similarly, because Syntropy, OMT or Fusion defines no level of structuring between individual classes and entire systems (or domains), constraints between associations and between attributes of different classes are expressed via navigation expressions and invariants which are local to particular classes. Such localisation may be arbitrary (the expressions and invariants could just as well be written in other classes) and may therefore reduce the clarity and abstraction of the specifications concerned.

For example, consider Figure 2. Here there are two alternative ways of expressing the constraint that every student taught by a college is either an external student or lives at the college. As an invariant of **College** we could write:

$$\mathbf{teaches} \subseteq \mathbf{accomodates} \cup \mathbf{external\_students}$$

As an invariant of **Student** we could alternatively express it as:

$$\mathbf{external\_at} = \mathbf{taught\_at} \vee \mathbf{lives\_at} = \mathbf{taught\_at}$$

Subset constraints could alternatively be used between associations. Again, we will give a more abstract presentation of this situation using subsystems in Section 3.

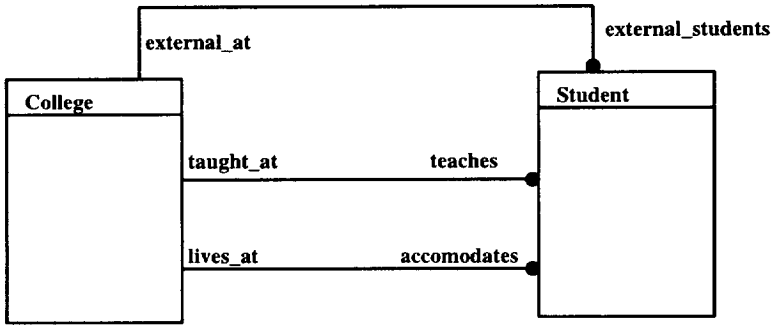


Fig. 2. Subsystem Example

Overview

In Section 2, the major part of this work, we describe a hierarchical formalisation of the Syntropy diagrammatic notations where each diagrammatic component is interpreted separately and the system description is built in a compositional way from these separate interpretations. We observe how many of the constructions are naturally interpreted in theories which correspond to identified parts of the overall system. In Section 3 we propose the notion of subsystem as a "first class object" which generalises the concept of object and yields a hierarchical approach to encapsulation where subsystems are nested one inside the other. Section 4 concludes with a summary of the achievements so far and discusses further work required.

2 Formalising Syntropy

2.1 Syntropy

Syntropy [5] is a methodology for object-oriented analysis and design similar to OMT [14] with additional formal specification elements derived from Z[15]. It represents a significant advance over previous object-oriented methods in giving mathematical specifications of data models and dynamic behaviour.

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 [10] 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 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.

2.2 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. [12]) and axioms are specified using temporal logic operators including: $\bigcirc$ (in the next state), $\bullet$ (in the previous state), $\mathcal{U}$ (strong until), $\mathcal{S}$ (strong since), $\Box$ (always in the future), $\blacksquare$ (always in the past), $\diamond$ (sometime in the future) and $\blacklozenge$ (sometime in the past). The predicate **BEG** is true exactly at the first moment. For the purposes of this paper only the "next" temporal operator will be required.

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

In the style of [9], 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

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

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

2.3 Interpreting Object Types

Figure 3 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 $\mathbf{T}_1$ and $\mathbf{T}_2$ respectively[2].

[2] Note object-typed attributes are given via associations, see Section 2.4.

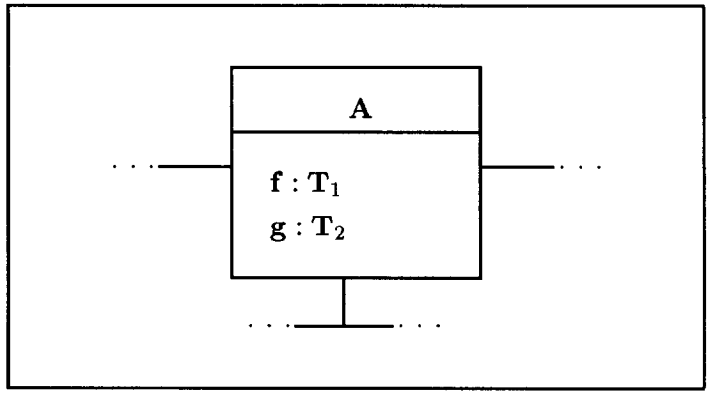


Fig. 3. Part of a type view diagram

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

The signature of a generic instance We define a theory, $\mathbf{A_i}$ 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.

$\mathcal{S} = \{\mathbf{T}_1, \mathbf{T}_2\}$
$\mathcal{A} = \{\mathbf{f} : \mathbf{T}_1, \mathbf{g} : \mathbf{T}_2\}$
$\mathcal{G} = \{\ldots\}$

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 [8], we interpret self using **A**-morphisms which add the object identifier as an extra parameter when attributes and actions are globalised (see Section 2.3).

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, $\overline{\mathbf{X}}$, to record the finite set of currently existing instances. In terms of [16], @**C** is $\mathbf{ext}(\mathbf{C})$ and the value of $\overline{\mathbf{C}}$ at time π is $\mathbf{ext}_\pi(\mathbf{C})$. There are actions of **M** to create and kill objects of **X**.

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

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[3] (*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:

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

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*)

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

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 4).

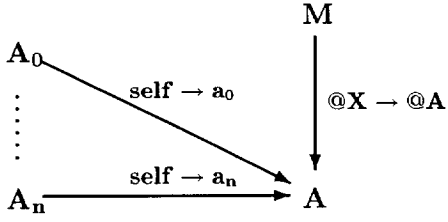


Fig. 4. Instance and class manager theories are embedded in the theory of the class

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 [8][4]. This is equivalent to defining **self** as a constant in the

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

[4] A morphism, σ, of object signatures from $\theta_1 = (\Sigma_1, \mathbf{A}_1, \Gamma_1)$ to $\theta_2 = (\Sigma_2, \mathbf{A}_2, \Gamma_2)$ is a triple comprising: a morphism of algebraic signatures $\sigma_\nu : \Sigma_1 \to \Sigma_2$; for each $\mathbf{f} : \mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}$ in $\mathbf{A}_1$, an attribute symbol $\sigma_\alpha(\mathbf{f}) : \sigma_\nu(\mathbf{s}_1), \ldots, \sigma_\nu(\mathbf{s}_n) \to \sigma_\nu(\mathbf{s})$ in $\mathbf{A}_2$; and, for each $\mathbf{g} : \mathbf{s}_1, \ldots, \mathbf{s}_n$ in Γ_1, an action symbol $\sigma_\gamma(\mathbf{g}) : \sigma_\nu(\mathbf{s}_1), \ldots, \sigma_\nu(\mathbf{s}_n)$ in Γ_2.

Given a signature morphism, the translation of formulae is defined according to their structure in the usual way, and given two object descriptions, (θ_1, Φ_1) and (θ_2, Φ_2), a morphism, $(\theta_1, \Phi_1) \to (\theta_2, \Phi_2)$, is a signature morphism which preserves

instance theory which acts as a (dummy) placeholder for later identification with the object identifiers in the class theory.

The resultant theory, **A**, has an attribute **att(a)** for each attribute **att** of each existing instance **a**. For example, for instance $\mathbf{a_i}$ and attribute **f**, there is an attribute $\sigma_i(\mathbf{f})$ in the class theory. In effect **f** is a (finite) partial function from **@A** to $\mathbf{T}_1$. We define a syntactic sugar which names the $\sigma_i(\mathbf{f})$ conveniently

$\mathbf{f} : @\mathbf{A} \rightarrow \mathbf{T}_1$

$\mathbf{a_i}.\mathbf{f} = \sigma_i(\mathbf{f})$

So, in **A**, **f** is a partial function from **@A** to $\mathbf{T}_1$ which is written in the right. A similar approach is taken to the naming of instance actions.

History and the state Note that we have, up to this point, avoided the use of any temporal operator other than "next". This is because all behaviour determining history has been explicitly stored in the attributes. However, for example, we may wish to require that it is not possible for an object to be "reborn". This can be given using temporal operators or can be given in the above style by augmenting the state with a "memory" of past objects. $\overline{\mathbf{X}}$ would then distinguish between objects that have lived and those which have not.

$\overline{\mathbf{X}} : @\mathbf{X} \rightarrow \{$ unborn, alive, dead $\}$

Axioms would chart the evolution of objects from unborn, through alive, to dead.

2.4 Interpreting Associations

We now formalise the notion of an association as depicted in Figure 5. We will interpret the association without any knowledge of the structure of the objects it associates[5]. Thus we have a generic theory of associations. We then use a renamed copy of this theory for each particular association in the model.

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.

---

validity and locality, ie. for which we have: $\Phi_2 \Rightarrow_{\theta_2} \sigma(\mathbf{p})$ is valid for each valid $\mathbf{p} \in \theta_1$; and $\Phi_2 \Rightarrow_{\theta_2} (\theta_1 \rightarrow_\sigma \theta_2)$, where $\Rightarrow_{\theta_2}$ is entailment in θ_2, and $(\theta_1 \rightarrow_\sigma \theta_2)$ is the θ_2 formula which is the translation of the locality axiom of θ_1.

Given a sort **A**, and a morphism of object signatures, σ, the **A**-morphism, $\sigma_{\mathbf{A}}$, is the same as σ except that it adds an extra parameter of sort **A** to each attribute and action symbol. Thus for example, for given $\mathbf{a} : \mathbf{A}$, for each attribute symbol **f** and $\mathbf{e_i} : \mathbf{s_i}$, $\sigma_{\mathbf{A}}(\mathbf{f}(\mathbf{e_1}, \ldots, \mathbf{e_n})) = \sigma_{\mathbf{A}}(\mathbf{f})(\mathbf{a}, \sigma_{\mathbf{A}}(\mathbf{e_1}), \ldots, \sigma_{\mathbf{A}}(\mathbf{e_n}))$.

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

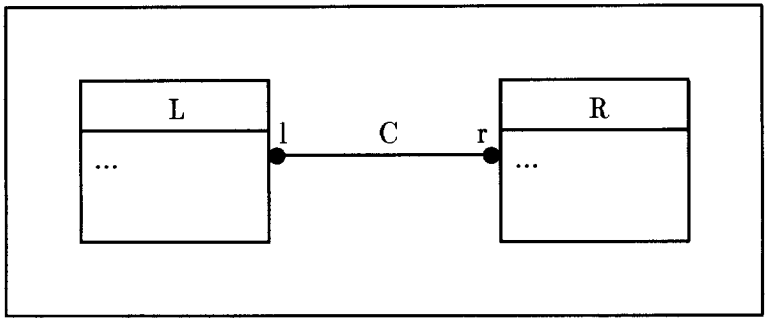


Fig. 5. A simple association

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**[6]. Note that **lr** plays the same role as $\overline{\mathbf{X}}$, it is the set of existing links in the association. The theory signature is:

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

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

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

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

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

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

In this case, as there are no identifiers for links, we do not require no-rebirth.

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

$\mathrm{BEG} \Rightarrow \mathbf{lr} = \varnothing$

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**.

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

Figure 6 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.

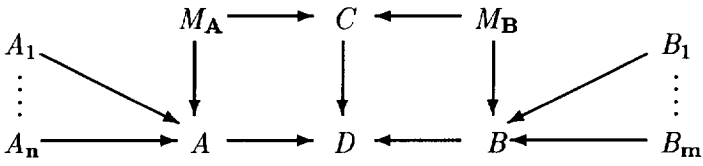


Fig. 6. The type view diagram is interpreted as the colimit of the object and association theories.

We now add axioms to **D** or to **C** which interpret the particular kind of association required.

Cardinality Constraints. Firstly, 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.

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

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**

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

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

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

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

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

$\forall \mathbf{a}, \mathbf{b}_1, \mathbf{b}_2 \cdot (\mathbf{a}, \mathbf{b}_1) \in \mathbf{ab} \wedge (\mathbf{a}, \mathbf{b}_1) \in \mathbf{ab} \Rightarrow \mathbf{b}_1 = \mathbf{b}_2$

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

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

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

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 $\overline{\mathbf{A}}$ is associated with exactly one $\overline{\mathbf{B}}$, then the map is total on $\overline{\mathbf{A}}$. Again this is a constraint between association and class manager theories

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

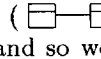
Note that we do not require the map to be surjective since an @**B** can be associated with an empty set of @**A**s.

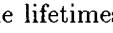
Again this can be ensured by conditions relating **create** and **kill** from the class manager for **A** with the relation **ab** from the association theory

$\mathbf{a}.\mathbf{create}_{\mathbf{A}} \Leftrightarrow \mathbf{a} \notin \mathrm{dom}\,\mathbf{ab} \wedge \mathbf{a} \in \mathrm{dom}\bigcirc\mathbf{ab}$

$\mathbf{a}.\mathbf{kill}_{\mathbf{A}} \Leftrightarrow \mathbf{a} \in \mathrm{dom}\,\mathbf{ab} \wedge \mathbf{a} \notin \mathrm{dom}\bigcirc\mathbf{ab}$

Conversely, for the one-many case, we have conditions on $\mathbf{ab}^{-1}$.

One-one associations () In the one-one case, we simply have both of these sets of axioms and so we can conclude that **A**s and **B**s must be created and deleted in lock-step and therefore that there are always the same number of $\overline{\mathbf{A}}$s as $\overline{\mathbf{B}}$s.

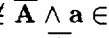
Lifetime Constraints () A "diamond" on the association is a constraint concerning the lifetimes of the associated objects. Diamonds can be interpreted independently of multiplicities.

A diamond at the right hand end of the association, indicates that **A**s can only exist if linked with some (set of) **B**s and that this set must be constant throughout the **A**'s lifetime[7]. This is ensured if **A**s can be linked to **B**s only when they are created, and unlinked only when deleted

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

$\mathbf{unlink}(\mathbf{a},\mathbf{b}) \Rightarrow \mathbf{a} \in \overline{\mathbf{A}} \wedge \mathbf{a} \notin \bigcirc\overline{\mathbf{A}}$

There are similar rules for a diamond on the left of an association.

Subtypes () In the semantics, we interpret subtyping as a particular form of association with particular cardinality and lifetime constraints and where the object identifiers are drawn from the same set of tokens. This interpretation can then be used to show the validity of all the subtyping transformations of Chapter 8 of [5] with the exception of target splitting[13] in case that the target state is already nested. From the perspective of system structuring, however, which is the focus of this work, details of this are not relevant but can be found in [3].

We have seen how the interpretation of components of a type view diagram can be interpreted in a hierarchy of theories each corresponding to the separate diagram elements. We note that the formalisation of some particular constraints available in the diagrammatic notation are interpreted in the theories resulting from the composition of the theories of the separate diagram elements. This trend will be continued in the interpretation of statecharts which follows and will lead us to identify the concept of subsystem described in Section 3. In general the set of attributes of a subsystem theory will be the union of the sets of attributes of its constituent class and association theories, and similarly for actions.

[7] It is not clear whether Syntropy requires the set associated in an aggregation to be non-empty. We assume it is not.

2.5 Interpreting Statecharts

Statecharts are the most complex and semantically rich notation employed by Syntropy. Based on [10], they depict the state space of an object, partitioned according to "those states which distinguish the possible orderings of events" ([5], p.91). Statecharts have distinct interpretations at the essential, specification and implementation modelling levels. We focus on the essential level, but many of the semantic interpretations also apply to the specification and implementation levels[8].

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.

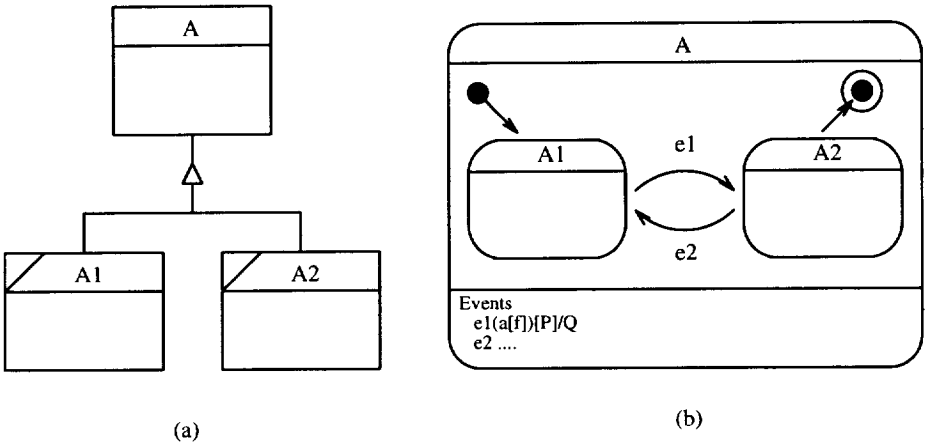


Fig. 7. State types and statechart for class **A**

For example, Figure 7 depicts a class with two state subtypes. $\mathbf{A}_1$ and $\mathbf{A}_2$. The subtypes in Figure 7(a) correspond to the states in Figure 7(b). The arrow from the solid blob indicates that the object is created in state $\mathbf{A}_1$ and the arrow to the ringed blob represents object deletion from state $\mathbf{A}_2$. The other arrows indicate state transitions $\mathbf{e}_1$ and $\mathbf{e}_2$ which take the object from state $\mathbf{A}_1$ to state $\mathbf{A}_2$ and back respectively. In the essential model, the lack of further arrows indicates that, for example, $\mathbf{e}_1$ "cannot happen" when the object is in state $\mathbf{A}_2$.

In Syntropy, the effect of transitions is specified by preconditions and postconditions similar to those used in Z or VDM. For example, $\mathbf{e}_1[\mathbf{P}]/\mathbf{Q}$, indicates

[8] Note that statecharts in Syntropy correspond only loosely to those of Harel [10, 11]. The semantics given here is intended to formalise statecharts as used in Syntropy and does not correspond to the semantics of Harel.

that transition $\mathbf{e}_1$ can only occur if the predicate **P** holds and that the two-state predicate **Q** must hold between the before and after states when $\mathbf{e}_1$ occurs.

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. The instance targeted by the event is passed as an extra parameter, **a**, of the object type (c.f. the extra parameter introduced in the **A**-morphism). In this case, the default filter, $\mathbf{a} =$ self, is assumed to indicate that only the object passed as parameter responds to the event. More complex situations can be modelled using this mechanism where the filter, **f**, is a predicate identifying which events the self object should react to.

The same event can correspond to more than one transition in the diagrammatic part of the statechart. Where the sources of the arrows are different states, the state gives the precondition for the transition. Where the same event name labels two arrows from the same state, the choice between them is indicated by separate explicit preconditions which are annotated directly on the arrows.

Unlike the use in Syntropy, we make a syntactic distinction between the event and its associated transitions by capitalising the event name and indexing the transition names.

2.6 Interpreting State Types and Events

The information in the class diagram is interpreted as in Sections 2.3 and 2.4. The statechart defines the actions of the classes and instances which were omitted in Section 2.3. Each arrow in the statechart represents a state transition and is interpreted as an action $\mathbf{e}_i$ of the instance theory. Several arrows can be used to describe different cases of a particular system event. The event itself is interpreted as an action **E** 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 the above diagram, if $\mathbf{e}_1$ and $\mathbf{e}_2$ are different transitions for the event **E(a)** then $\mathbf{e}_1$ and $\mathbf{e}_2$ are interpreted as separate actions in the instance theory of **A** whereas **E(a)** is interpreted in the theory of the $\{ \mathbf{A}, \mathbf{A}_1, \mathbf{A}_2 \}$ subsystem and then synchronised with $\mathbf{e}_1$ and $\mathbf{e}_2$ via an axiom of the form

$$\mathbf{E(a)} \;\Rightarrow\; \mathbf{a.e}_1 \vee \mathbf{a.e}_2$$

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

$$\mathbf{E(p[F])} \;\Rightarrow\; \forall \mathbf{a} \in @\mathbf{A} \cdot \mathbf{a.F} \Rightarrow \mathbf{a.e}_1 \vee \mathbf{a.e}_2$$

Here $\mathbf{a.F}$ is $\mathbf{F}$ with $\mathbf{a}$ substituted for self. For example, in the default case, where $\mathbf{F}$ is $\mathbf{p} = \mathbf{self}$, then $\mathbf{a.F}$ is $\mathbf{p} = \mathbf{a}$ and we regain the simpler condition above.

Note that when an event is not listed for a statechart, we require the event to go undetected by the object (rather than be blocked). In order to ensure this, we interpret unlisted events as having a filter of **false**.

Interpreting preconditions Preconditions in the 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 the transition $\mathbf{e}_1[\mathbf{P}]/\mathbf{Q}$ from state $\mathbf{A}_1$ to state $\mathbf{A}_2$. We define a *permission axiom* in the instance theory which expresses that $\mathbf{e}_1$ can only occur when $\mathbf{P}$ holds.

$$\mathbf{e}_1 \Rightarrow \mathbf{P}$$

Note, that this interpretation prevents preconditions from being weakened in refinement, that is, such transformations do not yield theory extensions. Thus subtyping form 5 of Chapter 8 of [5] (weakening preconditions) is not valid in essential models[9].

At the class level, each transition is also guarded by the state from which it occurs, for example, we have

$$\mathbf{a}.\mathbf{e}_1 \Rightarrow \mathbf{a} \in \overline{\mathbf{A}_1}$$

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.

For the above transition with postcondition, $\mathbf{Q}$, we have the *state-transition* axiom

$$\mathbf{e}_1 \Rightarrow \mathbf{Q}$$

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

At the class level, the event additionally moves the targeted instances to state $\mathbf{A}_2$

$$\mathbf{a}.\mathbf{e}_1 \Rightarrow \mathbf{a} \in \bigcirc\overline{\mathbf{A}_2}$$

Orthogonal state machines Syntropy allows non-interfering concurrency to be specified via orthogonal statecharts although multiple simultaneous events are not supported. The state space is now the cartesian product of the spaces indicated by the two statechart components. The above approach supports a simple interpretation of orthogonal statecharts in terms of their components. Again the reader is referred to [3] for details.

The above interpretation of filters, preconditions and postconditions assumes only local attributes are used in the defining expressions. However, Syntropy

[9] 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.

allows "navigation expressions" in which the conditions depend on the attributes of associated objects.

In such cases the relevant permission and state-transition axioms have to be "lifted" to an appropriate theory. Where the expression simply refers to an attribute of an associated object, the theory built for the association (**D** in the above diagram) suffices. Where the expression "navigates" further afield, a larger theory must be used. In this case, the theory required is that including all the visited theories. (The order of inclusion is not important as the co-limit construction is independent of the order in which the theories are combined.)

These larger theories are in any case part of the construction of the interpretation of the system as a whole, but until now their construction has been completely implicit. Therefore, in making explicit which theory interprets each navigation expression we add considerable complexity to the interpretation of the general system since each navigation expression potentially visits a collection of objects from different classes and hence identifies a new theory in its interpretation.

For this reason, we here advocate that a hierarchical approach is adopted when constructing system descriptions where predefined subsystems are employed and navigation expressions are confined as far as possible to the enclosing subsystem. (The same effect can be achieved by post-processing a system description to identify which collections of classes are linked by navigation expressions.) In the next section we discuss the use of subsystems as "first class" constructions in system description which leads to a definition of subsystem which is a generalisation of the concept of object and yields a hierarchical form of object orientation.

An example of application of the above semantics to the radio button statechart of Section 1 yields instance actions

1. $\mathbf{turn\_on_1(x)}$ representing the transition from **On** to itself in the case that $\mathbf{x = self}$;
2. $\mathbf{turn\_on_2(x)}$ representing the transition from **On** to **Off** in the case that $\mathbf{x \neq self}$;
3. $\mathbf{turn\_on_3(x)}$ representing the transition from **Off** to **On** in the case that $\mathbf{x = self}$;
4. $\mathbf{turn\_on_4(x)}$ representing the transition from **Off** to **Off** in the case that $\mathbf{x \neq self}$.

The class action **Turn_on(x)** then has the axiom:

$$\mathbf{Turn\_on(x)} \Rightarrow \forall \mathbf{a} : @\mathbf{RadioButton} \cdot \mathbf{a.turn\_on_1(x)} \vee \mathbf{a.turn\_on_2(x)} \vee \mathbf{a.turn\_on_3(x)} \vee \mathbf{a.turn\_on_4(x)}$$

because the filter is **true** (ie, every object **a** may potentially react to the event). But we know that

$$\mathbf{a.turn\_on_1(x)} \Rightarrow \mathbf{a} \in \overline{\mathbf{On}} \wedge \mathbf{x = a} \wedge \mathbf{a} \in \bigcirc\overline{\mathbf{On}}$$

by the axioms for this transition, and similarly for the other transitions, so that:

$$\begin{array}{l}\mathbf{Turn\_on(x)} \Rightarrow \\ \quad \forall \mathbf{a} : \mathbf{@RadioButton} \cdot \\ \qquad \mathbf{a} \in \overline{\mathbf{On}} \wedge \mathbf{x} = \mathbf{a} \wedge \mathbf{a} \in \bigcirc\overline{\mathbf{On}} \vee \\ \qquad \mathbf{a} \in \overline{\mathbf{On}} \wedge \mathbf{x} \neq \mathbf{a} \wedge \mathbf{a} \in \bigcirc\overline{\mathbf{Off}} \vee \\ \qquad \mathbf{a} \in \overline{\mathbf{Off}} \wedge \mathbf{x} = \mathbf{a} \wedge \mathbf{a} \in \bigcirc\overline{\mathbf{On}} \vee \\ \qquad \mathbf{a} \in \overline{\mathbf{Off}} \wedge \mathbf{x} \neq \mathbf{a} \wedge \mathbf{a} \in \bigcirc\overline{\mathbf{Off}}\end{array}$$

From this we can, after some work, deduce that $\bigcirc\overline{\mathbf{On}}$ must be $\{\mathbf{x}\}$ and $\bigcirc\overline{\mathbf{Off}}$ must be the complement of this set.

3 Subsystems

In the introduction we identified two key aspects of object orientation. Firstly, Objects aggregate related attributes, and secondly, Object Identifiers globally identify particular instances of one of these aggregations. In Section 2 we have seen how the formal interpretation of objects, classes and associations leads us to consider the concept of subsystem as a means of interpreting the constraints between related objects. From this perspective, subsystems provide aggregation, just as did objects, but at a coarser level of granularity. In this section we discuss whether other aspects of object orientation, in particular instance identity, can also usefully be applied to subsystems.

Although the concept of subsystem is not defined in Syntropy, we have seen how some notations implicitly assume it. We considered examples such as cardinality and lifetime constraints which must be interpreted in the subsystem comprising the association and the associated objects. In this respect, associations provide the simplest form of subsystem. Constraints on associations define particular properties of the subsystem. Navigation expressions which are interpreted in the theory including all visited constructs (objects and associations) are another example of a notation which requires the identification of a subsystem that encompasses the navigated path. Relationships between associations (Syntropy allows us to state that one association is a sub-relation of another) are also subsystem properties.

The above are particular cases of conditions we might give concerning the structures defined in a type hierarchy. Other, more general constraints might be:

- arbitrary properties for association,
- arbitrary properties relating associations,
- arbitrary properties inter-relating associated classes.

Subsystems provide the construction at the correct level of granularity to allow us to formalise such descriptions.

In the case of the **RadioButton** example, for instance, we can specify the effect of the **Turn_on(x)** event at the level of the subsystem which includes **RadioButton** and the two subtypes **On** and **Off**:

$$\mathbf{Turn\_on(x)} \Rightarrow \bigcirc\overline{\mathbf{On}} = \{\ \mathbf{x}\ \} \wedge \bigcirc\overline{\mathbf{Off}} = \overline{\mathbf{RadioButton}} \setminus \{\ \mathbf{x}\ \}$$

Likewise, the student teaching example of Section 1 can be expressed via a subsystem which includes the two classes and the three associations (Figure 8). **Tuition** can be regarded as a class with attributes $\overline{\mathbf{Student}}$ and $\overline{\mathbf{College}}$, the

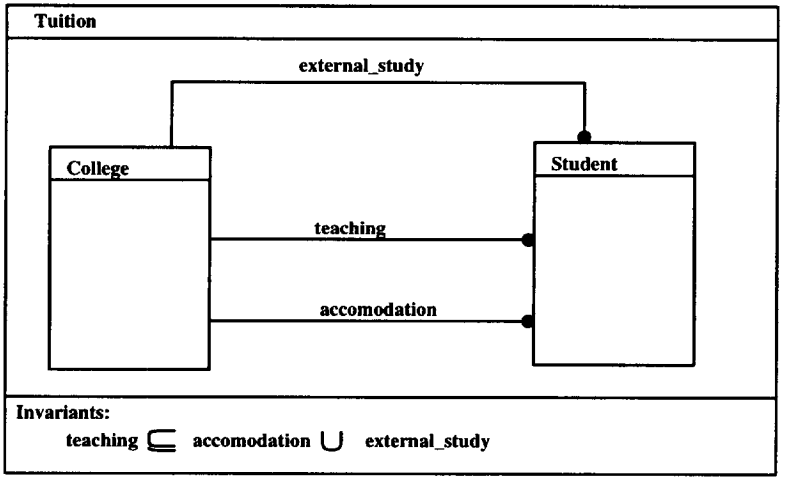


Fig. 8. Subsystem Example – Extended Notation

sets of existing students and colleges, and attributes

$$\mathbf{external\_study} : \overline{\mathbf{College}} \leftrightarrow \overline{\mathbf{Student}}$$
$$\mathbf{teaching} : \overline{\mathbf{College}} \leftrightarrow \overline{\mathbf{Student}}$$
$$\mathbf{accomodation} : \overline{\mathbf{College}} \leftrightarrow \overline{\mathbf{Student}}$$

representing the current value of the associations concerned. The local invariants of the classes can be deduced from this more global model. Operations that change associations would also be most naturally specified at the level of a subsystem which includes the association and the classes it connects.

The concept of subsystem exists in a restricted form in Fusion or in Octopus [2]. An aggregate in Fusion is a group of classes, associations and attributes (but not operations, at the analysis level). In Octopus it is used to partition an application into functional sub-applications, rather like the domain concept of Syntropy [6]. The concept of subsystem we are proposing is more general in that it is hierarchical construct which can be used to define more than two levels of structure, and allows properties and operations to be localised at an appropriate level.

Subsystems therefore support the principle that we should specify operations and properties as *locally as possible*, without compromising comprehensibility, but not more locally.

We propose the concept of subsystems as a generalisation of objects and associations. Subsystems should include the fundamental aspects of objects (aggregation and identity) and associations (relationships between objects) and should be usable as "first class objects" to give a nested hierarchy with encapsulation at each level (ie. vertical and horizontal structuring). We discuss each aspect in turn.

3.1 Aggregation - The class-instance approach

We advocate that the class-instance approach, as employed for objects, is also used for subsystems. A subsystem class is a collection of object classes (or subsystem classes when subsystems are nested) which are related by associations and which collect together related objects at the next lower level of granularity. To formally interpret a subsystem we would therefore define a manager theory and instance theories as was done for objects in Section 2.

A subsystem can be identified from an existing (one level) type view diagram by encircling those object classes which are to be included in the subsystem class. The included object classes must be linked by associations also within the subsystem.

The interface which encapsulates the subsystem is simply those associations of the constituent components which link to components outside the subsystem. For specification and implementation models, where a set of operations define the interface to an object (its protocol in Smalltalk terms), we identify a distinguished set of the lower level operations to act as subsystem operations.

3.2 Identity - The root object

Syntropy employs a useful convention by which every system description should have a root object class to which every class is associated (either directly or via other classes). The root class must have precisely a single instance in any particular instantiation of the system. All instances in the system must also be associated to the root instance (either directly or indirectly) at all times.

We propose that the concept of root object can usefully be employed at the level of subsystems as well as for the entire system. The identity of the root instance can then be used as the identity of the subsystem instance or a new indexing of subsystem instances by subsystem identifiers can be provided. It is of little consequence whether subsystem identifiers are globally unique or whether they are unique within the enclosing subsystem instance at the next higher level. In the latter case, tuples of subsystem identifiers (corresponding to the nested levels of subsystem) can be used to identify subsystem instances.

3.3 Open issues

Two important open issues remain in the definition of the concept of subsystem. Firstly, it is not clear whether enclosed subsystems can be shared between enclosing ones and secondly, it remains to be defined how notions of subclass and inheritance should be applied to subsystems.

4 Conclusions and further work

We have formalised some aspects of class diagrams and statecharts as used in the "Syntropy" method of Object Oriented Analysis and Design. This interpretation has been axiomatic as opposed to others which are primarily denotational such as [1].

We have shown that a formal and modular semantics can be given to Syntropy essential models where separate theories are defined for instances, class managers and associations. Statecharts are interpreted with their diagrammatic part defined in the instance theory, and textual part in the class theory. We employed a style of axiomatisation where preconditions and postconditions of actions are defined in terms of the attributes so giving a style of specification very similar to that of model-oriented formalisms such as VDM and Z.

This formalisation could form the basis for a system supporting reasoning about the models developed enabling the use of proof for the validation and verification of designs. The same approach could be taken to the interpretation of other notations such as UML and therefore improve usefulness of these methods and also the process of the definition of the methods themselves.

In interpreting Statecharts, we distinguished between local actions for instance state transitions and system actions for events. We adopted a style of axiomatisation where the local effect of actions is interpreted directly in terms of the local attributes and synchronisation between actions in different theories is given implicitly by trans-theory constraints on attributes. Were theories to be executable, these implicit constraints would require implementation to ensure the synchronisations between actions of different classes.

The formalisation has brought to the fore some features of the langauge which might otherwise be unclear. For example it distinguishes between preconditions and guards, shows the orthogonality of cardinality and lifetime constraints, and seperates concepts of the generic instance from those of the class manager.

This formalisation of Syntropy has indicated some areas where notations are non-modular. We observed that the formalisation of associations has to be undertaken in the theory which incorporates the object and association primitives and that the formalisation of navigation expressions requires an amalgamation of an arbitrary collection of theories. We proposed the concept of subsystems as a coarser grained generalisation of objects and suggested that they can then be used in a hierarchical description of systems employing nested subsystems. We advocate that the class-instance approach can be usefully employed for subsystems just as it is for objects. The use of instance identity is also proposed for subsystems.

The treatment of unborn and dead instances requires infinite colimit diagrams. It is believed that these infinite colimits are well behaved because the morphisms are almost everywhere trivial, however, the underlying mathematics for this does need to be rehearsed.

Significantly, we have only interpreted essential models, some aspects of Specification and Implementation models are similar, others would require further

work. Within essential models, we have not attempted to formalise nested statecharts nor associations with attributes. We believe that the concept of subsystem will also be of use here. Demonstrating the correctness of refinements between levels of model is not even addressed informally in Syntropy.

References

1. M Abadi and L Cardelli, *An Imperative Object Calulus*, TAPSOFT '95, Mosses, Nielsen and Schwartzbach (Eds), Springer-Verlag, LNCS 915, 1995.
2. Maher Awad, Juha Kuusela and Jurgen Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion.* Prentice Hall, Upper Saddle River, NJ, March 1996.
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. Coleman D. *et al., Object-oriented Development: The FUSION Method.* Prentice Hall Object-oriented Series, 1994.
5. Cook and Daniels, *Designing Object Systems with Syntropy*, Prentice Hall, 1994.
6. S Cook and J Daniels. Syntropy Case Study: The Petrol Station. Technical report, Object Designers Ltd., 1996.
7. J. Fiadeiro and T. Maibaum, *Temporal Theories and Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing, Vol.4, No. 3, 1992. Springer-Verlag.
8. 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.
9. Goguen, J. and Burstall, R. Introducing Institutions. In Clarke and Kozen, eds. Logics of Programs, pp. 221-256, Springer-Verlag, 1984.
10. D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Sci. Comput. Prog. 8 pp. 231-274 (1987).
11. D. Harel and E. Gery, *Executable Object Modelling with Statecharts* Proc. 18th Int. Conf. Soft. Eng., IEEE Press, 1996, pp. 246-257.
12. L. Lamport, The Temporal Logic of Actions, Digital Technical Report 79, 130 Lytton Avenue, Palo Alto, Califiornia 94301. December 25th, 1991.
13. K. Lano, Enhancing Object-Oriented Methods with Formal Notations, TAPOS, to appear, 1997.
14. Rumbaugh, J. et al. Object-Oriented Modelling and Design, Prentice-Hall, Englewoods Cliffs, New jersey, 1991.
15. M. Spivey, The Z Notation: a reference manual, Prentice-Hall, 1992.
16. 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.