ROO – A Model for Object-Oriented Reuse

Tony Clark
Computing Department
Phoenix Building
University of Bradford
BD7 1DP
e-mail: a.n.clark@comp.brad.ac.uk
September 18, 1997

1 Abstract

Both object-orientation and the Internet make the widespread reuse of software a possibility. Unfortunately, the potential benefits from these facilities have not been forthcoming. One reason for this is the lack of a coherent model for software development and reuse. This paper proposes such a model which is based upon modelling software components using state transition machines. Reuse is made possible by defining matching relations between component descriptions in terms of machine simulations. Both the development process and matching relations are given a formal semantics.

2 Introduction

The reuse of software components has long been a major aim of Software Engineering [12]. We propose that there are two major reasons why this aim has not yet been realised. Firstly, no single organisation can afford to develop all the software which it will subsequently need to reuse (and until recently there has been no effective mechanism which allows different organisations to pool software resources). Secondly, if different organisations *are* to contribute to a shared pool of software there must be a high degree of standardisation in terms of software components, the mechanisms by which the components are made available for reuse and the retrieval mechanisms which match a software requirement with a software component.

The worldwide use of computers is currently undergoing a revolution in terms of information availability. In principle, a computer may gain access to information at any point in the world using standard communications networks. The scope for information sharing and reuse is potentially enormous since information at a single site may be shared with all other sites in the world simply by locally making it available to the Internet [4].

The amount and format of information which is available is a problem which must be overcome in order to effectively use the Internet as a shared resource. In order to deal with the quantity of information, searching must be automated. However, in order to automate the search for information, the format of the information must be complete, usable by the recipient and understood by the search mechanisms.

The object-oriented approach to software development offers advantages for the reuse of software components. It is claimed that object-oriented designs and implementations are problem oriented, have a resilience to evolution and are amenable to domain analysis [16]. Software components are organised into collections of relatively small independent units, each of which performs a specific task. The object-oriented model of computation is organised around a message passing metaphor, where individual objects interact by sending data bearing messages.

Java is an object-oriented programming language which compiles to a standard binary format. Different computer platforms may support Java by implementing an interpreter for the binary format. A Java program consists of a collection of separate class definitions. A class definition on one computer may be transported via the Internet to another computer and executed without recompilation. This high degree of modularity and portability makes Java an ideal candidate to support code reuse.

This work is aimed at developing a systematic method of producing object-oriented software which incorporates reuse. In particular we intend to target Java as the development language since it provides excellent support for multi-platform portability. The method will support both publication of code for reuse and the development of new software which involves reuse. The proposed method is described in the rest of the paper: §3 gives an overview of our model for object-oriented software development, documentation and reuse; §4 gives a formal semantics to the model; §5 gives some simple examples of the approach; finally, §6 concludes by analysing the approach and outlining future work.

3 A model of development and reuse

This section gives an overview of a model for software development and reuse. The model, called ROO, is the basis for a simple prototype tool, called ROOT, which has been developed. The model is given a formal semantics in §4.

We view software development as a sequence of transformations which are applied to an initial system description to produce a final system description. Each transformation adds detail to the current description and may be viewed as constraining the collection of all programs which will satisfy it. The development process starts with an empty initial system description which is satisfied by all possible programs. The development process ends when sufficient detail has been added to ensure that only the required programs will satisfy the system description. Note that one way of ensuring this is to transform the initial description into executable code, *i.e.* to a singleton set of programs.

Once a program has been developed in this way, it may be registered as available for reuse. Each of the system descriptions produced during a program's development will document the program to varying degrees of specificity. In general, the most specific system description is used to document a program.

Once a library of system descriptions is established, it is possible to introduce reuse to the development process. After applying a transformation to produce a system description d, the library is searched for a program p whose system description matches d. Given suitable matching criteria we may conclude that p implements d. Providing that the only programs which satisfy d are those which match our software requirements, then we conclude that p meets the software requirements and may be *reused*.

This section proceeds as follows: §3.1 describes the transformations which may be applied to system descriptions; §3.2 describes the issues which apply to matching system descriptions; §3.3 describes a model which is used for system descriptions; finally, §3.4 gives an example using ROO.

3.1 Refinement

Software development proceeds by performing transformations on a system description. Each transformation increases the specificity of the description and reduces the number of programs which satisfy it. A system may be described at different levels of abstraction. Transformations will change the level of abstraction. Abstraction levels are similar to viewing a distant object: at high levels of abstraction (viewing from far away) it is difficult to distinguish between different elements of an object; at low levels of abstraction (viewing from close up) individual elements can be distinguished. In our model, development moves from a highly abstract system description to a detailed system description. There are two types of system transformation [22] [13]:

1. Horizontal transformations leave the level of abstraction unchanged. Such transformations tend to modify the behaviour of the system, either by extending, replacing or removing behaviour. Horizontal transformations tend to change the character of a system description, for example by adding persistence capabilities to a data type or removing the 'reset' button on a menu.

2. Vertical transformations change the level of abstraction but leave the essential behaviour of the system unchanged. Vertical transformations are like taking one step towards a system or away from it. A vertical transformation which increases abstraction will coalesce distinct elements of a system and it will no longer be possible to distinguish between them after the transformation; for example, merging a collection of non-empty stack configurations into a single atomic *nonempty* configuration. A vertical transformation which decreases abstraction will decompose elements of a system into collections of elements which were previously indistinguishable; for example, decomposing the atomic *nonempty* stack back into a collection of individual stack configurations.

Software development proceeds from an initial system description by applying a sequence of horizontal and vertical transformations. The development proceeds from high to low abstraction until an error is discovered in the current system description. At this point the developer backtracks to the point at which the transformation which injected the error occurred and performs and different transformation. The result of performing an entire development is a tree structure.

3.2 Matching

In order to reuse programs we require a measure of similarity between two program descriptions. This measure is formally defined in §4.3. This section gives an overview of the possible definitions for *similarity* in the context of the proposed refinement model. We propose a denotational semantics for system descriptions, and then use the semantics to define matching constraints.

Let d be a system description. The set $D(d)$ contains all the sub-descriptions of d. The set $P(d)$ contains all the possible programs which satisfy d. For example, if d is the empty description then $P(d)$ contains all the possible programs which could ever be written. If d is refined to d' in a development process then d' contains more detail than d but $P(d') \subseteq P(d)$ since fewer programs will satisfy the extra behavioural constraints.

The set $N(d)$ contains the names in the external interface of the system d. The set $S(d)$ is the signature of d, *i.e.* the set of names and types in the interface of d. A renaming θ may be applied to a description $\theta(d)$ to change the names in the signature of d.

Given two system descriptions d_1 and d_2 we wish to decide whether or not any program which satisfies one will also satisfy the other. The description d_1 is to be thought of as a query, arising during a development process, and description d_2 is to be thought of as a library entry which documents existing code. In practice, it is likely to be very difficult to be certain of this relationship, so we intend to introduce a spectrum where descriptions matching at one end of the spectrum do so *weakly* and descriptions matching at the other do so *strongly*.

At the weak end of the spectrum we consider the names and signatures of the two descriptions. If:

$$N(d_1) \subseteq N(d_2) \tag{1}$$

then we can conclude that all of the names in the interface of d_1 are in the interface of d_2 and there is a small likelihood that any program which implements d_2 also implements d_1. Less weakly, if:

$$S(d_1) \subseteq S(d_2) \tag{2}$$

then both the names and their types in the interface of d_1 are contained in the interface of d_2. This increases the likelihood that any program which implements d_2 also implements d_1.

Both weak versions of the matching relation do not take the behaviour of the descriptions into account. This can be taken into account as follows, if:

$$d_3 \in D(d_2) \wedge d_4 \in D(d_1) \wedge P(d_3) \subseteq P(\theta(d_4)) \tag{3}$$

then we conclude that the implementations of some sub-description d_3 of d_2 are also implementations of some sub-description d_4 of d_1 after a renaming. We must be careful about how *sub-description* is defined, since the degenerate case is the empty description. In practice, there will be some limit as to the minimum size of a sub-description, for example one which minimally spans the behaviour of d_1 in some sense.

Stronger still, is the relation induced by the following condition:

$$d_3 \in D(d_2) \wedge d_4 \in D(d_1) \wedge P(d_3) \subseteq P(d_4) \tag{4}$$

since no renaming is necessary. Next is the relation induced by the following condition:

$$d_3 \in D(d_2) \wedge P(d_3) \subseteq P(\theta(d_1)) \tag{5}$$

which forces the whole of the behaviour defined by d_1 to be taken into account, albeit after a renaming. Finally, the strongest condition is:

$$d_3 \in D(d_2) \wedge P(d_3) \subseteq P(d_1) \tag{6}$$

the whole of the description d_1 must be dealt with. Notice that equivalence $P(d_1) = P(d_2)$ is not really sensible as a constraint since it implies that d_1 and d_2 are at the same level of abstraction at which point we have implemented the code which we intend to reuse.

Many current matching strategies use module signatures with pre- and post-conditions on each module operation (for example [14] [21] [1] [23]). Whilst languages which are based on such methods are very expressive they are not necessarily executable and matching may involve theorem proving which may be arbitrarily complex. The work described in [24] is an example of a strategy based on matching signatures.

Other strategies are based on keyword matching [3] which requires foresight on the part of the library designer in order to predict the keywords which will be used in a library search. We propose it is better to base a library search mechanism on the behaviour of the components, since this can be described in abstract terms without resorting to subjective keywords. The relationship between behaviours and subtyping is described in [2].

3.3 Modelling

The proposed software development process involves system descriptions which are denoted by *software component descriptions*. We would like a software component description language to support the object-oriented model of computation, to be executable so that development is interactive, to support transformations and to support matching.

The *3C* model [22] requires that a software component description language supports the following three views of a software component: *concept*, *content* and *context*. The concept view describes what a component does; the content view describes how the component achieves the behaviour; the context view describes the domain of applicability for the component.

Object-oriented development is particularly amenable to description using the *3C* model. Each software object in a development is an independent unit which often corresponds closely to a domain object. The behaviour of an object is often loosely coupled to that of other objects and the independent nature of objects behaviour tends to lead to wide domains of applicability.

Many current development methods for object-oriented software (for example [7] [15] [11] [17] [6] [18]) describe the behaviour of objects using state transition machines [5]. Such behaviour descriptions are suitable as a vehicle for component description since they can be used to capture the abstract behaviour of an object (conceptual view) and they can be used to produce an implementation of an object (content view) as a result of transformations.

The use of state transition machines for developing object-oriented software meets our objectives since a machine is executable and is amenable to both horizontal and vertical transformations (see [9] for a description of state transition machine transformation in the context of Knowledge Based Systems). A component description language which is based upon state transition machines can be given a formal semantics. The criteria for component reuse can be defined in terms of machine simulation. A request for a software component is made in terms of the required behaviour, supplied as a state transition machine. A given software component is a likely candidate for reuse when its behaviour description is capable of simulating the required behaviour.

A labelled state transition machine is represented as a triple (Θ, Σ, Γ). The machines are used to represent system components and are precisely defined in §4. Informally, Θ describes the states which the software component can exist in, Σ describes the operators which are available in the interface of the software component and Γ describes the state changes which occur in response to activating an operator.

3.4 Example

This section gives a very simple example of a component development and shows matching relationships between the resulting components. The aim is to produce a simple stack behaviour using horizontal and vertical transformations. The development is shown in the following diagram:

$$
\begin{tikzcd}
7 \arrow[d, "t_1"] & \\
8 \arrow[d, "t_2"] & \\
9 \arrow[r, "t_3"'] & 10 \arrow[d, "t_4"] \\
& 11
\end{tikzcd}
$$

where nodes refer to components produced during the development and labelled edges refer to horizontal and vertical transformations. The universal initial component description, 7, contains a single state and a single operator which takes the state to itself. This behaviour describes any component at the highest possible level of abstraction: all component states are coalesced into a single state and all component operators are coalesced into a single operator which appears to do nothing:

$$
\begin{array}{l}
\Theta = \{s\} \\
\Sigma = \{f : 7 \rightarrow 7\} \\
\Gamma = \{f : s \longmapsto s\}
\end{array}
\tag{7}
$$

The signature of 7 defines the type of f to be $7 \rightarrow 7$, *i.e.* it expects a value whose behaviour is described by 7 and produces a value whose behaviour is described by 7. There is one transition for 7 which is labelled with the operator f and has s as source and target states.

The first development transformation is to lower the level of abstraction by distinguishing between two stack states *empty* and *nonempty*. These are mutually exclusive and are produced by applying the horizontal transformation t_1 to 7 producing 8:

$$
\begin{array}{l}
\Theta = \{\mathit{empty}, \mathit{nonempty}\} \\
\Sigma = \{f : 7 \rightarrow 7\} \\
\Gamma = \{f : \mathit{empty} \longmapsto \mathit{empty}, \\
\qquad f : \mathit{empty} \longmapsto \mathit{nonempty}, \\
\qquad f : \mathit{nonempty} \longmapsto \mathit{nonempty}, \\
\qquad f : \mathit{nonempty} \longmapsto \mathit{empty}\}
\end{array}
\tag{8}
$$

The component 8 has two states, a single operator and four transitions. The four transitions are produced by systematically splitting the source and target states in the single transition $f : s \rightarrow s$ in 7.

In general, at this point in development, a number of horizontal transformations would be applied to 8 in order to tailor the new transitions. This is not necessary for 8 since all the transitions are possible at this level of abstraction. A second vertical transformation is applied to 8 to decrease the level of abstraction with respect to the operation f. The transformation t_2 produces 9:

$$\begin{aligned}
\Theta = \{&\mathit{empty}, \mathit{nonempty}\}\\
\Sigma = \{&\mathit{push} : 7 \rightarrow 7,\\
&\mathit{pop} : 7 \rightarrow 7,\\
&\mathit{top} : 7 \rightarrow 7\}\\
\Gamma = \{&\mathit{push} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{push} : \mathit{empty} \longmapsto \mathit{nonempty},\\
&\mathit{push} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{push} : \mathit{nonempty} \longmapsto \mathit{empty},\\
&\mathit{pop} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{pop} : \mathit{empty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{empty} \longmapsto \mathit{nonempty},\\
&\mathit{top} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{top} : \mathit{nonempty} \longmapsto \mathit{empty}\}
\end{aligned} \tag{9}$$

The single operator f in 8 has been decomposed into three operators *push*, *pop* and *top*. In 9 it is now possible to distinguish between three operators each of which affects the stack state in the same way as operation f in 8. Notice that not all of the transitions which are defined in 9 are required. A horizontal transformation t_3 is applied which removes undesirable transitions (in general, this would be performed by a sequence of horizontal transformations) producing 10:

$$\begin{aligned}
\Theta = \{&\mathit{empty}, \mathit{nonempty}\}\\
\Sigma = \{&\mathit{push} : 7 \rightarrow 7,\\
&\mathit{pop} : 7 \rightarrow 7,\\
&\mathit{top} : 7 \rightarrow 7\}\\
\Gamma = \{&\mathit{push} : \mathit{empty} \longmapsto \mathit{nonempty},\\
&\mathit{push} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{nonempty} \longmapsto \mathit{nonempty}\}
\end{aligned} \tag{10}$$

The component 10 correctly describes the behaviour of a simple stack object. Notice that the type of the elements which are stored in the stack are defined by 7, *i.e.* they are totally unrestricted. It is unlikely that in a real development, the contents of a particular stack would be unrestricted. To show how the model supports this, the elements of the stack are restricted to be components of type 10, *i.e.* stacks of unrestricted stacks. This is achieved by performing a vertical transformation t_4 to produce 11:

$$\begin{aligned}
\Theta = \{&\mathit{empty}, \mathit{nonempty}\}\\
\Sigma = \{&\mathit{push} : 10 \rightarrow 7,\\
&\mathit{pop} : 7 \rightarrow 7,\\
&\mathit{top} : 7 \rightarrow 10\}\\
\Gamma = \{&\mathit{push} : \mathit{empty} \longmapsto \mathit{nonempty},\\
&\mathit{push} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{nonempty},\\
&\mathit{pop} : \mathit{nonempty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{empty} \longmapsto \mathit{empty},\\
&\mathit{top} : \mathit{nonempty} \longmapsto \mathit{nonempty}\}
\end{aligned} \tag{11}$$

The sequence of component descriptions 7 – 11 describes a simple development process and gives an overview of the types of transformations which are used. At each step in the process further detail is added to the description which restricts the number of possible programs which satisfy it. At each step we would like to search a software library in the hope of finding programs whose component descriptions match the current step. Consider the constraints 1 – 6 described in §3.2. Each constraint is discussed with respect to the development given above where a current development component will be referred to as a *d-component* and a library component will be referred to as an *l-component*:

- Constraint 1 requires that the operator names of the d-component to be present in the l-component. In general, during development the d-component names at the outset will differ from those at the end, for example $N(7) = \{f\}$ and $N(11) = \{push, pop, top\}$. Keyword matching is therefore unlikely to yield a high number of desirable matches.

- Constraint 2 requires that the signature of the d-component is present in the signature of the l-component. This suffers from the same problems as constraint 1; although, once type information has been added to d-component operators, for example 11, then this can be used to distinguish the d-component signature from l-components with the same interface of names.

- Constraint 3 requires that some sub-behaviour of the d-component be consistent with some sub-behaviour of the l-component after a renaming. We must be careful when defining the criteria for *consistent sub-behaviour.* One possibility is that all the distinct d-states must be supported by the l-component, *i.e.* for each d-state there is a set of potential l-states. The sub-behaviour issue arises due to the following constraint: from each d-state there is a collection of possible d-operators, also each d-state is associated with a collection of l-states, all the d-operators must be supported by a corresponding l-operator but each l-state need not support *all* the corresponding l-operators.

  This constraint does not occur in the stack example, however consider the following two component descriptions:

$$\begin{array}{l} \Theta = \{s_1, s_2\} \\ \Sigma = \{f : 7 \rightarrow 7\} \\ \Gamma = \{f : s_1 \longmapsto s_2, f : s_2 \longmapsto s_1\} \end{array} \tag{12}$$

$$\begin{array}{l} \Theta = \{p_1, p_2, p_3\} \\ \Sigma = \{g : 7 \rightarrow 7\} \\ \Gamma = \{g : p_1 \longmapsto p_2, g : p_3 \longmapsto p_1\} \end{array} \tag{13}$$

  where 13 has been derived from 12 using a sequence of transformations. In 12 it is possible to move freely from s_1 to s_2 and back again. In 13 the state s_2 has been split into p_2 and p_3 and many of the resulting transitions have been deleted. We can associate s_1 with p_1 and associate s_2 with p_2 and p_3; however only part of the behaviour allowed by 12 in state s_1 is supported by 13 and the behaviour allowed by 12 in state s_2 has been divided between p_2 and p_3.

- Constraint 4 is similar to constraint 3 except that no renaming is possible. This would force g and f to be the same in components 12 and 13.

- Constraint 5 requires the behaviour of the d-component to be completely supported by the l-component possibly after a renaming. Consider a d-component

7 being matched against l-components 8 – 11. Using suitable renamings for f, all the l-components support the behaviour of the d-component.

Now consider a d-component 8 being matched against l-components 7, 9 – 11. The d-component does not match 7 since there are insufficient states in the l-component. The d-component does match all of the l-components 9 – 11 using constraint 5 since the d-behaviour can be completely embedded within the corresponding l-behaviour.

The d-component 11 does not match any of the l-components 7 – 10 using constraint 5 since its behaviour is too complex to be embedded.

- Constraint 6 is similar to constraint 5 except that no renaming is possible. For example 8 matches 10 using constraint 5, but not using constraint 6.

In conclusion, we have given an overview of ROO, an approach to software development using simple state transition machines and outlined a space of possible matching constraints which can be used as criteria for reuse. The rest of this paper gives the formal semantics of ROO and then gives an example in greater detail.

4 Semantics of development and reuse

Earlier sections in this paper have described a model for software development and reuse. This section makes this model precise by giving it a formal semantics.

4.1 Definitions

A *software component* is a triple (Θ, Σ, Γ) where Θ is a set of states, Σ is a *signature* which is a set of operators, and Γ is a set of transitions.

A *state* $\theta \in \Theta$ is an atomic value. An *operator* $\sigma : \tau_1 \rightarrow \tau_2 \in \Sigma$ consists of an operator identifier σ and a pair of types τ_1 and τ_2 which denote the domain and range of the operator. A *type* τ is a software component. A *transition* $\sigma : \theta_1 \longmapsto \theta_2$ consists of an operator identifier σ and a pair of states; θ_1 is referred to as the *source* state and θ_2 is referred to as the *target* state.

The substitution $[\sigma_1/\sigma_2]$ is used to replace σ_2 with σ_1. Similarly, $[\theta_1/\theta_2]$ replaces θ_2 with θ_1. Substitutions are applied to operator identifier and states respectively; the definition, where x is an operator identifier or state, is as follows:

$$\begin{array}{ll} x[x_1/x_2] = x_1 & \text{when } x = x_2 \\ x[x_1/x_2] = x & \text{otherwise} \end{array}$$

The infix operators $\otimes$ and $\oplus$ are used to denote binary relations.

4.2 Transformations

A transformation may be either horizontal or vertical. Horizontal transformations add or delete states, transitions and operators. Vertical transformations refine the type of operators or split states and operators.

The horizontal transformations each add or delete a value from the relevant entries in the software components. They are defined below without further comment:

$$\begin{array}{l} \mathit{addstate}(\theta)(\Theta, \Sigma, \Gamma) = (\Theta \cup \{\theta\}, \Sigma, \Gamma) \\ \mathit{delstate}(\theta)(\Theta, \Sigma, \Gamma) = (\Theta - \{\theta\}, \Sigma, \Gamma) \\ \mathit{addop}(\sigma : \tau_1 \rightarrow \tau_2)(\Theta, \Sigma, \Gamma) = (\Theta, \Sigma \cup \{\sigma : \tau_1 \rightarrow \tau_2\}, \Gamma) \\ \mathit{delop}(\sigma)(\Theta, \Sigma, \Gamma) = (\Theta, \Sigma - \{\sigma\}, \Gamma) \\ \mathit{addtrans}(\gamma)(\Theta, \Sigma, \Gamma) = (\Theta, \Sigma, \Gamma \cup \{\gamma\}) \\ \mathit{deltrans}(\gamma)(\Theta, \Sigma, \Gamma) = (\Theta, \Sigma, \Gamma - \{\gamma\}) \end{array}$$

The vertical transformations are slightly more complex. Each transformation is formally defined and then explained.

$$\begin{array}{l} \mathit{splitstate}(\theta_1, \theta_2, \theta_3)(\Theta, \Sigma, \Gamma) = ((\Theta - \{\theta_1\}) \cup \{\theta_2, \theta_3\}, \Sigma, \Gamma') \\ \quad \textbf{where} \\ \qquad \Gamma' = \bigcup\{\{\sigma : \theta[\theta_2/\theta_1] \longmapsto \theta', \sigma : \theta[\theta_3/\theta_1] \longmapsto \theta'\} \mid \sigma : \theta \longmapsto \theta' \in S\} \\ \qquad\quad \textbf{where} \\ \qquad\qquad S = \bigcup\{\{\sigma : \theta \longmapsto \theta'[\theta_2/\theta_1], \sigma : \theta \longmapsto \theta'[\theta_3/\theta_1]\} \mid \sigma : \theta \longmapsto \theta' \in \Gamma\} \end{array}$$

The *splitstate* transformation involves an existing state θ_1 and two new states θ_2 and θ_3. The effect is to replace θ_1 by both θ_2 and θ_3 in a given software component. To do this, the existing state is removed from Θ and the new states are added. Then each transition in Γ must be duplicated, replacing θ_1 with both θ_2 and θ_3, producing a new transition set Γ'.

$$\begin{array}{l} \mathit{splitop}(\sigma_1, \sigma_2, \sigma_3)(\Theta, \Sigma, \Gamma) = (\Theta, (\Sigma - \{\sigma_1\}) \cup \{\sigma_2, \sigma_3\}, \Gamma') \\ \quad \textbf{where} \\ \qquad \Gamma' = \bigcup\{\{\sigma[\sigma_2/\sigma_1] : \theta_1 \longmapsto \theta_2, \sigma[\sigma_3/\sigma_1] : \theta_1 \longmapsto \theta_2\} \mid \sigma : \theta_1 \longmapsto \theta_2 \in \Gamma\} \end{array}$$

The *splitop* transformation involves an existing operation σ_1 and two new operations σ_2 and σ_3. The effect is to replace the existing operation σ_1 by both σ_2 and σ_3 in a given software component. To do this, the existing operator is removed from Σ and the two new operators are added. Then each transition in Γ which is labelled with the σ_1 is replaced with two new transitions which are labelled with σ_2 and σ_3 respectively.

$$\begin{array}{l} \mathit{refineop}(\sigma, \tau_1, \tau_2, \oplus)(\Theta, \Sigma, \Gamma) = \\ \quad \begin{cases} (\Theta, (\Sigma - \{\sigma : \tau_1\}) \cup \{\sigma : \tau_2\}, \Gamma) & \text{when } \tau_1 \oplus \tau_2 \\ \perp & \text{otherwise} \end{cases} \end{array}$$

The *refineop* transformation involves an existing operator σ whose type is τ_1 and a new type τ_2. The effect is to replace the existing operator type with the new operator type only when the types are consistent with respect to the supplied relation $\oplus$.

Transformations may be constructed without reference to a specific software component. Such transformations are combined using the infix operator $\star$ which is defined as follows:

$$(t_1 \star t_2)(\Theta, \Sigma, \Gamma) = t_2(t_1(\Theta, \Sigma, \Gamma))$$

As an example of a composite transformation the following corresponds to the transformation used in the development in §5:

$$\begin{array}{l} t_1 \star t_{2a} \star t_{2b} \star t_3 \star \ldots \star t_4 \ldots \\ \quad \textbf{where} \\ \qquad t_1 = \mathit{splitstate}(s, \mathit{empty}, \mathit{nonempty}) \\ \qquad t_{2a} = \mathit{splitop}(f, \mathit{push}, g) \\ \qquad t_{2b} = \mathit{splitop}(g, \mathit{pop}, \mathit{top}) \\ \qquad t_3 = \mathit{deltrans}(\mathit{push} : \mathit{empty} \longmapsto \mathit{empty}) \\ \qquad t_4 = \mathit{refineop}(\mathit{push}, 7 \to 7, 10 \to 10, \oplus) \end{array}$$

where ... represents a repetition of the preceding type of transformation and $\oplus$ is a simulation relation as explained in §4.3.

4.3 Simulation

Matching software components is based on the notion of *simulation* [5]. One transition machine simulates another when some or all of the states and transitions which are performed by one machine can be embedded in the other. Simulation is *total* when all of the states and transitions can be embedded and is *partial* otherwise. During system development, the program associated with an l-component is a candidate for reuse when the l-component simulates the current d-component.

Simulation is formally defined using a family of relations. Each relation has the form (s, o) where s is $\mathbf{t}$ when the simulation is total and is $\mathbf{p}$ when the simulation is partial and where o is $\mathbf{f}$ when the operator names must be fixed and is $\mathbf{l}$ when the operator names may be different. Given two components c_1 and c_2, c_1 is said to be simulated by c_2, with respect to s and o when the corresponding relation holds. Each relation is defined in turn and then explained. The relation $(\mathbf{p}, \mathbf{l})$ is equivalent to 3 in §3.2 and is defined as follows:

$$
\begin{array}{l}
(\Theta_1, \Sigma_1, \Gamma_1)\ (\mathbf{p}, \mathbf{l})\ (\Theta_2, \Sigma_2, \Gamma_2)\ \textit{iff} \\
\quad \exists \oplus \subseteq \Theta_1 \times \Theta_2 \bullet \textit{dom}(\oplus) = \Theta_1 \wedge \\
\quad\quad \exists \otimes \subseteq \Sigma_1 \times \Sigma_2 \bullet \textit{dom}(\otimes) = \Sigma_1 \wedge \\
\quad\quad\quad ((\sigma_1 : \tau_1 \rightarrow \tau_2) \otimes (\sigma_2 : \tau_3 \rightarrow \tau_4) \Rightarrow (\tau_1\ (\mathbf{p}, \mathbf{l})\ \tau_3) \wedge (\tau_2\ (\mathbf{p}, \mathbf{l})\ \tau_4)) \wedge \\
\quad\quad\quad\quad \forall \sigma_1 : \theta_1 \longmapsto \theta_2 \in \Gamma_1 \bullet \\
\quad\quad\quad\quad\quad \exists \sigma_2 : \theta_3 \longmapsto \theta_4 \in \Gamma_2 \bullet \\
\quad\quad\quad\quad\quad\quad (\sigma_1 \otimes \sigma_2) \wedge (\theta_1 \oplus \theta_3) \wedge (\theta_2 \oplus \theta_4)
\end{array}
$$

The relation $(\mathbf{p}, \mathbf{l})$ holds between two components when the first is partially simulated by the second, *i.e.* *every* operation which is possible in state of the first component must be possible in *some* corresponding state of the second component. The names of the operations which are involved in the simulation need not be the same in both components. The relation $\oplus$ holds between the states of the two components and the relation $\otimes$ holds between the operations. For each transition which can be performed by the simulated component, the simulating component must contain a transition which is consistent given the state and operation relations. This is described by the following diagram:

$$
\begin{tikzcd}
\sigma_1 : \arrow[d, "\otimes"'] & \theta_1 \arrow[r] \arrow[d, "\oplus"'] & \theta_2 \arrow[d, "\oplus"] \\
\sigma_2 : & \theta_3 \arrow[r] & \theta_4
\end{tikzcd}
$$

The relation $(\mathbf{p}, \mathbf{f})$ is equivalent to 4 in §3.2 and is defined as follows:

$$
\begin{array}{l}
(\Theta_1, \Sigma_1, \Gamma_1)\ (\mathbf{p}, \mathbf{f})\ (\Theta_2, \Sigma_2, \Gamma_2)\ \textit{iff} \\
\quad \exists \oplus \subseteq \Theta_1 \times \Theta_2 \bullet \textit{dom}(\oplus) = \Theta_1 \wedge \\
\quad\quad \exists \otimes \subseteq \Sigma_1 \times \Sigma_2 \bullet \textit{dom}(\otimes) = \Sigma_1 \wedge \\
\quad\quad\quad ((\sigma_1 : \tau_1 \rightarrow \tau_2) \otimes (\sigma_2 : \tau_3 \rightarrow \tau_4) \Rightarrow (\tau_1\ (\mathbf{p}, \mathbf{f})\ \tau_3) \wedge (\tau_2\ (\mathbf{p}, \mathbf{f})\ \tau_4)) \wedge \\
\quad\quad\quad\quad \forall \sigma_1 : \theta_1 \longmapsto \theta_2 \in \Gamma_1 \bullet \\
\quad\quad\quad\quad\quad \exists \sigma_2 : \theta_3 \longmapsto \theta_4 \in \Gamma_2 \bullet \\
\quad\quad\quad\quad\quad\quad (\sigma_1 \otimes \sigma_2) \wedge (\theta_1 \oplus \theta_3) \wedge (\theta_2 \oplus \theta_4) \wedge (\sigma_1 = \sigma_2)
\end{array}
$$

The relation $(\mathbf{p}, \mathbf{f})$ holds between two components under almost exactly the same conditions as $(\mathbf{p}, \mathbf{l})$ except that the names of the corresponding operations σ_1 and σ_2 are forced to be the same; note that the types may be different in respective components. The relation $(\mathbf{t},\mathbf{l})$ is equivalent to 5 in §3.2 and is defined as follows:

$$\begin{array}{l}
(\Theta_1,\Sigma_1,\Gamma_1)\ (\mathbf{t},\mathbf{l})\ (\Theta_2,\Sigma_2,\Gamma_2)\ \textit{iff} \\
\quad \exists \oplus \subseteq \Theta_1 \times \Theta_2 \bullet dom(\oplus) = \Theta_1 \wedge \\
\quad\quad \exists \otimes \subseteq \Sigma_1 \times \Sigma_2 \bullet dom(\otimes) = \Sigma_1 \wedge \\
\quad\quad\quad ((\theta_1 : \tau_1 \to \tau_2) \otimes (\theta_2 : \tau_3 \to \tau_4) \Rightarrow (\tau_1\ (\mathbf{t},\mathbf{l})\ \tau_3) \wedge (\tau_2\ (\mathbf{t},\mathbf{l})\ \tau_4)) \wedge \\
\quad\quad\quad\quad \forall \theta_1 \in \Theta_1 \bullet \forall \theta_2 \in \Theta_2 \bullet \theta_1 \oplus \theta_2 \Rightarrow \\
\quad\quad\quad\quad\quad \forall \sigma_1 : \theta_1' \longmapsto \theta_3 \in \Gamma_1 \bullet \theta_1' = \theta_1 \Rightarrow \\
\quad\quad\quad\quad\quad\quad \exists \sigma_2 : \theta_2' \longmapsto \theta_4 \in \Gamma_2 \bullet \theta_2' = \theta_2 \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\sigma_1 \otimes \sigma_2) \wedge (\theta_3 \oplus \theta_4)
\end{array}$$

The relation $(\mathbf{t},\mathbf{l})$ holds between two components when the first is totally simulated by the second, *i.e.* *every* operation which is possible in a state of the first component must be possible in *every* corresponding state of the second component. The names of the operations need not be the same in the two components. The relation $(\mathbf{t},\mathbf{f})$ is equivalent to 6 in §3.2 and is defined:

$$\begin{array}{l}
(\Theta_1,\Sigma_1,\Gamma_1)\ (\mathbf{t},\mathbf{f})\ (\Theta_2,\Sigma_2,\Gamma_2)\ \textit{iff} \\
\quad \exists \oplus \subseteq \Theta_1 \times \Theta_2 \bullet dom(\oplus) = \Theta_1 \wedge \\
\quad\quad \exists \otimes \subseteq \Sigma_1 \times \Sigma_2 \bullet dom(\otimes) = \Sigma_1 \wedge \\
\quad\quad\quad ((\theta_1 : \tau_1 \to \tau_2) \otimes (\theta_2 : \tau_3 \to \tau_4) \Rightarrow (\tau_1\ (\mathbf{t},\mathbf{f})\ \tau_3) \wedge (\tau_2\ (\mathbf{t},\mathbf{f})\ \tau_4)) \wedge \\
\quad\quad\quad\quad \forall \theta_1 \in \Theta_1 \bullet \forall \theta_2 \in \Theta_2 \bullet \theta_1 \oplus \theta_2 \Rightarrow \\
\quad\quad\quad\quad\quad \forall \sigma_1 : \theta_1' \longmapsto \theta_3 \in \Gamma_1 \bullet \theta_1' = \theta_1 \Rightarrow \\
\quad\quad\quad\quad\quad\quad \exists \sigma_2 : \theta_2' \longmapsto \theta_4 \in \Gamma_2 \bullet \theta_2' = \theta_2 \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\sigma_1 \otimes \sigma_2) \wedge (\theta_3 \oplus \theta_4) \wedge (\sigma_1 = \sigma_2)
\end{array}$$

is similar to $(\mathbf{t},\mathbf{l})$ except that the names of the operations must be the same in both components.

5 Example development and reuse

This section gives an example development of a factory component and shows how matching supports its subsequent reuse. The required factory may be idle, making chemical x or making chemical y. Both manufacturing processes are mutually exclusive and are supplied with raw chemicals and produce refined chemicals as output. The definition of a chemical is not particularly important, but to be concrete the following is sufficient:

$$\begin{array}{l}
\Theta = \{hot, cold\} \\
\Sigma = \{heat : 7 \to 7, cool : 7 \to 7\} \\
\Gamma = \{heat : cold \longmapsto hot, \\
\qquad heat : hot \longmapsto hot, \\
\qquad cool : cold \longmapsto cold, \\
\qquad cool : hot \longmapsto cold\}
\end{array} \tag{14}$$

We assume that 14 is refined into two distinct component descriptions corresponding to two different types of chemical, these will be referred to as $14x$ and $14y$.

The development of the factory component is presented below. We give the transformations which are used, but skip steps which are repetitive. Starting with 7, the following transformation is applied:

$$splitstate(s, busy, idle) \star deltrans(f : idle \longmapsto idle)$$

to produce a basic process component:

$$\begin{array}{l}
\Theta = \{idle, busy\} \\
\Sigma = \{f : 7 \to 7\} \\
\Gamma = \{f : idle \longmapsto busy, f : busy \longmapsto busy, f : busy \longmapsto idle\}
\end{array} \tag{15}$$

Next, the state *busy* is split into two states which represent manufacturing the two chemicals:

$$splitstate(busy, makex, makey)$$

and the unrequired transitions between manufacturing processes are deleted:

$$deltrans(f : makex \longmapsto makey) \star deltrans(f : makey \longmapsto makex)$$

producing the following component:

$$\begin{aligned}
\Theta &= \{idle, makex, makey\} \\
\Sigma &= \{f : 7 \to 7\} \\
\Gamma &= \{f : idle \longmapsto makex, \\
&\qquad f : idle \longmapsto makey, \\
&\qquad f : makex \longmapsto makex, \\
&\qquad f : makex \longmapsto idle, \\
&\qquad f : makey \longmapsto makey, \\
&\qquad f : makey \longmapsto idle\}
\end{aligned} \tag{16}$$

Next, a vertical transformation is applied to the single operator f, identifying three operators at a lower level of abstraction:

$$splitop(f, process, startstop) \star splitop(startstop, start, stop)$$

producing many unrequired transitions which are deleted:

$$deltrans(start : makex \longmapsto idle) \star deltrans(stop : idle \longmapsto makex) \star \ldots$$

the result is as follows:

$$\begin{aligned}
\Theta &= \{idle, makex, makey\} \\
\Sigma &= \{start : 7 \to 7, process : 7 \to 7, stop : 7 \to 7\} \\
\Gamma &= \{start : idle \longmapsto makex, \\
&\qquad start : idle \longmapsto makey, \\
&\qquad stop : makex \longmapsto idle, \\
&\qquad stop : makey \longmapsto idle, \\
&\qquad process : makex \longmapsto makex, \\
&\qquad process : makey \longmapsto makey\}
\end{aligned} \tag{17}$$

Next, the manufacturing process states are split to identify start and end states:

$$splitstate(makex, initx, termx) \star splitstate(makey, inity, termy)$$

the starting and stopping operators are restricted to apply to the appropriate states:

$$deltrans(start : idle \longmapsto termx) \star deltrans(stop : initx \longmapsto idle) \star \ldots$$

and the processing operator is restricted:

$$deltrans(process : initx \longmapsto termx) \star deltrans(process : termx \longmapsto initx) \star \ldots$$

producing the following component:

$$\begin{aligned}
\Theta &= \{idle, initx, termx, inity, termy\} \\
\Sigma &= \{start : 7 \to 7, process : 7 \to 7, stop : 7 \to 7\} \\
\Gamma &= \{start : idle \longmapsto initx, \\
&\qquad start : idle \longmapsto inity, \\
&\qquad stop : termx \longmapsto idle, \\
&\qquad stop : termy \longmapsto idle, \\
&\qquad process : initx \longmapsto termx, \\
&\qquad process : inity \longmapsto termy\}
\end{aligned} \tag{18}$$

Both the *start* and *stop* operators are refined in order to restrict their respective domain and range to 14:

$$refineop(start, 7 \rightarrow 7, 14 \rightarrow 7) \star refineop(stop, 7 \rightarrow 7, 7 \rightarrow 14)$$

producing the component: description for a factory:

$$\begin{aligned}
\Theta &= \{idle, initx, termx, inity, termy\} \\
\Sigma &= \{start : 14 \rightarrow 7, process : 7 \rightarrow 7, stop : 7 \rightarrow 14\} \\
\Gamma &= \{start : idle \longmapsto initx, \\
&\quad start : idle \longmapsto inity, \\
&\quad stop : termx \longmapsto idle, \\
&\quad stop : termy \longmapsto idle, \\
&\quad process : initx \longmapsto termx, \\
&\quad process : inity \longmapsto termy\}
\end{aligned} \tag{19}$$

After further refinement, the following factory component description is produced:

$$\begin{aligned}
\Theta &= \{idle, initx, termx, inity, termy\} \\
\Sigma &= \{startx : 14x \rightarrow 7, \\
&\quad starty : 14y \rightarrow 7, \\
&\quad processx : 7 \rightarrow 7, \\
&\quad processy : 7 \rightarrow 7, \\
&\quad stopx : 7 \rightarrow 14x, \\
&\quad stopy : 7 \rightarrow 14y\} \\
\Gamma &= \{startx : idle \longmapsto initx, \\
&\quad starty : idle \longmapsto inity, \\
&\quad stopx : termx \longmapsto idle, \\
&\quad stopy : termy \longmapsto idle, \\
&\quad processx : initx \longmapsto termx, \\
&\quad processy : inity \longmapsto termy\}
\end{aligned} \tag{20}$$

To complete the software development process, the component would be implemented in a programming language, perhaps C++ [19] or Java [20], and the component would be associated with its program in a software library.

To show how reuse can be achieved using the simulation relations described in §4.3, the factory component is re-developed. We will use the relation $(\mathbf{p}, \mathbf{l})$ to search a library of component descriptions which contains 15 – 19.

The development starts with 7. The following transformation is applied to produce an initial factory description:

$$splitstate(s, on, off) \star deltrans(f : off \longmapsto off)$$

which is

$$\begin{aligned}
\Theta &= \{on, off\} \\
\Sigma &= \{f : 7 \rightarrow 7\} \\
\Gamma &= \{f : off \longmapsto on, f : on \longmapsto on, f : on \longmapsto off\}
\end{aligned} \tag{21}$$

Notice that this is essentially the same as 15 except that the names of the states have been changed. If the library is searched using this component description then many library components will match, so it is necessary to develop the component further. The operator f is split into two operators:

$$splitop(f, start, stop)$$

and the unrequired transitions are removed:

$$deltrans(start : on \longmapsto off) \star \ldots$$

producing:

$$\begin{aligned}
\Theta &= \{on, off\} \\
\Sigma &= \{start : 7 \rightarrow 7, stop : 7 \rightarrow 7\} \\
\Gamma &= \{start : off \longmapsto on, off : on \rightarrow off\}
\end{aligned} \tag{22}$$

Finally, the types of the operators are refined:

$$refineop(start, 7 \rightarrow 7, 14 \rightarrow 7) \star refineop(stop, 7 \rightarrow 7, 7 \rightarrow 14)$$

which produces:

$$\begin{aligned}
\Theta &= \{on, off\} \\
\Sigma &= \{start : 14 \rightarrow 7, stop : 7 \rightarrow 14\} \\
\Gamma &= \{start : off \longmapsto on, off : on \rightarrow off\}
\end{aligned} \tag{23}$$

Two components match the component 23, these are 19 and 20. The relations which match 20 are:

$$\begin{aligned}
\otimes = \{&(start : 14 \rightarrow 7, startx : 14x \rightarrow 7), \\
&(start : chemical \rightarrow 7, starty : 14y \rightarrow 7), \\
&(stop : 7 \rightarrow 14, stopx : 7 \rightarrow 14x), \\
&(stop : 7 \rightarrow 14, stopy : 7 \rightarrow 14y)\}
\end{aligned}$$

$$\oplus = \{(off, idle), (on, initx), (on, termx), (on, inity), (on, termy)\}$$

The development of a factory software component has been shown to take advantage reuse with respect to the proposed ROO model of development and reuse. The number of steps which are taken to effectively reuse a component, 21 – 23, is significantly shorter that the number of steps taken to develop the component in the first place, 15 – 20 (which includes many steps which were elided, including writing the code!).

6 Analysis and conclusions

The aims of this work were to produce a model of software development which supported the object-oriented programming model, was executable, had a formal semantics and supported reuse. The work which is described in this paper has set the foundations of such a model. The model has been implemented as a prototype tool, called ROOT, in the programming language Scheme [10]. ROOT is text based and supports the development and simulations of component descriptions through a menu driven interface.

The work is novel since it attempts to provide a formal model for object-oriented systems which encompasses both development and reuse. However, the model is very simple are will require further refinement before it can achieve its aim of supporting object-oriented development and reuse in Java with respect to the Internet.

A particular area for refinement is that of simulation and matching. The relations in 4.3 and the ROOT algorithms which test them are satisfactory for simple component descriptions, but are unlikely to be effective when dealing with large complex components. They do not prioritize possible matches, for example in terms of component coverage. The matching mechanisms are likely to require human input to help them navigate through a space of possible matches. The distinction between exact operator name matching and arbitrary operator matching is an example of human control. This is likely to be a fruitful area of research.

This paper deals only with machine simulation where machines must execute in step. If we allow a single step in one machine to be simulated by a sequence of steps in another machine then this allows greater flexibility when matching component descriptions. However, this introduces a collection of problems in deciding whether or not one machine simulates another. This is an area for further research.

Another area for refinement is the expressivity of the language used to represent components. Conditional transitions are likely to be necessary and a distinction made between deterministic and non-deterministic components. Currently, states are atomic and have no internal structure; it is likely that, at lower levels of abstraction, some form of record structure for states is desirable.

The language should support object-oriented features such a self reference and inheritance. Inheritance may be modelled by merging two or more machines together. A problem which is encountered in object-oriented type systems is the difference between co- and contra-variance. This also occurs in ROO where the type of one operator is compared to another. This is an area for further investigation.

Ideally, the component development process should lead to programs written in a particular language. We intend to extend the translation operations to allow components to be refined Java code.

The work in this paper is part of ongoing collaborative research, see [8] for more details.

References

[1] "Formal Specification of Reusable Interface Objects", P. Alencar, D. Cowan, C. Lucena and L. Nova, in *Proc. ACM SIGSOFT Symposium on Software Reusability*, April 1995.

[2] "Designing an Object-Oriented Programming Language with Behavioural Subtyping.", P. America, LNCS 489 Proc. Rex/Fool Conf. May/June 1990.

[3] "Web crawlers to index Java", D. Andrews, *Byte* 21(4), April 1996, p26.

[4] "Software Reusability and the Internet", G. Arango, in *Proc. ACM SIGSOFT Symposium on Software Reusability*, April 1995.

[5] *Finite Transition Systems*, A. Arnold, Prentice Hall International Series in Computer Science, 1994.

[6] *Essays on Object-Oriented Software Engineering*, Vol. 1, E. Berard, Prentice Hall International, Englewood Cliffs, NJ, 1993.

[7] *Object-Oriented Analysis and Design with Applications*, G. Booch, 2nd ed., The Benjamin/Cummings Publishing Company Inc., 1994.

[8] "Using Behavioural Object Descriptions to Reuse Java Code Over the Internet", A. Clark and I. Palmer, submitted to 3rd International Conference on Object-Oriented Information Systems, OOIS96.

[9] "A Formal Basis for the Refinement of Rule Based Transition Systems.", A. Clark. The Journal of Functional Programming 6(2), 1996.

[10] *Revised Report on the Algorithmic Language Scheme*, W. Clinger and J. Rees (eds), November 1991.

[11] *Object-Oriented Development – The Fusion Method*, D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes, Prentice Hall International, Englewood Cliffs, NJ, 1994.

[12] "Research Directions in Software Reuse", H. Gall, M. Jazayeri and R. Klosch, in *Proc. ACM SIGSOFT Symposium on Software Reusability*, April 1995.

[13] "Reusing and Interconnecting Software Components", J. Goguen, *IEEE Computer*, February, 1986.

[14] "Melding Software Systems for Reusable Building Blocks", G. Kaiser and D. Garlan, *IEEE Software* 4(4), 1987.

[15] *Object-Oriented Methods – A Foundation*, J. Martin and J. Odell, Prentice Hall International, Englewood Cliffs, NJ, 1995.

[16] "Reusing Software: Issues and Research Directions", H. Mili, F. Mili and A. Mili, *IEEE Trans. on Software Engineering*, 21(6), June 1995.

[17] *Object-Oriented Modelling and Design*, J. Runbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Prentice Hall International, Englewood Cliffs, NJ, 1991.

[18] *Object Lifecycles: Modeling the World in States*, S. Shlaer and S. Mellor, Yourdon Press: Prentice Hall International, Englewood Cliffs, NJ, 1992.

[19] *The Annotated C++ Reference Manual*, B. Stroustroup and M. Ellis, Addison Wesley. 1990.

[20] *The Java Language Specification*, Sun Microsystems Inc., 1995.

[21] "Software Component Interface Description for Reuse", B. Whittle and M. Ratcliffe, *IEE BCS Software Engineering Journal*, 8(6), November 1993.

[22] "Models and Languages for Component Description and Reuse", B. Whittle, *ACM SIGSOFT Software Engineering Notes*, 20(2), April 1995.

[23] "Specification Matching of Software Components", A. Zaremski and J. Wing, in *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.*

[24] "Signature Matching: A Tool for Using Software Libraries", A. Zaremski and J. Wing, *ACM Trans. on Software Engineering and Methodology*, 4(2), 1995.