Chapter 3

An Object Oriented Semantic Framework

3.1 An Overview of the Semantic Framework

The semantic framework, developed in this chapter, connects together the formality and high levels of expressibility of the ADT ACT ONE and our informal understanding of object oriented models, relationships and concepts. At this point in the thesis, our object oriented framework of understanding is dependent on the example object oriented behaviours previously given in chapter 2, together with our own informal conceptualisation of the object oriented paradigm. The object oriented semantic framework is developed to provide a formal model of object oriented concepts which can be used during object oriented analysis and requirements capture. Rather than defining object oriented concepts directly in ACT ONE, a more general approach is proposed in which a new object oriented semantics is developed. This semantics provides a more abstract model which can be implemented by more concrete models. This chapter defines such a semantics and uses the ADT ACT ONE to provide an executable model for the more abstract specifications. The structure of the remainder of this chapter is as follows:

- **Section 3.2: Object-Labelled State Transition System (O-LSTS) Semantics**
  In this section, the semantic model is defined as a particular kind of labelled state transition system (called an O-LSTS[1]). It seems natural to conceptualise the dynamic behaviour of an object as a sequence of states which the object can attain. The state transitions result from the object servicing requests at its external interface. This simple view is expanded upon to encompass the notion of class and relations between classes. A class is defined as a collection of object behaviours which form a set of states which are encapsulated within a common interface. An O-LSTS specification formally defines this notion. A diagramatic representation of an O-LSTS (an O-LSTSD) is introduced as an equivalent way of expressing the information in an O-LSTS specification.

[1]Object-Labelled State Transition System.

- **Section 3.3: An Object Oriented Interpretation of the O-LSTS Model**
  Section 3.3 defines a mapping between the O-LSTS model and the object oriented paradigm. The informal notions of class, object, attribute, service and the relationships between them are given a formal interpretation in our O-LSTS semantics. These definitions add much needed precision to the object oriented terminology. In particular, we formalise two different types of hierarchical model:

  - **Classification model:** the subclassing relationships between all classes in a system are represented in a class hierarchy diagram.
  - **Compositional model:** the structure of an object (in terms of its component parts) is represented in a structure diagram.

  These hierarchical diagrams are used to complement O-LSTSDs.
- **Section 3.4: OO ACT ONE: A Formal Object Oriented Analysis Language**
  This section defines a concrete syntax for the specification of O-LSTSs during analysis and requirments capture. The O-LSTS model is defined in terms of the well understood mathematical notions of sets, cartesian products, relationships and functions. It is necessary to wrap these constructs in a more 'friendly' syntactically sugared syntax. The syntax we define for this purpose is similar to ACT ONE, with a distinctly object oriented flavour: we call it object oriented ACT ONE (OO ACT ONE). Explicit object oriented mechanisms for re-using predefined classes of behaviour (O-LSTSs) are defined. These mechanisms facilitate the definition of different types of subclassing, composition and parameterised classes. Other mechanisms allow the definition of invariant properties, the hiding of internal behaviour and the specification of exceptions. Such a concrete syntax is also necessary when we consider the problem of statically analysing an O-LSTS specification.
- **Section 3.5: An ACT ONE Execution Model for O-LSTS Specifications**
  Section 3.5 provides a mapping from the O-LSTS semantics, as defined in an OO ACT ONE specification, to ACT ONE. This mapping formalises the relationship between object oriented terminology and ADT concepts (for example, type, sort, operation and equation). It should be emphasised that, although ACT ONE specifications can be used to model object oriented requirements, not all ACT ONE specifications have a meaningful object oriented interpretation. The ACT ONE which is produced from OO ACT ONE is used to provide the basis for a static analysis of the typing properties of an OO ACT ONE specification. Furthermore, the ACT ONE provides an 'executable' model for testing the dynamic behaviour of objects defined in OO ACT ONE. The exact nature of this execution model is made clearer in section 3.5.5.

3.2 Object-Labelled State Transition System (O-LSTS) Semantics

The semantic framework is based around the definition of a particular kind of labelled state transition system, namely an Object-LSTS (O-LSTS). It is defined as follows.

3.2.1 Definition: an O-LSTS Specification

An O-LSTS, C_0 say, is a 7-tuple $< O, UTT, HUTT, VTT, HVTT, USS, VSS >$ defined in an environment of O-LSTSs, Env_{C_0}. These eight separate elements of an O-LSTS are formally defined in sections 3.2.1.1 to 3.2.1.8, below.

3.2.1.1 The Environment

Env_{C_0} is specified as a 2-tuple $< C', Rel_{C_0} >$ where,

- C' is a possibly empty set of predefined O-LSTSs $\{C_1, \ldots, C_k\}$, say. We say that C_0 **uses** $C_i, \forall i \in \{1, \ldots, k\}$.
  Definition: Visible Class Set
  The **visible class set** of C_0, written $visible(C_0)$, $= \{C_0\} \cup C' \cup_{i=1}^{k} visible(C_i)$.
- Rel_{C_0} is a set of O-LSTS pairs $\{< C_i, C_j >| \ C_i \sqsubseteq C_j$ and $C_i, C_j \in visible(C_0)\}$.
  Rel_{C_0} represents all explicitly defined subclassing relationships between O-LSTSs visible in C_0. The relationship $\sqsubseteq$ can be defined in many ways[2]. The particular relation which we chose is given in section 3.3.3. Explicit subclassing[3] is reflexive and transitive.
- $\forall < C_i, C_j > \in Rel_{C_k}$, if $C_k \in visible(C_0)$ then $< C_i, C_j > \in Rel_{C_0}$. In other words, a class 'inherits' the subclassing relationships which are defined in the environments of the classes which are visible to it.

3.2.1.2 The Class Members

O is a nonempty set of **typed state labels** $\{O_1, \ldots, O_n\}, n \in \{1, 2, \ldots\}$, called the **typed state set**. Each **typed state label** is either unparameterised, parameterised or conditionally parameterised:

- unparameterised, written as *state-constructor*[4]
- parameterised, written as *state-constructor*$(P_1, \ldots, P_n)$ for $n \in \{1, 2, \ldots\}$ where,
  - i) $\forall P_i \in \{P_1, \ldots, P_n\}$, $P_i \in visible(C_0)$
  - ii) given $S_1(P_1, \ldots, P_r), S_2(Q_1, \ldots, Q_s) \in O$, then $S_1 = S_2 \Rightarrow r = s$ and $P_i = Q_i, \forall i \in \{1, \ldots, r\}$

  Definition: Parameter Classes:
  $P_1, \ldots, P_n$ are called the **parameter classes** of the *state-constructor*.
- conditionally parameterised, written as *state-constructor*$(P_1, \ldots, P_n)$ **on cond**$(P_1, \ldots, P_n)$ for $n \in \{1, 2, \ldots\}$ where,

[2] The O-LSTS model can be said to define a generic formal object oriented framework which is parameterised on the subclassing relationship.

[3] Explicit subclassing relationships are defined with respect to a class environment. We say that $C_i \sqsubseteq C_j$ in Env_{C_0} when C_i is explicitly defined as a subclass of C_j in the environment of C_0.

[4] *State-constructors* are represented as strings of characters — the exact syntax is defined in 3.2.2.1. Conventionally, all other string identifiers, in the O-LSTS definition, are represented in italics.

- i) $\forall P_i \in \{P_1, \ldots, P_n\}$, $P_i \in visible(C_0)$
- ii) given $S_1(P_1, \ldots, P_r), S_2(q_1 : Q_1, \ldots, q_s : Q_s) \in O$, then $S_1 = S_2 \Rightarrow r = s$ and $P_i = Q_i, \forall i \in \{1, \ldots, r\}$
- iii) **cond** is a boolean expression defined on the parameters of the state constructor[5].

Additionally, implicit in every O-LSTS, C_0 say, is an **unspecified state label** which is represented by the **unparameterised** *state-constructor*, written $\sim C_0$.
Definition: *States* function
$States(< O, UTT, HUTT, VTT, VUTT, USS, VSS >) = O$, or $States(C) = O_C$[6].
Definition: Untyped State Set
The **untyped state set** of C_0, written $US(C_0)$, is generated from the **typed state set** as follows:

$US(C_0) = RemoveTypes(States(C_0))$, and

$RemoveTypes(\{o_1, \ldots, o_n\}) = NoType(o_1) \cup \ldots \cup NoType(o_n)$, and

$NoType(state\text{-}constructor) = \{state\text{-}constructor\}$
$NoType(state\text{-}constructor(P_1, \ldots, P_n)) =$
$\{state\text{-}constructor(p_1, \ldots, p_n) \mid p_i \in US(P_i), \forall i \in \{1, \ldots, n\}\}$, and
$NoType(state\text{-}constructor(P_1, \ldots, P_n)\mathbf{on\ cond}(P_1, \ldots P_n)) =$
$\{state\text{-}constructor(p_1, \ldots, p_n) \mid \mathbf{cond}(p_1, \ldots p_n)$ and $p_i \in US(P_i), \forall i \in \{1, \ldots, n\}\}$

The elements of the **untyped state set** are called the **state labels**. Consequently, the **untyped state set** is also known as the **state label set**.

3.2.1.3 External Transformer Interface

UTT is a possibly empty set of **unvalued typed transitions**, called the **unvalued typed transition set**.
$\forall ut \in UTT$, ut is either:

- (i) an **unvalued unparameterised typed transition** of C_0 written as *transition-name*
- (ii) an **unvalued parameterised typed transition** of C_0, written as $transition\text{-}name< U_1, \ldots, U_r >$, such that $< U_1, \ldots, U_r > \in (visible(C_0))^r$
  The **parameter tuple** of $transition\text{-}name< U_1, \ldots, U_r >$ is defined to be $< U_1, \ldots, U_r >$.

Given an **unvalued typed transition set**, it is necessary to generate the set of all unvalued transitions through an actualisation of all possible combinations of **parameter tuple** values. The set generated is defined as follows.
Definition: Unvalued Actualised Transition Set

[5] The syntax and semantics of boolean expressions is defined by a **state label expression** with **type** boolean — see 3.2.3.

[6] Similarly, $UTT_C, HUTT_C, VTT_C, HVTT_C, USS_C, VSS_C$ represent the 2nd, 3rd, 4th, 5th, 6th and 7th elements of the 7-tuple O-LSTS C.

$UAT(UTT) = RemoveUParameters(UTT)$ where,

$RemoveUParameters(\{\}) = \{\}$, and
$RemoveUParameters(\{ut_1, \ldots, ut_n\})$
$= ActUParameters(ut_1) \cup \ldots \cup ActUParameters(ut_n)$, where

$ActUParameters(\textit{transition-name}) = \{\textit{transition-name}\}$
$ActUParameters(\textit{transition-name}< U_1, \ldots, U_n >) =$
$\{ \textit{transition-name}(u_1, \ldots, u_n) \mid u_i \in U_i, \forall i \in \{1, \ldots, n\}\}$.

3.2.1.4 Hidden Transformers

$HUTT$ is a subset of UTT called the **hidden unvalued typed transition set**. We define an **unhidden unvalued typed transition** to be any member of UTT which is not a member of $HUTT$.

3.2.1.5 External Accesors Interface

VTT is a possibly empty set of valued typed transitions, called the **valued typed transition set**. $\forall vt \in VTT$ vt is either:

- (i) a **valued unparameterised typed transition** of C_0, written as *transition-name*:V_{vt}, where: $V_{vt} \in visible(C_0)$ is called the **result type** of the transition
- (ii) a **valued parameterised typed transition** of C_0, written as *transition name*$< U_1, \ldots, U_r >$: V_{vt}, such that $V_{vt} \in visible(C_0)$ and $< U_1, \ldots, U_r > \in (visible(C_0))^r$
  The **parameter tuple** of *transition-name*$< U_1, \ldots, U_r >$:V_{vt} is defined to be $< U_1, \ldots, U_r >$.

Given a **valued typed transition set**, it is necessary to generate the set of all valued transitions through an actualisation of **parameter tuple** values. The set generated is defined as follows.

Definition: Valued Actualised Transition Set

$VAT(VTT) = RemoveVParameters(VTT)$, where

$RemoveVParameters(\{\}) = \{\}$ and
$RemoveVParameters(\{ut_1, \ldots, ut_n\})$, for $n \in \{1, 2, \ldots\}$,
$= ActVParameters(ut_1) \cup \ldots \cup ActVParameters(ut_n)$, where

$ActVParameters(\textit{transition-name}\text{:V}) = \{\textit{transition-name}\}$
$ActVParameters(\textit{transition-name}< U_1, \ldots, U_n >: V) =$
$\{ \textit{transition-name}(u_1, \ldots, u_n) \mid u_i \in U_i, \forall i \in \{1, \ldots, n\}\}$.

The **result type** of a **valued actualised transition** is defined to be the **result type** of the **valued typed transition** from which it was generated.

3.2.1.6 Hidden Accessors

$HVTT$ is a subset of VTT, called the **hidden valued typed transition set**. We define an **unhidden valued typed transition** to be any member of VTT which is not a member of $HVTT$.

3.2.1.7 Transformer Behaviour

USS is a possibly empty set of **unvalued state-to-state transitions** $\{From_{O_j} : O_j \in US(C_0)\}$, one, and only one, for every $O_j \in US(C_0)$, where $From_{O_j} \subseteq UAT(UTT) \times US(C_0)$.
Now, $\forall O_j \in US(C_0)$ the following **completeness conditions** must be upheld:

- (i) given $ul \in UAT(UTT)$, $\exists O_k \in US(C_0)$ such that $< ul, O_k > \in From_{O_j}$
- (ii) given $< ul1, O_1 > \in From_{O_j}$ and $< ul2, O_2 > \in From_{O_j}$, $ul1 = ul2 \Rightarrow O_1 = O_2$

The **unvalued state-to-state transitions** from the unspecified state $\sim C_0$ do not have to be explicitly defined. Unless otherwise specified, $< ul, \sim C_0 > \in From_{\sim C_0}, \forall ul \in UAT(UTT)$.

3.2.1.8 Accessor Behaviour

VSS is a possibly empty set of **valued state-to-state transitions** $\{Valfrom_{O_j} : O_j \in O\}$, one, and only one, for every $O_j \in US(C_0)$, where $Valfrom_{O_j} \subseteq VAT(VTT) \times visible(C_0) \times US(C_0)$
Now, $\forall O_j \in US(C_0)$ the following **completeness conditions** must be upheld:

- (i) given $vl \in VAT(VTT)$ with **result type** V, $\exists O_k \in US(C_0)$ and $res \in US(V)$ such that $< vl, res, O_k > \in Valfrom_{O_j}$
- (iii) given $< vl1, res1, O_1 >, < vl2, res2, O_2 > \in Valfrom_{O_j}$, $vl1 = vl2 \Rightarrow res1 = res2$ and $O_1 = O_2$

The **valued state-to-state transitions** from the unspecified state $\sim C_0$ do not have to be explicitly defined. Unless otherwise specified, $< vl, \sim C', \sim C_0 > \in ValFrom_{\sim C_0}, \forall vl \in VAT(VTT)$, where C' is the **result type** of the **valued typed transition** vl.

3.2.1.9 Additional Syntactic Constraints

The following additional syntactic constraints are defined to enable **state labels** and **typed transitions** to be uniquely identified. They also make the O-LSTS models easier to translate to ACT ONE (see section 3.5). The additional constraints are as follows:

- *State-Constructors* must be uniquely defined as being unparameterised, parameterised or conditionally parameterised.
- All *state-constructors* and *transition-names* are uniquely defined in each O-LSTS by strings of characters. These strings can include alphanumeric characters (and the '-' character for constructing structured strings). The final character must not be a digit.
- *Transition-names* must not correspond to *state-constructors*.
- **State labels** in the USS and VSS tuples must be represented as **state label expressions** (see 3.2.3).

3.2.2 O-LSTS Examples

To clarify the formal definition, some examples follow. These examples do not illustrate every aspect of the O-LSTS model. In particular they do not show the significance of the subclass hierarchy defined in the environment of an O-LSTS. The examples specify purely object based systems. We examine the specification of object oriented systems after defining subclassing in 3.3.3.

O-LSTS Example 1: Resetable-Traffic-Light

The environment of the O-LSTS Resetable-Traffic-Light is defined to be $< C', \{\} >$, where $C' = \{$ Bool, traffic-light-colour$\}$, $States$(Bool) $= \{true, false\}$ and $States$(traffic-light-colour) $= \{red, green, am$
Resetable-Traffic-Light $= < O, UTT, HUTT, VTT, HVTT, USS, VSS >$, where

- $O = \{t-l(traffic-light-colour)\}$
  (Consequently, $US(Resetable-Traffic-Light) = \{t-l(red), t-l(green), t-l(amber)\}$)
- $UTT = \{next, reset < traffic-light-colour >\}$
  Consequently, $UAT(UTT) = \{next, reset(red), reset(green), reset(amber)\}$
- $HUTT = \{\}$
- $VTT = \{safe-to-go$:Bool, eq <Resetable-Traffic-Light>: Bool$\}$
  Consequently, $VAT(VTT) = \{safe{-}to{-}go$:Bool, $eq(t{-}l(red))$:Bool, $eq(t{-}l(amber))$:Bool, $eq(t{-}l(green))$:Bool$\}$.
- $HVTT = \{\}$
- $USS = \{From_{O_j} : O_j \in O\}$ where
  - $From_{t-l(red)} = \{< next, t-l(green) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
  - $From_{t-l(amber)} = \{< next, t-l(red) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
  - $From_{t-l(green)} = \{< next, t-l(amber) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
- $VSS = \{ValFrom_{O_j} : O_j\}$ where
  - $ValFrom_{t-l(green)} = \{< safe-to-go, true, t-l(green) >, < eq(t-l(red)), false, t-l(green) >, < eq(t-l(amber)), false, t-l(green) >, < eq(t-l(green)) true, t-l(green) >\}$
  - $ValFrom_{t-l(red)} = \{< safe-to-go, false, t-l(red) >, < eq(t-l(green)), false, t-l(red) >, < eq(t-l(amber)), false, t-l(red) >, < eq(t-l(red)), true, t-l(red) >\}$
  - $ValFrom_{t-l(amber)} = \{< safe-to-go, true, t-l(amber) >, < eq(t-l(green)), false, t-l(amber) >, < eq(t-l(amber)), true, t-l(amber) >, < eq(t-l(red)), false, t-l(amber) >\}$

This 7-tuple is a valid O-LSTS, since it fulfils all the necessary and sufficient conditions of the definition. It is represented by the O-LSTS Diagram (O-LSTSD) in figure 3.1.

The O-LSTSD is a rectangle containing a graph of labelled nodes and links. The class name is given above the rectangle and the environment of the class is defined by the list of O-LSTSs following the `using` keyword and `NO HIERARCHY` specifies that there are no explicit class relationships to consider. Each node in the diagram contains a unique **state label**. All **state labels** in the O-LSTS are represented by nodes. The set of transitions between nodes (represented by the set of links) is isomorphic to the union of the **state-to-state transition sets**. In other words, $\forall < ua, O_k > \in From_{O_j}$, $\exists$ a unique *Node-LabelledLink-Node* connection, in the O-LSTSD, from O_j to O_k. Similarly, $\forall < va, val, O_l > \in Valfrom_{O_m}$, $\exists$ a unique *Node-LabelledLink-Node* connection, in the O-LSTSD,

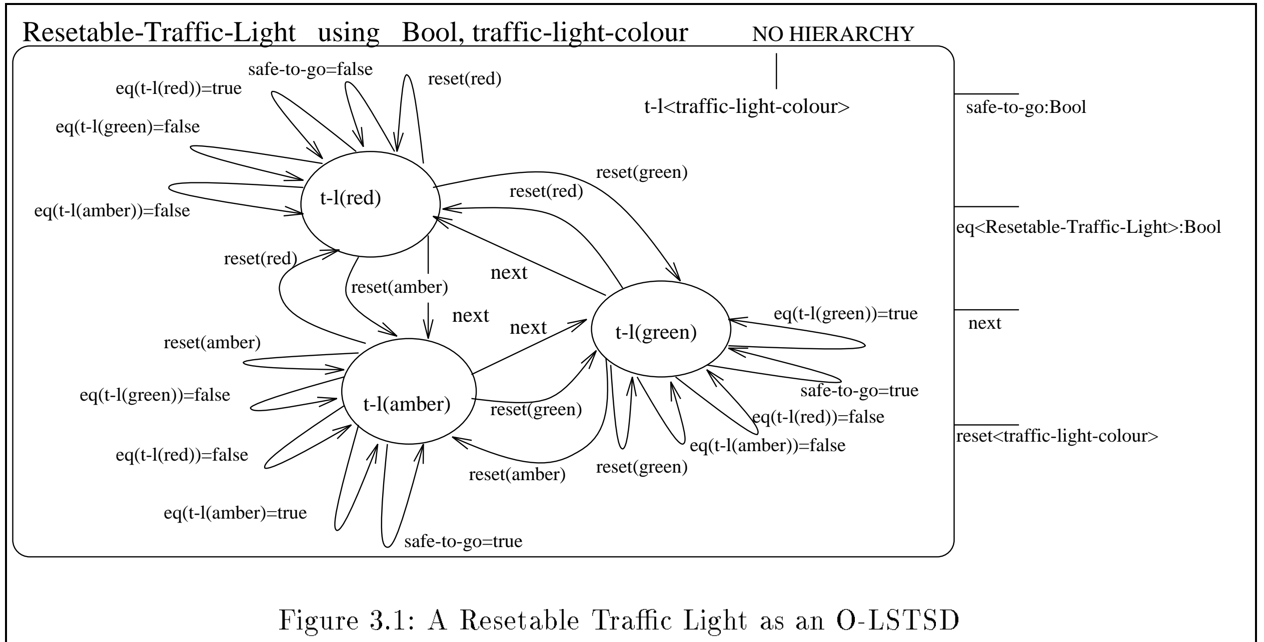
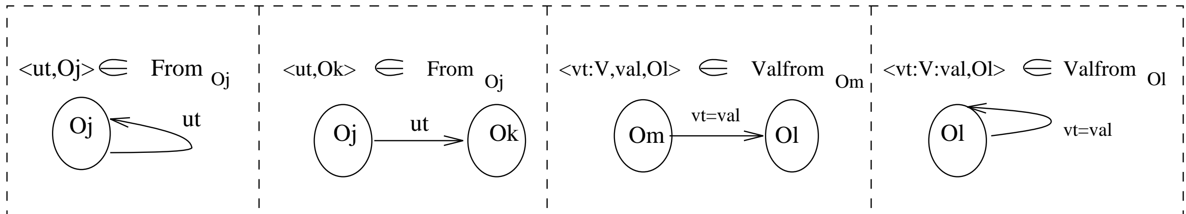


Figure 3.1: A Resetable Traffic Light as an O-LSTSD

from O_l to O_m. (When $O_j = O_k$, or $O_l = O_m$, the link connects the node with itself.) These four cases are illustrated in the diagram below:



The (valued and unvalued) **typed transition sets** are represented by connections to the outside of the rectangle around the O-LSTSD. The **hidden** transitions (there are none in this system) must be identified by appending the transition name with the label `HIDDEN`. Similarly, the **state label types** are represented by connections on the inside of the rectangle, together with the conditions placed on the parameters (if there are any).

Diagramatic Syntactic Sugaring

As even simple O-LSTSDs get very cluttered with nodes and links, there are a number of extensions which can be used to sugar the diagramatic representation. In the example above, an obvious extension is to parameterise the result of the reset transition from each node. This is done in figure 3.2. The dotted link represents a set of transitions (one for each value the parameter can take, i.e. one for every member of the **state label set** of the **parameter type**). The result of a parameterised transition is a parameterised node (a node with a dotted circumference whose **state label** is also parameterised). All parameters in an O-LSTSD are represented between diamond brackets.

It can be seen from figure 3.2 that the parameterised resets are the same for each node. A further sugaring permits the parameterisation of the node labels at both ends of a transition. The result

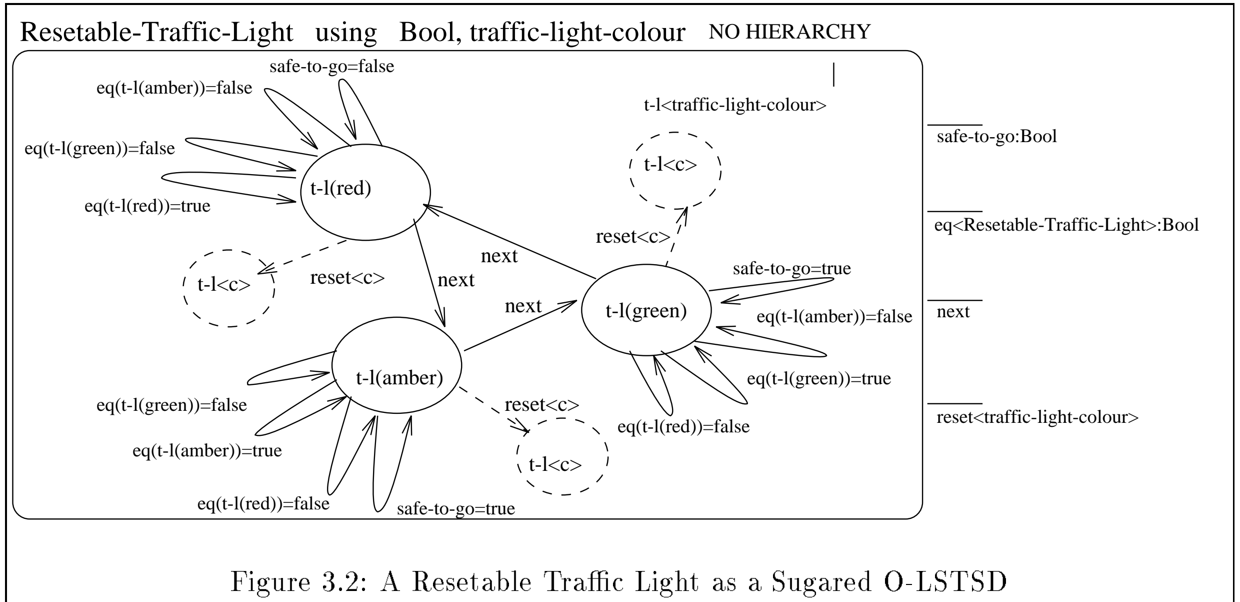


Figure 3.2: A Resetable Traffic Light as a Sugared O-LSTSD

of a transition is a parameterised expression (also in diamond brackets), [7] which is dependent on the transition parameter values and the **state label** parameter values of the node from which the transition is taking place. Two examples of such a parameterisation are illustrated in figure 3.3.

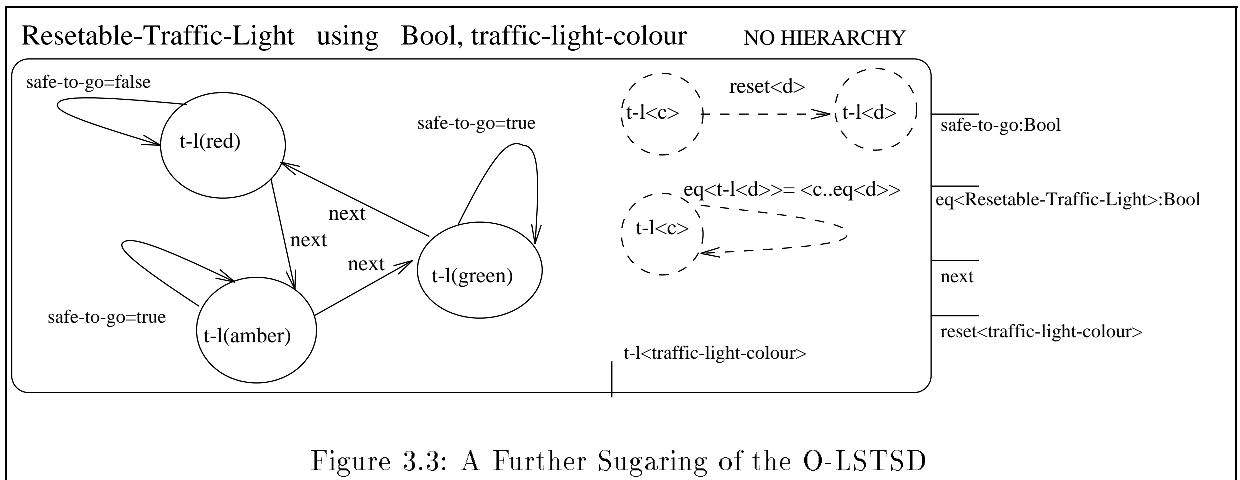


Figure 3.3: A Further Sugaring of the O-LSTSD

O-LSTS Example 2: An Integer Counter

The resetable traffic light example illustrates the specification of a system with a finite number of states in which it is not necessary to parameterise the behaviour definition, although the parameterisation does simplify and clarify the specification. It is necessary to provide facility for defining O-LSTSs in a parameterised fashion. An unbounded integer counter, for example, cannot be represented by a finite state machine and so we must provide a suitable means of defining an infinite set of behaviours

[7] The syntax and semantics of such **state label expressions** is formalised later in this section. For now, the interpretation is informal, but intuitive and well explained by the examples.

in a parameterised form. Consider the O-LSTSD in figure 3.4.

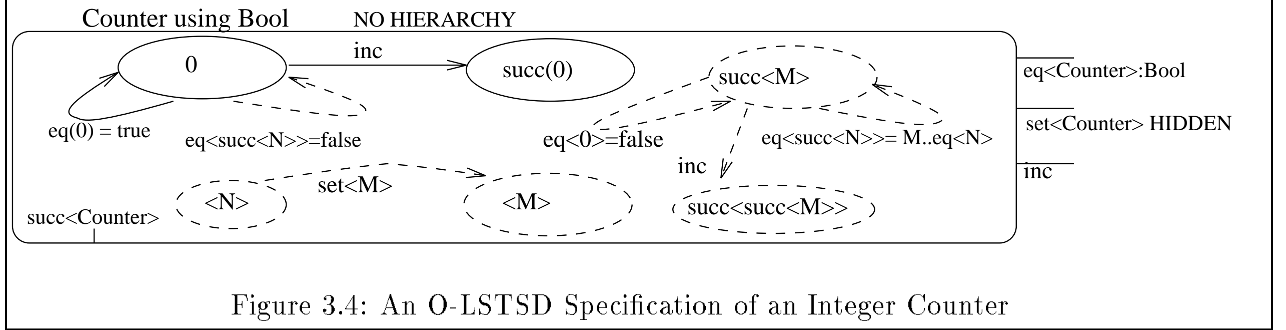


Figure 3.4: An O-LSTSD Specification of an Integer Counter

The following should be noted:

- The connection on the inside of the diagram is labelled by the expression $succ < Counter >$. This specifies that $\forall n \in US(Counter), succ(n) \in US(Counter)$.
- The inc transition from 0 to $succ(0)$ specifies that $< inc, succ(0) > \in From_0$.
- The $eq(0) = true$ transition from 0 to 0 specifies that $< eq(0), true, 0 > \in Valfrom_0$.
- The parameterised transition $eq(succ < N >) = false$ from 0 to 0 specifies that $\forall N \in US(Counter)$,
  $< eq(succ(N)), false, 0) \in ValFrom_0$.
- The parameterised transition $eq(0)$ from $succ < M >$ back to itself specifies that $\forall M \in US(Counter)$,
  $< eq(0), false, succ(M) > \in ValFrom_{succ(M)}$.
- The parameterised transition inc from $succ < M >$ to $succ < succ < M >>$ specifies that $\forall M \in US(Counter)$,
  $< inc, succ(succ(M)) > \in From_{succ(M)}$.
- The parameterised transition $eq < succ < N >>= M..eq(N)$ from $succ < M >$ back to itself specifies that $\forall N, M \in US(Counter), < eq(succ(N)), M..eq(N), succ(M) > \in From_{succ(M)}$. (In this case $M..eq(N)$ is a **state label expression** which represents a **state label** in the **untyped state set** of Bool The meaning of such a **state label expression** is defined in the next section (3.2.3)).
- The parameterised transition $set < M >$ from N to M specifies that $\forall N, M \in US(Counter)$,
  $< set < M >, M > \in From_N$.

The O-LSTSD is equivalent to the OLSTS specification:
$Counter = < O, UTT, HUTT, VTT, HVTT, USS, VSS >$ in $< \{Bool\}, \{\} >$, where

- $O = \{0, succ(Counter)\}$,
  consequently $US(Counter) = \{0, succ(0), succ(succ(0)), \ldots\}$
- $UTT = \{inc, set < Counter >\}$,
  consequently $UAT(UTT) = \{inc\} \cup \{set(n) \mid n \in O\} = \{inc, set(0), set(succ(0)), \ldots\}$
- $HUTT = \{set < Counter >\}$. This is the first example of a **hidden** transition.

- $VTT = \{eq < Counter >: Bool\}$,
  consequently $VAT(VTT) = \{eq(n) \mid n \in O\} = \{eq(0), eq(succ(0)), eq(succ(succ(0)), \ldots\}$
- $HVTT = \{\}$.
- $USS = \{From_x \mid x \in O\}$, where
  $\forall x \in O,\ From_x = \{< inc, succ(x) >\} \cup \{< set(n), n > \mid n \in O\}$
- $VSS = \{ValFrom_y \mid y \in O\}$, where
  $ValFrom_0 = \{< eq(0), true, 0 >\} \cup \{< eq(succ(n)), false, 0 > \mid n \in O\}$, and
  $\forall y \in O,\ ValFrom_{succ(y)} = \{< eq(0), false, succ(y) >\} \cup \{< eq(succ(p), q.eq(y), succ(y) > \mid p \in O\}$.

3.2.3 State Label Expressions

The **state labels** which represent the newstate of an object after it services a request and the value returned by a valued attribute can be defined by an expression (called a **state label expression**) which evaluates to a **state label**. The syntax and semantics of such expressions are defined below.

C is the **serving class**, *sl* is the **server** and *att* is the **service** of the **state label expression**. When a **state label expression** is such that the **service class** cannot be uniquely identified from the **server** and the **serving class**, then the class identifier must be included in the expression to remove the risk of ambiguity. For example, in a system which uses the Nat O-LSTS, $M.eq(N)$ must be represented by $M{:}Nat.eq(N)$ if Nat is not the only visible class of the system which has $eq < Nat >$ as a **typed transition** and M as a **state label**.

Additional Hiding Constraint

The members of $HUTT$ and $HVTT$ model internal transitions of a class which are available only to the class in which they are defined. Consequently, we place the additional syntactic constraint on **state label expressions** that if $C_0 \neq C$ then *att*, the **service**, must be an **unhidden transition** of C.

3.3 An Object Oriented Interpretation of the O-LSTS Model

Section 3.3 identifies five relationships which collectively must exist in an analysis model for it to be considered object oriented: classification, interaction, subclassing, composition and configuration. Classification and subclassing are static properties of an object oriented system. Composition and configuration have both static and dynamic aspects, although during analysis they are most usefully given a static interpretation. Interaction is the only purely dynamic property of an object oriented system which is important in analysis.

Definition: state label expression
A **state label expression** in an O-LSTS (C_0) definition is said to be defined on a class C, where $C \in visible(C_0)$. Such an expression is either:

- i) a simple expression, written *sl*, where $sl \in US(C)$.
  C is called the **type** of the simple expresion.
- ii) a transition expression, written *sl.att*, where
  - $sl \in US(C)$
  - *att* is an unparameterised transition, or
    att is a parameterised transition $att(p_1, \ldots, p_n)$ such that $\forall i \in \{1, \ldots, n\}$:
    $p_i \in US(P_i)$ and $att(Q_1, \ldots, Q_n) \in UTT_C$ and $P_i \sqsubseteq Q_i$ in Env_{C_0}

  We define *sl.att* to be an equivalent representation of the **state label** sl', where either $< att, sl' > \in From_{sl} \in USS_C$ or sl'' where $< att, val, sl'' > \in ValFrom_{sl} \in VSS_C$).
  C is called the **type** of a transition expression.
- iii) a result expression, written *sl..att*, where
  - $sl \in US(C)$, and
  - *att* is an unparameterised value transition, or
    att is a parameterised value transition $att(p_1, \ldots, p_n)$ such that $\forall i \in \{1, \ldots, n\}$: $p_i \in US(P_i)$ and $att(Q_1, \ldots, Q_n) \in VTT_C$ and $P_i \sqsubseteq Q_i$ in Env_{C_0}

  We define *sl..att* to be val, where $< att, val, sl' > \in From_{sl} \in USS_C$.
  The **type** of a result expression is the **result type** of the **valued actualised transition attribute** *att*.

3.3.1 O-LSTS Classification

An interpretation of classification in the O-LSTS model is given through the following definitions of class, object and external interface[8].

- **Class definition**
  An O-LSTS defines a class of behaviour. A class is specified by:
  - The set of class members, which is defined by the **typed state label set** in the O-LSTS definition.
  - The external interface (a set of attributes), which corresponds to the **unhidden typed transitions** $(UTT \setminus HUTT) \cup (VTT \setminus HVTT)$ in the O-LSTS definition.
  - The behaviour of the member objects in response to service requests at their external interface, which corresponds to the behaviour defined by the **state-to-state transition sets** of the O-LSTS definition $(USS \cup VSS)$.

[8]The definitions of class, object and external interface are mutually dependent — like the chicken and the egg, it is difficult to say in which order they come!

- **Object definition**

  The term object is used to represent a dynamic instance of a member of a class. In the definition of a class as an O-LSTS, the set of **state labels** are used only to represent the set of possible values (or states) that an object of the class can attain. An object of a class must be associated with a unique identification, which is then used to provide a reference to a particular member of the **state label set** of the class (O-LSTS) of which the object is a member. Consequently, the *state* of an object is defined precisely by the **typed state label** being referenced by the object. The external interface of an object is defined by the external interface of the class to which it belongs. The behaviour provided by the external interface of an object is defined in the **valued** and **unvalued transition sets** of its class ($From_{object}$ and $ValFrom_{object}$).

- **External Interface definition**

  The external interface of a class is defined as a set of attributes. These external attributes are categorised as follows:

  - **Dual:**

    a service request at a dual attribute results in the receiving object updating its internal state and returning some result value to the service requester.
  - **Transformer:**

    a service request at a transformer attribute results in the receiving object updating its internal state without returning any result value to the requester.
  - **Accessor:**

    a service request at an accessor attribute results in the receiving object returning a result value to the requester with no change to its state. (An accessor is a particular type of dual in which the state is 'updated' to the value it was previously.)

  This categorisation is reflected in the O-LSTS semantics as follows:

  - **Dual attributes** are defined by the **valued state-to-state transitions** of an O-LSTS. For example, a **dual attribute** of an object O_j is represented by: al, say, such that $< al$, $result$, $O_k >\in Valfrom_{O_j}$
  - **Transformer attributes** are defined by **unvalued state transitions** of an O-LSTS. For example, a **transformer attribute** of an object O_j is represented by: al, say, such that $< al, O_k >\in From_{O_j}$
  - **Accessor attributes** are particular types of **dual attributes** in which the state of the object is not affected by fulfilling the accessor service. In other words, an accessor attribute of an object O_j is represented by: al, say, such that $< al$, $result$, $O_j >\in Valfrom_{O_j}$

3.3.2 O-LSTS Interaction: The Executable Semantics

Dynamic behaviour of an object oriented system corresponds to the behaviour of the object representing the system. Object behaviour is defined as a sequence of interactions between the object,

O say, and its external environment[9]. The environment of O is made up of a set of **service requesters** (themselves objects) which interact with O, the **service provider**, by requesting services of its external interface.

Creating an object of a class, C say, corresponds to referencing the unique identification for the new object with one of the **state labels**, O_j say, in the set $States(C)$. When the object receives a service request which corresponds to an external attribute, the object behaves as follows:

- When the corresponding external attribute is a transformer, $ua \in UTT$ say, then, by definition, there exists one and only one $< ua, O_k > \in From_{O_j}$. The new state referenced by the object is O_k, and this can be represented, without risk of ambiguity, by the **state label expression** $O_j.ua$. The object does not return any value to the service requester and it proceeds to fulfil the external behaviour as defined for O_k in the O-LSTS C.
- When the corresponding external attribute is a dual, $va : V \in VTT$ say, then, by definition, there exists one and only one $< va : V, result, O_k > \in ValFrom_{O_j}$. The new state of the object is set to be O_k (which can be represented by the **state label expression** $O_j.va$) and the value returned to the service requester is a reference to the **state label** $result$ in the O-LSTS V (which can be represented by the **state label expression** $O_j..va$). The object then behaves like O_k in C.

An implementation of this dynamic model is defined by a mapping from O-LSTS specifications to ACT ONE. The evaluation of certain ACT ONE expressions corresponds to the processing that an object performs in response to a service request (see 3.5 for more details).

3.3.3 O-LSTS Subclassing (and Subtyping)

A subclassing relationship between classes is defined as a relationship between the O-LSTSs corresponding to these classes. Informally, there are four constraints which must be fulfilled for one O-LSTS, A say, to be a subclass of another O-LSTS, B say, (written $A \sqsubseteq B$):

- i) A must provide the external interface of B. If B can service a particular request (i.e. if all the members of B can service the request) then A must also be able to service that request. This is a subtyping relationship.
- ii) All members of A must also be members of B. More precisely, all ways of identifying member objects of class A must also be valid identifications for members of class B.
- iii) The members of A must offer the same behaviour as their corresponding members in B. In other words, it must be impossible to distinguish between corresponding members of A and B by requesting services of these objects which B is capable of fulfilling (we abstract away from the fact that A may offer services which B does not offer).

[9]Note that the environment of an object is not directly related to the environment of the class to which it belongs. The environment of a class represents a set of classes which are used in its definition.

- iv) The environment of classes which B uses must be 'contained' (in some way) in the environment of classes which A uses. The exact definition of containment must take into account the relative visibility of classes and the subclassing relationships between them.

These (informal) necessary and sufficient conditions are formally specified in section 3.3.3.3. The first two conditions are purely syntactic constraints which are based on the static properties of the classes concerned. The third condition is a semantic constraint founded on the dynamic behaviour of the classes. The fourth condition needs examination only when the environments of the two classes are different.

Informally, subclassing is similar to the mathematical notion of the subset relationship. Before proceeding to define subclassing, we review our claim that subclassing is not the same as subtyping. To do this, we formally define a **subtyping** relationship between O-LSTSs and argue that **subtyping** is a necessary, but not sufficient, condition for subclassing (in the intuitive sense). Then we formally define **subclassing** between O-LSTSs and prove that **subclassing** $\Rightarrow$ **subtyping** and **subtyping** $\not\Rightarrow$ **subclassing**

3.3.3.1 Subtyping

Subtyping between O-LSTSs guarantees that any object of a given class, C say, can be replaced by an object which is a member of any subtype (class) of C without introducing the possibility of *syntax errors*[10] into the system in which the replacement is made. Subtyping between O-LSTSs must similarly guarantee only the non-introduction of *syntax errors* when a subtype is used to provide a

[10] In object oriented systems, a *syntax error* results when an object cannot respond to a service requested of it.

replacement for a supertype. Such a relationship is defined below.

Definition: O-LSTS Subtyping ($\leq$)
$< O, UTT, HUTT, VTT, HVTT, USS, VSS >$in $Env_1 \leq$
$< O', UTT', HUTT', VTT', HVTT', USS', VSS' >$in $Env_2 \Leftrightarrow$

- $(UTT \setminus HUTT) = (UTT' \setminus HUTT')$ or
  - (i) given an **unvalued unhidden unparameterised typed transition** of UTT', written *transition-name*, then *transition-name* $\in (UTT \setminus HUTT)$.
  - (ii) given an **unvalued unhidden parameterised typed transition** of UTT', written *t-n*$< U'_1, \ldots, U'_r >$, $\exists$ *t-n*$< U_1, \ldots, U_r > \in (UTT \setminus HUTT)$, such that $U'_i \leq U_i, \forall i \in \{1, \ldots, r\}$.
- $VTT = VTT'$ or
  - (iii) given a **valued unhidden unparameterised typed transition** of VTT', written *transition-name*:V',
    $\exists$ *transition-name*:$V \in (VTT \setminus HVTT)$, such that $V \leq V'$.
  - (iv) given a **valued unhidden parameterised typed transition** of UTT', written *t-n*$< U'_1, \ldots, U'_r >$: V',
    $\exists$ *t-n*$< U_1, \ldots, U_r >$: $V \in (VTT \setminus HVTT)$ such that:
    a) $U'_i \leq U_i, \forall i \in \{1, \ldots, r\}$ and
    b) $V \leq V'$

Conditions (ii) and (iv a) correspond to the 'rule of contravariance' for subtyping — a subtype can accept parameter values which are supertypes of the values which the supertype can accept. Conditions (iii) and (iv b) correspond to the 'rule of covariance' for subtyping — a subtype can respond with values which are subtypes of the values which the supertype responds with. **Subtyping** is reflexive and transitive. Note that the **hidden** transitions are not important in the subtyping relation. When two O-LSTS are subtypes of each other they are said to be **type compatible**:

Definition: Type Compatibility
Two O-LSTSs, A and B, are **type compatible**, written $A \leq\geq B$, $\Leftrightarrow$ A $\leq$ B and B $\leq$ A.

Type compatibility is an equivalence relation.

Example 1: A Simple Subtyping Relationship

The *Store-evens* and *Store-fours* classes in figure 3.5 illustrate a nontrivial **subtyping** relationship. The environments of these O-LSTSs are composed from the O-LSTSs $evens$, $fours$ and $eights$, which are defined such that $evens \leq fours \leq eights$, and $States(evens) = \{2, 4, 6, 8\}$, $States(fours) = \{4, 8\}$ and $States(eights) = \{8\}$. Consequently, by the **subtyping** definition, *Stores-fours* $\leq$ *Stores-evens*. The rule of contravariance is upheld since $evens \leq fours$. The rule of covariance is upheld since $fours \leq eights$.

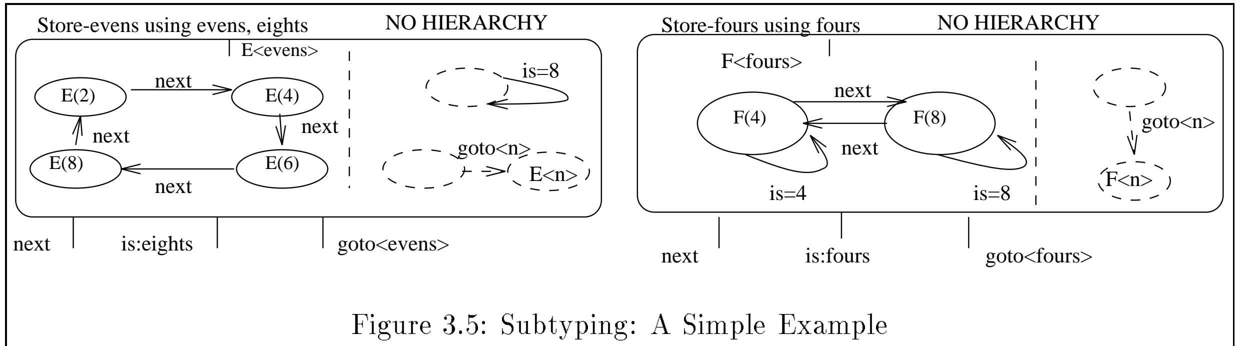


Figure 3.5: Subtyping: A Simple Example

3.3.3.2 Additional Syntactic Constraint Between State Label Sets

From the previous examples, it is clear that **subtyping** does not place any requirements on the representation of **state labels**: it is purely a relationship between O-LSTS interfaces. In our informal definition of subclassing it was stated that all members of a class must also be members of its superclass(es). As O-LSTS members are **state labels** (or more precisely references to **state labels**) of the O-LSTS, it is necessary to place the additional restriction that the **state label** sets of O-LSTSs, which are related by the subclassing relationship, must be related by a subset relationship. This additional syntactic constraint is a necessary but not sufficient condition for **subclassing**. Example 2 shows that when both syntactic conditions hold a subclassing relationship is still not guaranteed.

Example 2: Subtyping is not Subclassing

At the beginning of this section, **subtyping** was said to be 'too weak' to be equated to our informal behavioural notion of subclassing. Subclassing between classes of behaviour requires a formal relationship between the behaviour offered by the members of each of the classes. The need for this additional behavioural requirement is re-iterated by the O-LSTSs defined in figure 3.6.

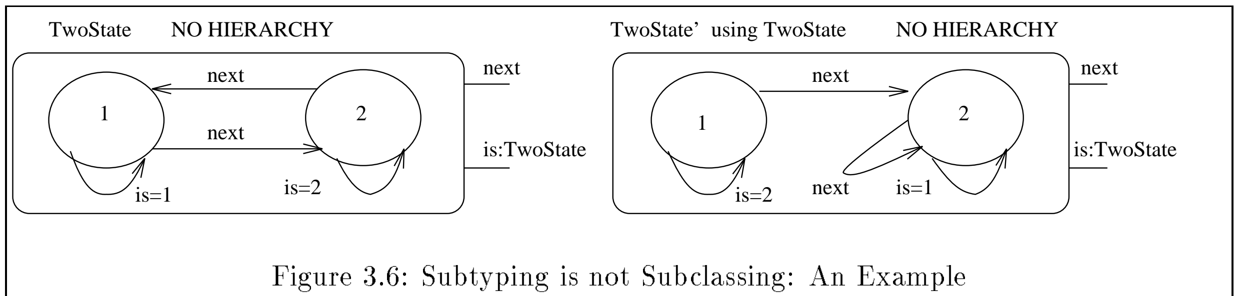


Figure 3.6: Subtyping is not Subclassing: An Example

The two O-LSTSs are **type compatible**, even though the behaviour offered by them is quite different. It is not possible to replace a member of one O-LSTS with a member of the other whilst guaranteeing the behaviour of the system in which the change is made. In this case the **subtyping** relationship guarantees only the non-introduction of *syntax errors* when a member of one class is replaced by the corresponding member of another. Consequently, a semantic relationship (in which replacement somehow guarantees behavioural compatibility) must address the relationship between the **state-to-state transition sets** of O-LSTSs, and not just their external interfaces. This semantic

requirement is precisely the notion we capture in our **subclassing** relationship. We return to the formalisation of this relationship in the knowledge that **subtyping** must be a necessary but not sufficient condition for **subclassing**.

3.3.3.3 Subclassing

Definition: Subclassing ($\sqsubseteq$)
Given A, specified as $< O, UTT, HUTT, VTT, HVTT, USS, VSS >$ and B,
specified as $< O', UTT', HUTT', VTT', HVTT', USS', VSS' >$ then $A \sqsubseteq B \Leftrightarrow$

- i) $US(A) \subseteq US(B)$.
- ii) Every **unvalued unhidden unparameterised typed transition** of B is also an **unvalued unhidden unparameterised typed transition** of A.
- iii) For every **unvalued unhidden parameterised typed transition** of B, written *transition-name*$< U_1, \ldots, U_r >$, there is an **unvalued unhidden parameterised typed transition** of A, written *transition-name*$< V_1, \ldots, V_r >$, such that $U_i \sqsubseteq V_i$ in $Env_A, \forall i \in \{1, \ldots r\}$.
- iv) For every **valued unhidden unparameterised typed transition** of B, written *transition-name*:Val_B, there is a **valued unhidden unparameterised typed transition** of A, written *transition-name*:Val_A, such that $Val_A \sqsubseteq Val_B$ in Env_A.
- v) For every **valued unhidden parameterised typed transition** of B, written *transition-name*$< U_1, \ldots, U_r >$:Val_B, there is an **valued unhidden parameterised typed transition** of A, written *transition-name*$< V_1, \ldots, V_r >$:Val_A, such that $U_i \sqsubseteq V_i$ in $Env_A, \forall i \in \{1, \ldots r\}$ and $Val_A \sqsubseteq Val_B$ in Env_A.
- vi) When at is an **unhidden typed transition**:
  - $\forall < at, b' > \in From_b \in USS_B, < at, b' > \in From_b \in USS_A$, and
  - $\forall < at, val1, b' > \in ValFrom_b \in VSS_B, < at, val1, b' > \in ValFrom_b \in VSS_A$.
- vii) $Env_A =< C'_A, Rel_A >$ and $Env_B =< C'_B, Rel_B >$ are such that:
  - $visible(B) \setminus \{B\} \subseteq visible(A) \setminus \{A\}$
  - $\forall C_i, C_j \in C'_B$ such that $C_i \sqsubseteq C_j$ in Env_B, then $C_i \sqsubseteq C_j$ in Env_A.
  - $\forall C$ such that $B \sqsubseteq C$ in Env_B, then $A \sqsubseteq C$ in Env_A.
  - $\forall C$ such that $B \sqsupseteq C$ in Env_B, then $A \sqsupseteq C$ in Env_A.

3.3.3.4 Subclassing Examples

The list of examples that follow do not exhaustively identify interesting properties of the O-LSTSs semantics with regard to the **subclassing** relationship. However, the following examples do introduce some of the more important concepts. In particular, the examples illustrate the types of behaviour which are related by a **subclassing** relationship, and contrasts them with similar behaviours which are not related in this way.

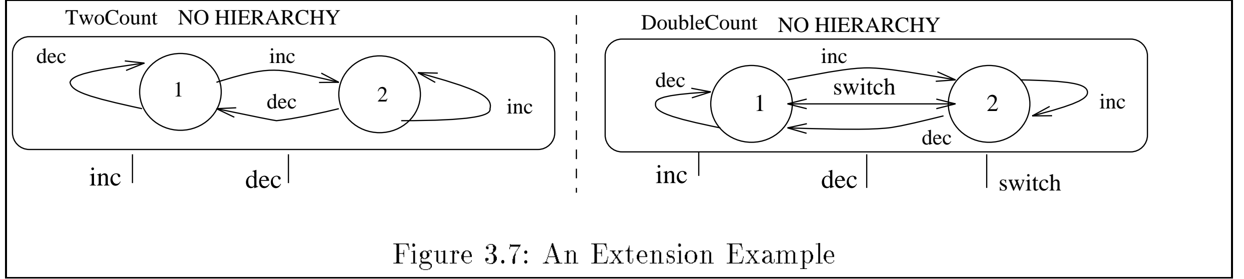


Figure 3.7: An Extension Example

Subclassing Example 1: Extension

Consider the O-LSTSs defined in figure 3.7.

Each of the O-LSTSs are '2-state machines'[11]. For simplicity, neither of the O-LSTSs has any **valued transitions**: they are said to be **non-responsive**. Furthermore, the O-LSTSs are defined in trivial environments ($< \{\}, \{\} >$). In this case, the **subclassing** relationship corresponds to some sort of structural (or topological) relationship between the internal representation of the O-LSTSs.

It is straightforward to prove, by checking the necessary and sufficient conditions of $\sqsubseteq$, that $DoubleCount \sqsubseteq TwoCount$ whilst $TwoCount \not\sqsubseteq DoubleCount$.

> **Proof:** $DoubleCount \sqsubseteq TwoCount$, since conditions (i) to (vii) of the subclassing relationship are upheld:
>
> - i) $\{1,2\} \subseteq \{1,2\}$
> - ii) $\{inc, dec\} \subseteq \{inc, dec, switch\}$
> - iii) $\{\} \subseteq \{\}$
> - iv) $\{\} \subseteq \{\}$
> - v) $\{\} \subseteq \{\}$
> - vi) $\{< inc, 2 >, < dec, 1 >\} \subseteq \{< inc, 2 >, < dec, 1 >, < switch, 2 >\}$ and $\{< inc, 2 >, < dec, 1 >\} \subseteq \{< inc, 2 >, < dec, 1 >, < switch, 1 >\}$ and $VSS_{DoubleCount} = VSS_{TwoCount} = \{\}$
> - vii) The environments of both O-LSTSs are trivially identical.
>
> **Proof:** $TwoCount \not\sqsubseteq DoubleCount$, since
> Condition (ii) of the subclassing relation is not fulfilled, since $switch$ is an unhidden transition of $DoubleCount$ but $switch$ is not an unhidden transition of $TwoCount$.

These O-LSTSs illustrate a particular relationship which we refer to as **extension**.

[11] An O-LSTS is said to be an 'n-state' machine iff the cardinality of its **state label set** equals n.

Definition: extension
A is an **extension** of B, written A **ext** $B \Leftrightarrow$
$A \sqsubseteq B$ and $(UAT(UTT_A) \cup VAT(VTT_A)) \subset (UAT(UTT_B) \cup VAT(VTT_B))$.

In other words, when A **ext** B, A offers all the attributes which B offers together with some additional attributes. When these additional attributes are ignored, every object in A behaves exactly like its corresponding object in B. The inverse of the **extension** relation is **restriction**:

Definition: restriction
A is a **restriction** of B, written A **res** $B \Leftrightarrow B$ **ext** A

Subclassing Example 2: Specialisation

The simple subclassing example in the O-LSTSD in figure 3.7 shows only a semantic relationship between unresponsive O-LSTSs with identical state sets. Both these restrictions are removed in the behaviours defined in figure 3.8.

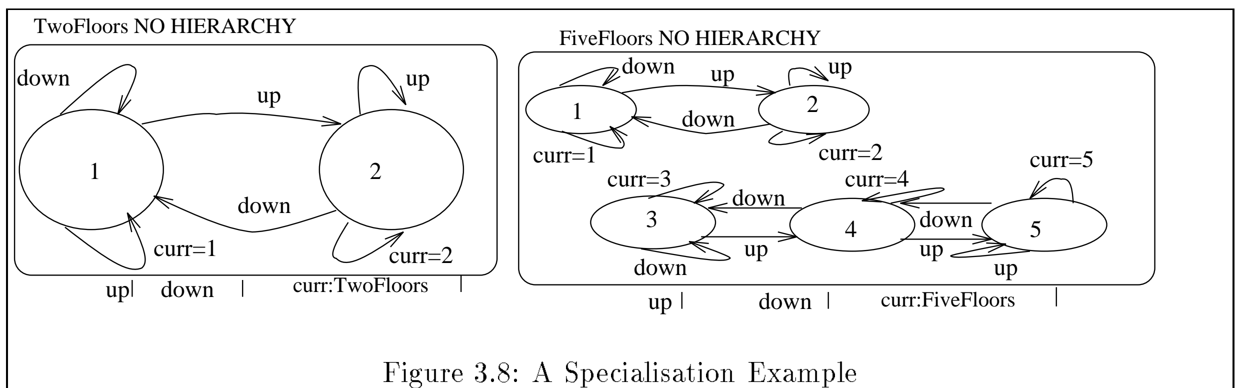


Figure 3.8: A Specialisation Example

Informally, *TwoFloors* specifies a lift system with 2 floors. The system can be requested to go up or down a floor. Also, it can respond with its current floor status when asked. It ignores requests to go up when it is on its top floor (in this case floor 2). Similarly, it ignores requests to go down on floor 1. *FiveFloors* specifies a lift system with 5 floors which cannot move between floors 2 and 3 (perhaps the lift system is broken). *FiveFloors* also ignores requests to go up and down whenever these movements are not possible. It should be clear that, by definition, $TwoFloors \sqsubseteq FiveFloors$. This is an example of **specialisation**.

Definition: specialisation
A is a **specialisation** of B, written A **spec** $B \Leftrightarrow A \sqsubseteq B$ and $States(A) \subset States(B)$.

Informally, if A **spec** B then B is partitioned into distinct sets of behaviour and A provides the behaviour of one or more, but not all, of these partitions. It is useful to be able to define a new class as a partition of an existing class. Such a class, which we refer to as a **partition class**, is specified by identifying a subset of the state set of the original class, provided this set is disjoint from the other state members. We say that an O-LSTS is **nonpartitionable** iff it has no **partition classes**. Like extension, specialisation has an inverse relation. It is called generalisation:

Definition: generalisation
A is a **generalisation** of B, written A **gen** $B \Leftrightarrow B$ **spec** A.

Extension and specialisation (and their inverse relations restriction and generalisation) are the only types of class relations which we consider in figure=Sem-Chp3/Figures. Different combinations of these relationships give rise to an unlimited number of possibilities.

3.3.3.5 Subclassing Guarantees Subtyping

We wish to show that $A \sqsubseteq B \Rightarrow A \leq B$ (The *TwoState* example in figure 4.8 has already shown that $A \leq B \not\Rightarrow A \sqsubseteq B$.) A simple example illustrates the relationship betwen subclassing and subtyping more clearly. Consider $Lift12s$ and $Lift15s$ defined in figure 3.9.

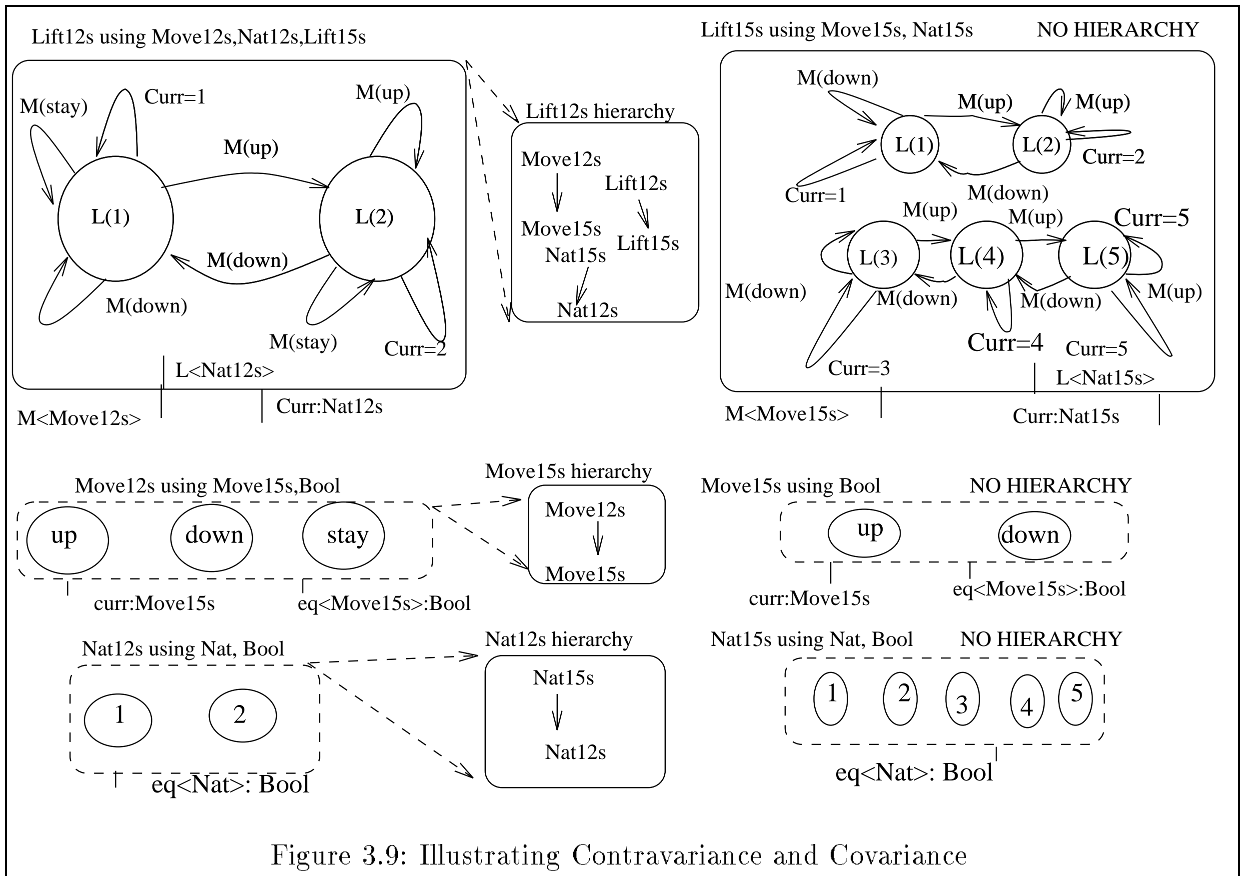


Figure 3.9: Illustrating Contravariance and Covariance

The O-LSTD representation has been extended in this example with two new constructs:

- The O-LSTSs with dotted boundaries are partial specifications in which the state-to-state transition sets are not shown. The partial specifications are used when we wish to represent only the interface of a class of behaviour: consequently, we refer to them as **interface** diagrams.
- The class hierarchies associated with each O-LSTS environment are represented by **class hierarchy diagrams** (the syntax and semantics of such diagrams is defined in section 3.3.3.6).

Consider the syntactic subtyping relationship between the different Lift O-LSTSs. It is clear, by the subtyping definition, that $Nat12s \leq\geq Nat15s$ and $Move12s \leq\geq Move15s$. Subsequently, we can

prove that $Lift12s \leq Lift15s$.

Proof: Lift12s is a subtype of Lift15s

The subtyping relationship is true since the rules of contravariance and covariance hold.

- contravariance:
  $M < Move15s > \in UTT_{Lift15s}$, $M < Move12s > \in UTT_{Lift12s}$ and $Move12s \geq Move15s$
- covariance:
  $curr : Nat15s \in VTT_{Lift15s}$, $curr : Nat12 \in VTT_{Lift12s}$ and $Nat12s \leq Nat15s$

Consider now the semantic subclassing relationship. It is obvious that if we assume that $Move15s \sqsubseteq Move12s$ and $Nat12s \sqsubseteq Nat15s$ then we can prove that $Lift12s \sqsubseteq Lift15s$, and as such the O-LSTSs are well-defined.

Proof: Lift12s is a subclass of Lift15s (given the above assuptions hold)

All the necessary and sufficient conditions for subclassing hold:

- The **state label set** of $Lift12s$ is a subset of the **state label set** of $Lift15s$, since $\{L(1), L(2)\} \sqsubseteq \{L(1), L(2), L(3), L(4), L(5)\}$
- The **unvalued unhidden actualised transition** set of $Lift15s$ is a subset of the **unvalued unhidden actualised transition** set of $Lift12s$, since $\{M(up), M(down), M(stay)\} \supseteq \{M(up), M(down)\}$and the **valued unhidden actualised transition** set of $Lift15s$ is a subset of the **valued unhidden actualised transition** set of $Lift12s$, since $\{curr\} = \{curr\}$
- The input parameters of $Lift12s$ are superclasses of the corresponding input parameters of $Lift15s$ in $Env_{Lift12s}$. This is true by the original assumption.
- The output parameters of $Lift12s$ are subclasses of the corresponding output parameters of $Lift15s$ in $Env_{Lift12s}$. This is true by our original assumption.
- The **state-to-state transitions** of $Lift15s$ are members of the **state-to-state transition** sets of $Lift12s$:
  $From_{L(1)_{Lift15s}} \subseteq From_{L(1)_{Lift12s}}$ and $ValFrom_{L(1)_{Lift15s}} \subseteq ValFrom_{L(1)_{Lift12s}}$
- The environment of $Lift12s$ is contained within the environment of $Lift15s$ since:
  - $visible(Lift15s) \backslash \{Lift15s\} = \{Bool, Nat, Nat15s, Move15s\} \subseteq visible(Lift12s) \backslash \{Lift12s\} = \{Move15s, Bool, Nat15s, Nat, Move12s, Nat12s, Lift15s\}$.
  - $Rel_{Lift15s} = \{\}$ and so there are no conditions to be met concerning the containment of subclassing relationships.

Proof: subclassing implies subtyping

Quite simply, removing conditions (i), (vi) and (vii) of the subclassing definition results in precisely the covariance and contravariance requirements for subtyping. Consequently, it is clear that subclassing is stronger than subtyping. More formally, $A \sqsubseteq B \Rightarrow A \leq B$.

3.3.3.6 Class Hierarchy Diagrams

The class hierarchy diagrams given in the previous example are an explicit statement of subclassing relationships which exist in the environment of an O-LSTS specification. The diagram is a graph of nodes and directed links between nodes. The nodes in the graph correspond isomorphically to the set of visible classes of the particular class, C say. We say that a path exists between nodes A and B iff there is a directed link from A to B, or $\exists$ a node C such that there is a directed link from A to C and there is a path from C to B. For every pair of visible classes, A and B say, related by the subclassing relationship $A \sqsubset B$ in Env_C, there is a path from the superclass node to the subclass node[12]. It is important to note that when one class is used by another, its class hierarchy is contained within the using class.

4.3.3.6 Fulfils — subclassing after syntactic relabelling

Strict conditions on the syntactic labelling of states and transitions are placed between classes related by the subclassing relationship and this can be a hindrance to re-use. Consider the simple examples of 2-state machines in figure 3.10.

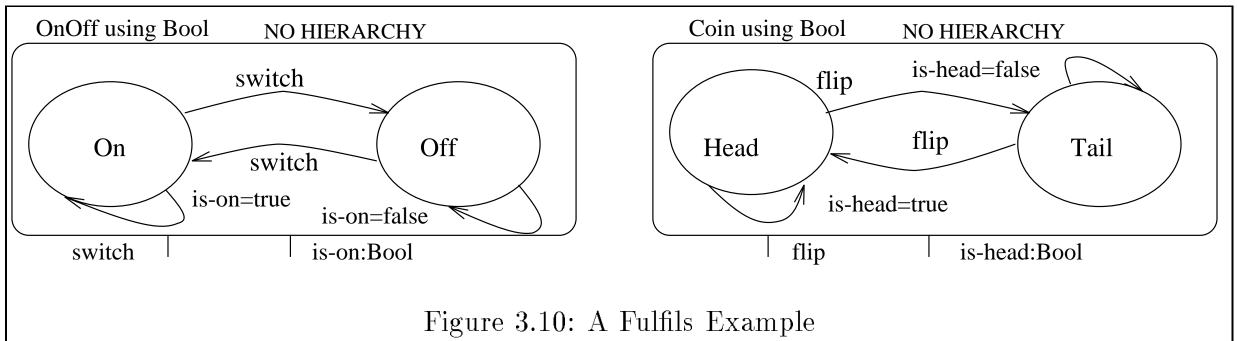


Figure 3.10: A Fulfils Example

These O-LSTSs are not related by the subclassing relationship. However, intuitively, they offer 'the same' behaviour. A simple syntactic relabelling of class names, state labels and transition labels 'transforms' either class into the other. When such a syntactic relabelling, using the transformation T on the class C say, results in a O-LSTS then the new class produced is labelled $T(C)$.

Definition: fulfils
A **fulfils** $B \Leftrightarrow \exists$ some syntactic relabelling transformation, T say, such that $A \sqsubseteq T(B)$.

This definition introduces one particular means of identifying one class of behaviour as a suitable candidate for providing the behaviour as specified by another class of behaviour. The examples in

[12] The symmetric subclassing relationships are not shown in the diagram.

figures 3.11 and 3.12 illustrate two particular types of fulfilment, namely transition reduction (**tr**) and state reduction (**sr**).

Fulfils Example 1: transition reduction

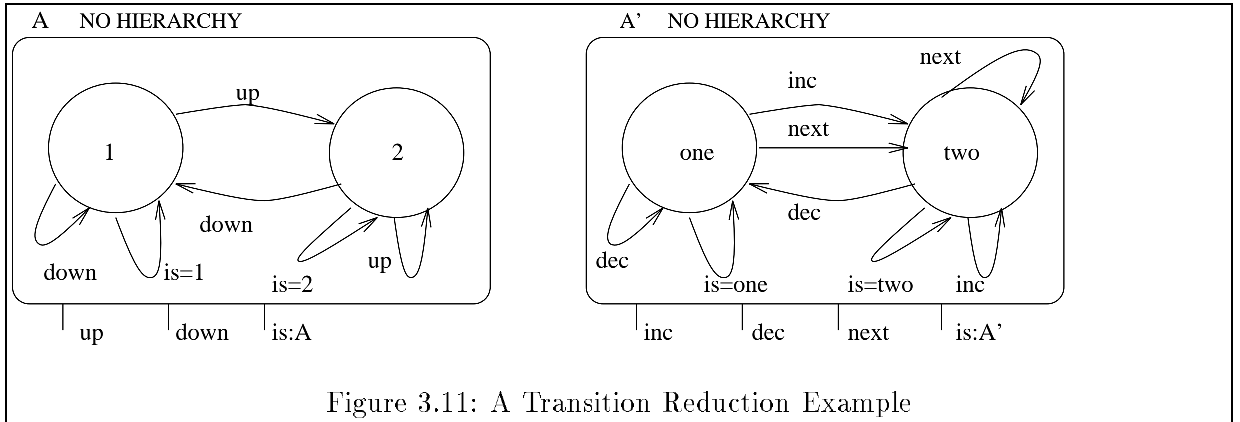


Figure 3.11: A Transition Reduction Example

Consider the O-LSTS in figure 3.11. A **fulfils** A' since $A \sqsubseteq T(A')$, where $T(A') = A$, $T(one) = 1$, $T(two) = 2$, $T(next) = up$, $T(inc) = up$, and $T(dec) = down$. The central idea is that the *next* and *inc* transitions in A' are equivalent (i.e. have the same effect). Consequently, the *up* transition in A can be used to fulfil both *next* and *inc* functionality.

This example illustrates how one class of behaviour simplifies the specification of the behaviour of another class. The O-LSTS A simplifies the specification of A' by reducing two equivalent transitions into one. This is formally defined as **transition reduction**.

> **Definition: transition reduction (tr)**
> A **tr** $B \Leftrightarrow A$ **fulfils** B and the cardinality of the union of the **typed transition sets** of A is less than the cardinality of the union of the **typed transition sets** of B.

Fulfils Example 2: state reduction

Consider the O-LSTSs defined in figure 3.12.

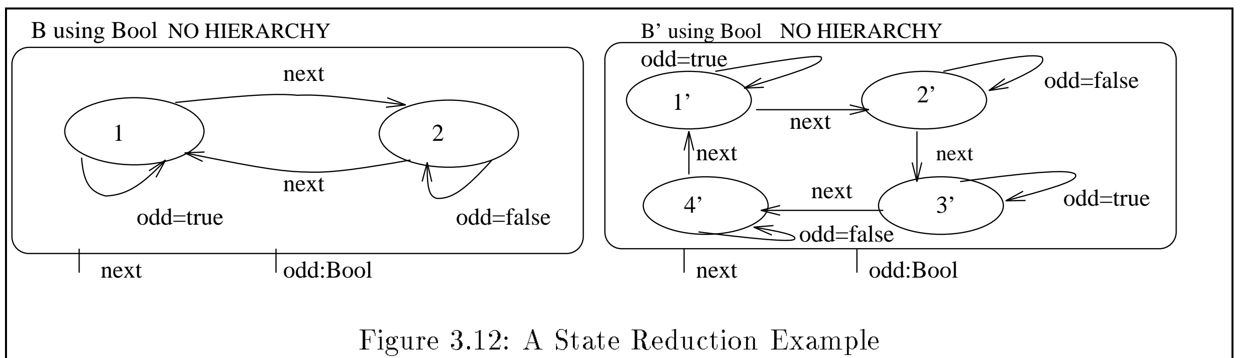


Figure 3.12: A State Reduction Example

B **fulfils** B' since $B \sqsubseteq T(B')$, where $T(B') = B$, $T(1') = 1$, $T(2') = 2$, $T(3') = 1$ and $T(4') = 2$. $T(B')$ is an O-LSTS which simplifies the specification of the behaviour of B' by reducing the number

of states in the system. In B' it is impossible to distinguish states $1'$ and $3'$ (and $2'$ and $4'$) through the external interface offered by the attribute set. The simplification which almalgamates equivalent states is called a state reduction.

Definition: state reduction (sr)
A **sr** $B \Leftrightarrow A$ **fulfils** B and the cardinality of the **state label** set of A is less than the cardinality of the **state label** set of B.

Fulfils: A Preview of Design Issues

The **fulfils** relationship is important when we consider re-use. When the behavioural requirements of a system (or system component) can be **fulfilled** by another already encoded component then it is sensible to re-use that implementation (after the appropriate syntactic relabelling). Note that the **fulfils** definition guarantees that a class always **fulfils** the behaviour of all of its superclasses. Consequently, subclassing provides a very particular kind of re-use facility. Confusion arises in object oriented terminology because subclassing is often thought of as a re-use mechanism rather than a relationship between classes which facilitates re-use.

Fulfils Example 3: Restructuring for design

The notion of fulfilment is important in design because it allows for the restructuring of class specifications whilst guaranteeing that the restructuring does not alter the requirements being defined. A simple example in figure 3.13 illustrates this.

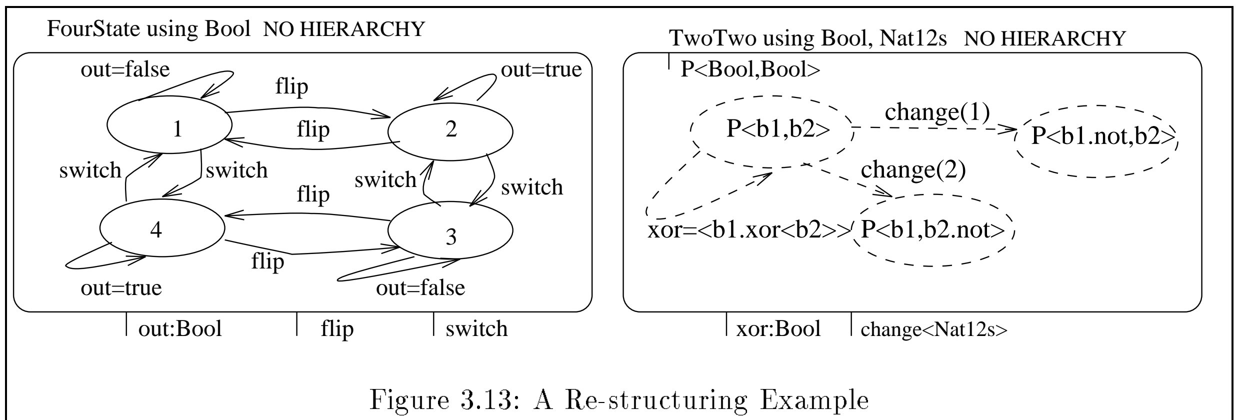


Figure 3.13: A Re-structuring Example

$TwoTwo$ **fulfils** $FourState$ since $TwoTwo \sqsubseteq T(FourState)$, where $T(out) = xor$, $T(flip) = change(2), T(switch) = change(1), T(1) = P(true, true), T(2) = P(true, false), T(3) = P(false, true)$, and $T(4) = P(false, false)$. $TwoTwo$ is a more structured specification than FourState. Adding meaningful structure to a specification (without changing the behaviour offered at the external interface) is an important aspect of design. Structure is fundamental to understanding — it encourages the re-use of pre-defined behaviours and the generation of re-usable behaviour. For example, the xor behaviour (provided by class $Bool$) is well understood and its re-use in $TwoTwo$ improves the specification.

In chapter 5, we consider design as a sequence of correctness preserving transformations. The *correctness* property between designs is related to the **fulfils** relationship in the analysis.

3.3.3.8 Inclusion

The **inclusion** example in figure 3.14 illustrates a form of re-use which is neither compositional nor subclassing. Nevertheless, this form of re-use is very common and effective. We refer to it as **inclusion**.

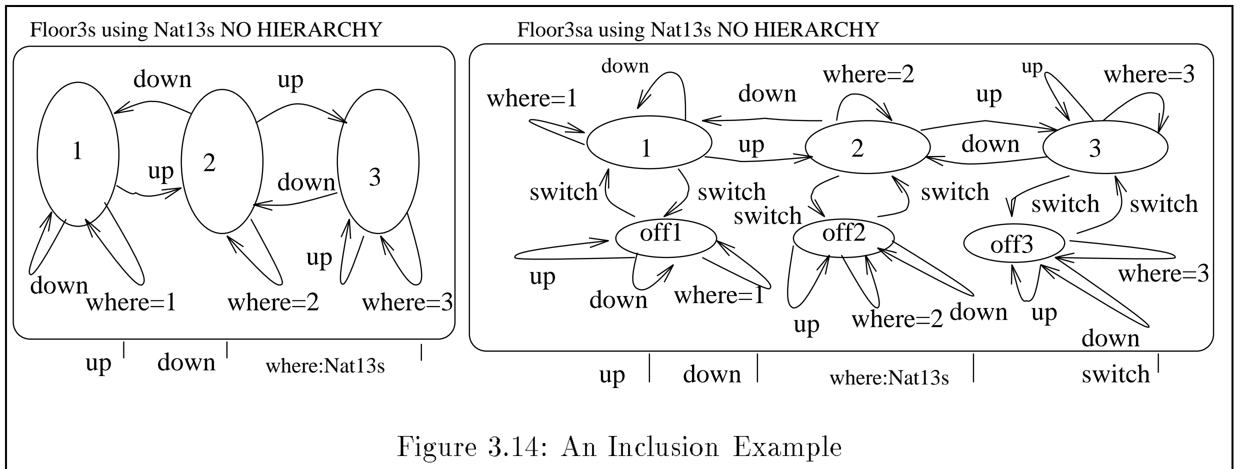


Figure 3.14: An Inclusion Example

$Floor3sa \not\sqsubseteq Floor3s$ since the 'off states' in $Floor3sa$ do not have any corresponding members in $Floor3s$. Also, $Floor3s \not\sqsubseteq Floor3sa$ since $Floor3s$ does not offer the *switch* attribute. Neither class can **fulfil** the other behaviour and neither class is **dependent** on the other. By ignoring the *switch* operator one can say that $Floor3s$ **specialises** $Floor3sa$. Clearly, it is advantageous to be able to define either one of these classes in terms of the other through some explicit re-use mechanism, which is based on the **inclusion** definition, below.

Definition: inclusion
B **includes** A , written B **inc** $A \Leftrightarrow$

- $\forall x \in US(B), From_x \in USS(B)$ is a subset of $From_x \in USS(A)$
- $\forall x \in US(B), ValFrom_x \in USS(B)$ is a subset of $ValFrom_x \in USS(A)$

This simple definition provides the foundation for a powerful purely syntactic re-use mechanism. To define A in terms of B it is necessary only to specify a subset of the **typed transition set** of B together with a subset of the **state label** set of B. To define B in terms of A it is necessary only to define new transitions and new states together with their associated behaviour.

3.3.4 O-LSTS Composition

An object in a class is represented by a **typed state label**. By definition, a **typed state label** is either unparameterised or parameterised. An unparameterised **typed state label** is represented by

a unique *state-constructor*. A parameterised **typed state label** is represented by a *state-constructor* followed by a list of parameter values.

An object which references an unparameterised state label is said to be **unstructured**. Such objects are not said to be composed from any component objects. An object which references a parameterised state label is said to be **structured**. A **structured** object is said to be composed from a set of component objects, which are precisely the objects corresponding to the parameter values. The optional boolean expression corresponds to an invariant property which collectively the components of the object must fulfil.

Reconsider the two O-LSTSs, $FourState$ and $TwoTwo$, specified in figure 3.13. The **state labels** of $FourState$ are unparameterised and consequently the objects which reference those labels are **unstructured**. Contrastingly, the **state labels** of $TwoTwo$ are parameterised — $\forall b1, b2 \in Bool, P(b1 : Bool, b2 : Bool) \in States(TwoTwo)$.

Composition is a relationship between objects. The notion can be extended to classes as follows. When all the **state labels** in an O-LSTS are represented by the same parameterised *state-constructor* then, since by definition the **parameter classes** are uniquely defined, the class is said to be **composed** from the **parameter classes** of the *state-constructor*. The *state-constructor* is said to define the **fixed structure** of the O-LSTS (class).

Definition: Composition

- **Object composition:** An object, O say, with corresponding **state label** *state-constructor*$(p_1, \ldots p_n)$ is said to be **composed** from $p_1, \ldots, p_n$, which are called the **components** of O.
- **State composition:** A class, C say, is said to be **composed** from classes $C_1, \ldots, C_n \Leftrightarrow \exists$ a state constructor, sc say, such that $\forall o \in States(C), \exists$ **state labels** $c_1, \ldots, c_n$ such that o is represented as $sc(c_1 : C_1, \ldots c_n : C_n)$.

When an object recieves a service request at one of its external attributes, it responds by returning a value and/or updating its internal state. A **structured** object achieves this functionality by requesting services of its components. These services may update the state of the components and/or return a result. With this in mind, it is now possible to formally define an object oriented interpretation of the internal processing that occurs when an object services a request. This interpretation is then used to formalise the representation of parameterised expressions in the O-LSTSD syntax. Consider the specification of two interacting stacks given in figure 3.15.

Two new pieces of syntax have been introduced:

- **Unspecified operations**

  The specifiers may not wish to define the result returned by a `pop` on an `empty Stack`. However, the implementors must provide the `pop` service for all `Stack` class members. The result of the `pop` operation must be a `Nat`. An unspecified member of the `Nat` class is represented by `~Nat`. This value is used to define the result returned by an `empty Stack` in response to a `pop` request. Unless otherwise specified, a service requested of an unspecified member always results in the

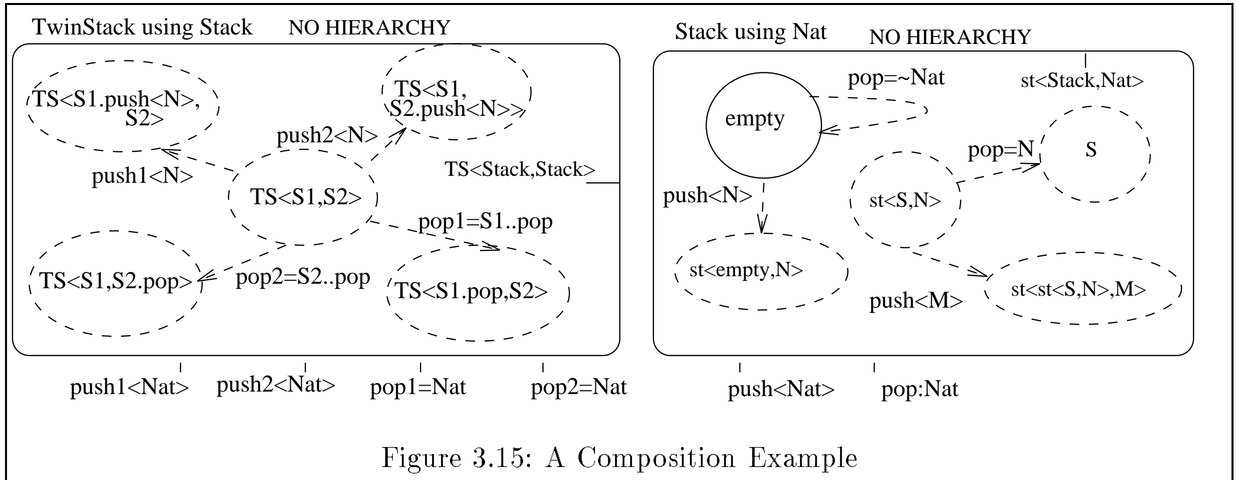


Figure 3.15: A Composition Example

unspecified member of the appropriate class. This default behaviour is implicit in every O-LSTS specification.

- **Using Component Services**

  A component object offers two 'responses' when it is sent a message request: it updates its internal state and/or replies with a result. The `pop` operation on a `Stack` results in the `Stack` replying with the last integer which had been pushed on, and updating its internal state by removing the top value. It is necessary to consider how the `pop` operation of a `Stack` component is used by the `TwinStack`. Service `pop1` on a `TwinStack` returns the value on the top of the first `Stack` component and updates the state of the component accordingly. Service `pop2` returns the value on the top of the second stack component and updates its state accordingly. Operations `push1` and `push2` are similarly defined.

  Two different pieces of syntax are applied:

  - The value returned in response to a valued service request, SR say, at an object O is represented by $O..SR$.
  - The new state of an object O after receiving a service request SR is represented by $O.SR$.

In section 4.2 we return to the notion of composition when using OO ACT ONE. More complex examples are considered, and the notion of an object 'restructuring' itself is examined. In particular, classes of objects with dynamic structure are investigated.

3.3.5 O-LSTS Configuration

Configuration is related to the notion of composition. Composition defines a hierarchical relationship between a client (containing object) and a server (component object). Configuration is a relationship between peer objects which are components of the same containing object. Components of the same containing object may, or may not, be configured.

Consider again the *TwinStack* O-LSTS (in figure 3.15). The class is composed from two components, both of which are `Stacks`. However, a `TwinStack` object never needs to use both components

to service one single request: `pop1` and `push1` use only the first `Stack` whilst `pop2` and `push2` use only the second component. In this case, the two `Stack` components are said to be **unconfigured**.

Consider the O-LSTS in figure 3.16.

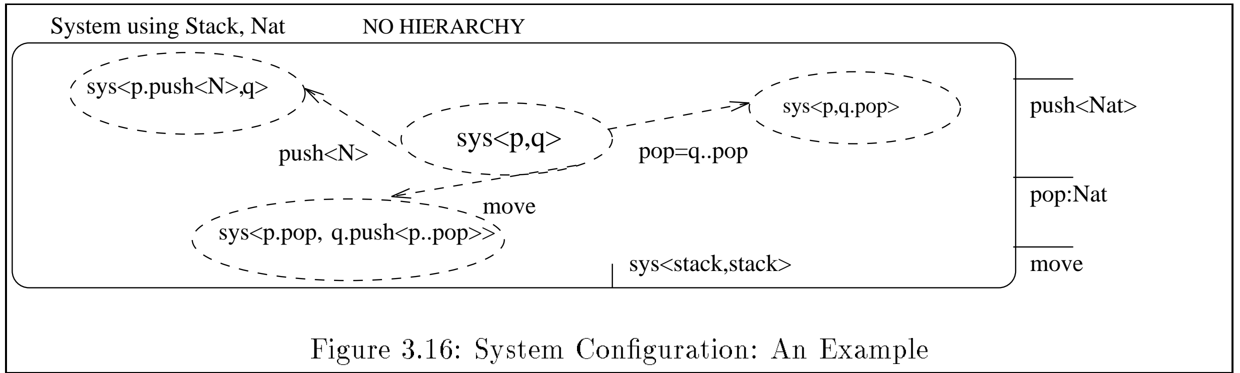


Figure 3.16: System Configuration: An Example

`System` is composed from two `Stack` components. The `push` transition puts the input parameter `Nat` value onto the first `Stack`. The `pop` transition removes the top `Nat` element from the second `Stack` and returns its value. The `move` transition transfers the top element of the first `Stack` onto the second `Stack`. The `move` transition is defined in terms of both components and so we say that the two `Stacks` are **configured** (by `move`). Note that configuration between objects does not necessarily imply interaction between the two components.

Configuration is formally defined below. The definition is based on the idea of one **state label expression** being used in the definition of another. The first expression is said to **depend on** the second. In particular, one object is said to **depend on** one of its components if the component is needed to fulfil an external service request.

Definition: Configuration
Objects A and B are **configured** (in object C) $\Leftrightarrow$

- A and B are **components** of C
- Either:
  - i) $\exists < uat, newstate > \in From_C$ such that $newstate$ **depends on** A and $newstate$ **depends on** B, or:
  - ii) $\exists < vat, val, newstate > \in ValFrom_C$ such that ($newstate$ **depends on** A or val **depends on** A) and ($newstate$ **depends on** B or val **depends on** B)

Definition: Dependence
A **state label expression**, SLE say, **depends on** another **state label expression**, sl say $\Leftrightarrow$ An expression of the form $sl.att$ or $sl..att$ appears in the representation of SLE.

3.3.6 Structure Diagrams

The internal structure of an object, i.e. an object's composition and configuration properties, is usefully represented in diagramatic form. For example, the O-LSTSs *System* and *TwinStack* are represented in the class structure diagrams in figure 3.17. The dotted circles represent classes of object. The class name is given above the circle. Attribute dependencies are shown as links joining the container class with the component classes[13] which it **depends on** to fulfil that particular attribute. Class structure diagrams are appropriate only when the class has a **fixed structure**.

Object structure diagrams differ from class structure diagrams in that actual objects are represented by solid circles and components are given concrete values. Figure 3.17 shows an object structure diagram for an element of the O-LSTS system which has two empty stacks as its internal components.

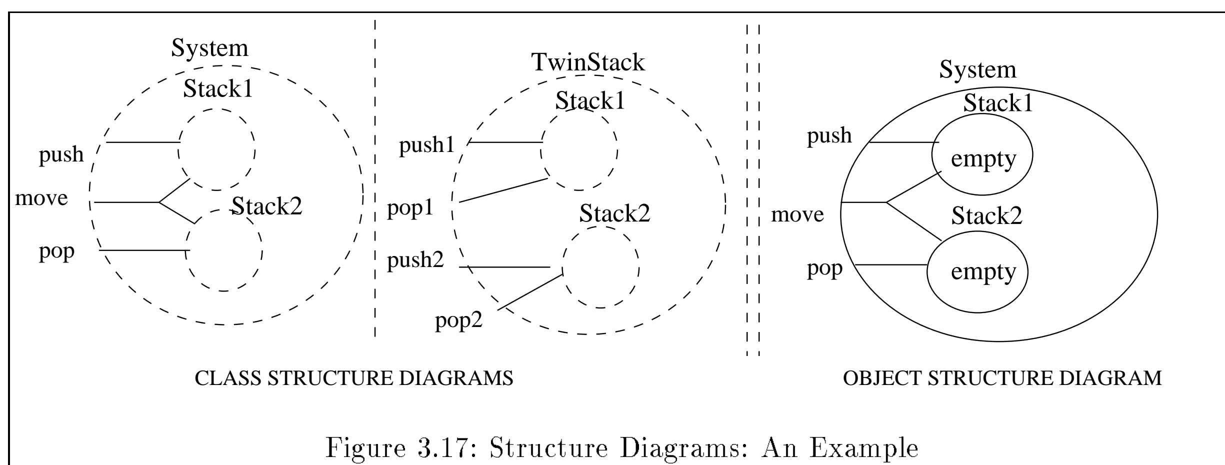


Figure 3.17: Structure Diagrams: An Example

3.4 OO ACT ONE: A Formal Object Oriented Analysis Language

3.4.1 Motivation

In this section we develop a concrete syntax for the specification of O-LSTS behaviour. The previous two sections illustrate how O-LSTSs can be defined using mathematical notation. This syntax is fine for the specification and illustration of simple behaviours. However, for the specification of more complex requirements, a *better* object oriented analysis language is required:

- An object oriented language should have an object oriented 'flavour'. It must be possible to reason about object oriented specifications in an object oriented conceptual framework. OO ACT ONE facilitates a more direct correspondence between the requirements being specified and the object oriented paradigm. It provides a number of high level mechanisms which syntactically sugar the O-LSTS model. The advantage of using OO ACT ONE over an informal object oriented language is that the underlying model is formally specified. Furthermore, OO ACT

[13]The components are identified by their class name and their index in the fixed structure. This index is used to distinguish components of the same class.

ONE specifications say *what* the system being specified should do rather than *how* it should do it.

- An O-LSTS specification must be statically analysed to check that all the necessary and sufficient conditions are met. In particular, the environment of an O-LSTS specifies a class hierarchy (set of subclassing relationships between O-LSTSs) which must be validated. Furthermore, there are 'typing constraints' on the visibility of state constructor (and state transition) parameters and result types which must be checked. A strict syntax for the specification of O-LSTS behaviour can help to make the 'type checking' easier to perform[14].
- The final goal of this chapter is to map our formal object oriented model onto ACT ONE. By defining a formal language which is similar in structure to ACT ONE, the translation to ACT ONE is simplified.

OO ACT ONE provides a practical means of specifying object oriented requirements in a formal framework.

3.4.2 The OO ACT ONE Syntax: Some Examples

Chapter 2 examines many of the practical issues in the design of a figure=Sem-Chp3/Figures language. It identifies the need for:

- A means of distinguishing between accessor, transformer and dual attributes.
- Comprehensive re-use facilities.
- A means of defining invariant properties which all class members must fulfil.
- A mechanism for hiding internal/local definitions or behaviour.
- A means of representing the structure of an object.
- A way of defining exceptions or unspecified behaviour.
- Explicit sub(super)classing mechanisms.

OO ACT ONE, a formal language which fufils all these needs, is best illustrated by the following list of examples. The first eight examples consider object based specifications, in which no subclassing relationships are explicitly defined. The final four examples consider in turn the four explicit (sub/super)class mechanisms, namely specialisation, generalisation, extension and restriction. We argue, in section 4.1, that these four mechanisms are sufficiently powerful for the general construction of class hierarchies during formal object oriented analysis. For simplicity, we are not yet concerned with the static analysis of the specifications which guarantees their correctness. All the example OO ACT ONE specifications that follow are well defined in the sense that they correspond to valid O-LSTSs.

[14] Perhaps 'type checking' is more accurately termed 'class checking'. However, since the notion of 'type checking' is pervasive in all areas of computing (even in object oriented development), we persist with this 'weaker terminology'.

Example 1: Classes Nat, Stack and System Revisited

The O-LSTS specifications of the O-LSTS `System` (see figure 3.16), the O-LSTSs `Stack` (see figure 3.15) and the unspecified `Nat` component provide good examples with which to illustrate the OO ACT ONE syntax. The `Nat` behaviour was not previously defined in O-LSTS form because its behaviour was not relevant to the example. For completeness, a simple `Nat` O-LSTS is defined in figure 3.18.

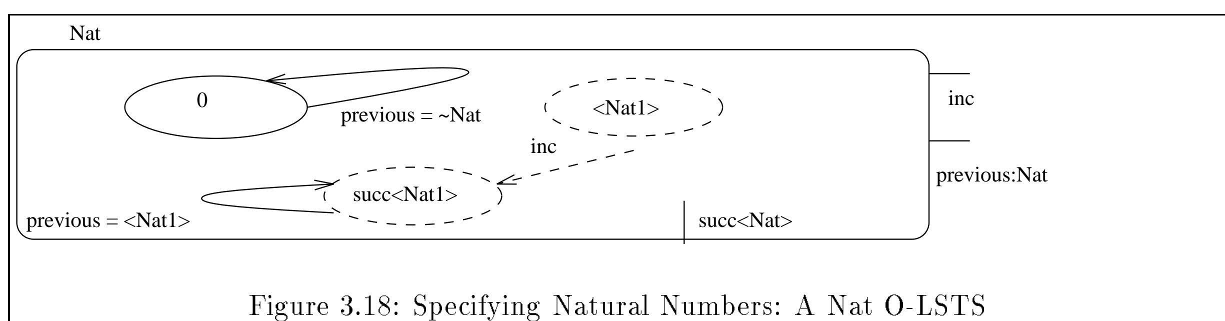


Figure 3.18: Specifying Natural Numbers: A Nat O-LSTS

The OO ACT ONE specification corresponding to the *Nat* O-LSTS is given below.

```
CLASS Nat OPNS
LITERALS: 0
STRUCTURES: succ<Nat> TRANSFORMERS: inc
ACCESSORS: previous -> Nat
EQNS
Nat1.inc = succ(Nat1); 0..previous = ~Nat; succ(Nat1)..previous = Nat1
ENDCLASS (*Nat*)
```

The following should be noted:

- Class `Nat` is not dependent on any other classes and consequently there are no class relationships defined between `Nat` and any other classes.
- The **state label expression** syntax and semantics is incorporated in OO ACT ONE.
- Variable parameters of a particular class (in equation definitions) are represented by the class name followed by an integer. In this way, unlike ACT ONE, the specifier does not need to explicitly 'type' the parameters of an expression. This elegantly concise syntax is unambiguous given the additional syntactic constraint that class names ending in an integer are not permitted (see 3.2.1.1).
- The unspecified member of a class is implicit in an OO ACT ONE specification.
- The OO ACT ONE syntax forces all operations and equations on a class to be defined in that class body. No new operation for a class can be defined in a different class.

The mapping between the `Nat` class and the *Nat* O-LSTS is very simple:

- $Env_{Nat} = < \{\}, \{\} >$, since `Nat` is defined independently of any other classes.
- The `LITERAL 0` corresponds to the **unparameterised typed state label** 0.

- The `STRUCTURE` operation `succ<Nat>` corresponds to the **parameterised typed state label** $succ(Nat)$.
- The `TRANSFORMER` operation `inc` corresponds to the **unvalued unparameterised typed transition** inc.
- The `ACCESSOR` operation `previous -> Nat` corresponds to the **valued unparameterised typed transition** $previous : Nat$.
- The equation `Nat1.inc = succ(Nat1)` corresponds to a parameterised set of **unvalued state-to-state transitions**: $< inc, succ(Nat1) > \in From_{Nat1}, \forall Nat1 \in US(Nat)$.
- The equation `0..previous = ~Nat` corresponds to the **valued state-to-state transition**: $< previous, \sim Nat, 0 > \in ValFrom_0$.
  (As `previous` is an accessor attribute, the state of the object servicing a `previous` request does not change.)
- The equation `succ(Nat1)..previous = Nat1` corresponds to a parameterised set of **valued state-to-state transitions**:
  $< previous, Nat1, Nat1 > \in ValFrom_{succ(Nat1)}, \forall Nat1 \in US(Nat)$.
- There are no **hidden** operations and consequently $HUTT$ and $HVTT$ in the corresponding O-LSTSs are empty sets.

The OO ACT ONE specification of the *Stack* O-LSTS uses the `Nat` behaviour. It is defined below.

```
CLASS Stack USING Nat OPNS
LITERALS: empty STRUCTURES: st<Stack, Nat>
TRANSFORMERS: push<Nat>
DUALS: pop -> Nat
EQNS
empty.push(Nat1) = st(empty,Nat1);
st(Stack1,Nat1).push(Nat2) = st(st(Stack1,Nat1),Nat2);
empty.pop = empty AND ~Nat;
st(Stack1,Nat1).pop = Stack1 AND Nat1
ENDCLASS (*Stack*)
```

This OO ACT ONE specification illustrates three new aspects of the OO ACT ONE syntax:

- `USING`: the `Stack` class is defined to 'use' the `Nat` class.
- `DUALS`: the `pop` operation is defined by an equation of the form: **state label expression1**.pop = **state label expression2** `AND` **state label expression3**. This defines a parameterised set of **valued state to state transitions**:
  $<$ *pop*, state label expression2, state label expression3$> \in ValFrom_{statelabelexpression1}$.
- `Paramaterised Attributes`: the operation `push<Nat>` corresponds to a **parameterised unvalued typed transition** $push(Nat)$. It has two associated equations:
  - `empty.push(Nat1) = st(empty, Nat1)`, which corresponds to a parameterised set of **unvalued state-to-state transitions**:
    $< push(Nat1), st(empty, Nat1) > \in From_{empty}, \forall Nat1 \in US(Nat)$.

- `st(Stack1, Nat1).push(Nat2) = st(st(Stack1, Nat1), Nat2)`, which corresponds to a
  parameterised set of **valued state-to-state transitions**:
  $< push(Nat2), st(st(Stack1, Nat1), Nat2) > \in From_{st(Stack1,Nat1)}$,
  $\forall Nat1, Nat2 \in US(Nat), Stack1 \in US(Stack)$.

The `Stack` specification is used by the `System` OO ACT ONE specification, defined below.

```
CLASS System USING Stack OPNS
STRUCTURES: sys<Stack, Stack>
TRANSFORMERS: push<Nat>, move
DUALS: pop -> Nat
EQNS
sys(Stack1,Stack2).push(Nat1) = sys(Stack1.push(Nat1), Stack2);
sys(Stack1,Stack2).move = sys(Stack1.pop, Stack2.push(Stack1..pop));
sys(Stack1,Stack2).pop = sys(Stack1, Stack2.pop) AND Stack2..pop
ENDCLASS (*System*)
```

Example 2: Grouping classes into modules

In ACT ONE, one type (a group of related sorts) can be defined using the group of sorts defined in another type using the `IS` construct. In other words, the `IS` construct in ACT ONE defines a relationship between types (not sorts). In OO ACT ONE we are more interested in the dependencies between classes of behaviour rather than the modules in which they are defined. However, it is still desirable to be able to re-use sets of related classes. As a compromise, we define a re-use facility between a class and a set of other classes which have been grouped together in a module definition. Module definitions are given before the set of class definitions which make up an OO ACT ONE specification. Module definitions are removed by a simple pre-processing of an OO ACT ONE specification. This syntactically substitutes the names of modules which are used by classes with the list of classes which are grouped within these modules. In other words, the modules in OO ACT ONE are simple syntactic sugaring — they do not extend the semantics of the O-LSTS model in any way. Consequently, they do not need to be considered in the mapping between OO ACT ONE and the O-LSTS model.

The classes grouped together in `module Degrees`, defined below, can be re-used by another class simply by listing the module name in the class header. For example, we can define a class 'Example' to use the classes Bool, Nat, Stack and all the classes in the module Degrees as follows: `Class Example USES Bool, Nat, MODULE Degrees, Stack`. The OO ACT ONE preprocessor removes the module definition and changes the `Example` class header to: `Class Example USES Bool, Nat, Joints, Singles, Stack`.

```
MODULE Degrees GROUPS Joints, Singles ENDMODULE (* Degrees *)
CLASS Joints USING Subject, Type OPNS
STRUCTURES: JointDegree<Subject, Type, Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
JointDegree(Subject1,Type1,Subject2)..studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* Joints *)
CLASS Singles USING Subject OPNS
STRUCTURES: SingleDegree<Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
SingleDegree(Subject1)..studies(Subject2) = Subject1..eq(Subject2)
ENDCLASS (* Singles *)
```

Example 3: Invariant Properties

To illustrate the notion of an invariant property, consider a class similar to `Joints` except that it has no `Type` component. This class is defined as `Joints2` below.

```
CLASS Joints2 USING Subject OPNS
STRUCTURES: JointDegree<Subject, Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
JointDegree(Subject1, Subject2)..studies(Subject3) =
(Subject1..eq(Subject3)).or(Subject2..eq(Subject3))
ENDCLASS (* Joints2 *)
```

`JointDegree(Maths, Maths)` is a member of the `Joints2` class provided that `Maths` is a literal member of the `Subject` class. This may not be desirable behaviour since we may require, in the system that we are modelling, that a joint degree consist of two different subjects. To specify this condition we have two options:

- Explicitly list all joint degree combinations which are valid.
- Define an invaraint property on the structure `JointDegree` to specify that the first component cannot be the same as the second component.

The second option is better since an explicit statement of the invariant property improves the understandibility of the specification. Furthermore, using an invariant property follows the principle of encapsulation and makes the specification simpler to extend. For example, if the `Subjects` class is to be extended to include a new literal then this change should be possible without affecting the classes which use the `Subjects` class. This is not possible with the first option, in which the principle of encapsulation has to be broken for the behaviour of the degree class to be well defined. The `JointDegree` structure is respecified in class `Joints3` to incorporate the new invariant property.

All the members of a class now correspond to the literals and the structure expressions whose component values fulfil the relevant structure invariant(s) (if there are any). Invariant properties

```
CLASS Joints3 USING Subject OPNS
STRUCTURES: JointDegree<Subject, Subject>
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: JointDegree(Subject1, Subject2) REQUIRES Subject1..neq(Subject2)
EQNS
JointDegree(Subject1,Subject2)..studies(Subject3) =
(Subject1..eq(Subject3)).or(Subject2..eq(Subject3))
ENDCLASS (* Joints3 *)
```

introduce the possibility of 'run time' errors in an execution model. For example, consider an extension to the `JointDegrees` class in which a new transformer operation allows either of the `Subject` fields to be changed. Now, a transformer service may result in a new state which does not fulfil the invariant property. In this case the behaviour of the resulting object is undefined.

The invariant above is termed a **structure invariant** because the invariant property is defined in terms of properties of the components of a structure of the class. It is also often desirable to be able to define an invariant on a whole class rather than a structure in a class. The syntax of such a **class invariant** is illustrated by the `MathsJoints` specification below. Class invariant properties are required to be true for all literal values (this is checked by a static analysis). This requirement makes the specification of class invariants much easier to transform (during pre-processing) into a set of structure invariants. It also makes the concept of class invariant much easier to understand — why define a literal value which does not fulfil an invariant property?

```
CLASS MathsJoints USING Subject OPNS
STRUCTURES: JointDegree(Subject, Subject)
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: MathsJoints1..studies(Maths)
EQNS
JointDegree(Subject1, Subject2)..studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* MathsJoints *)
```

Note that the class invariant mechanism is simply syntactic sugaring for defining sets of structure invariants. For example, the invariant `MathsJoints1..studies(Maths)` can be re-written as: `JointDegree(Subject1,Subject2) REQUIRES (Subject1..eq(Maths))..or(Subject2..eq(Maths))`. As the external attributes of a structured class are defined in terms of the external attributes of its components, a class invariant is just a more concise way of expressing a set of **structure invariants**. Consequently, in mapping OO ACT ONE to the O-LSTS model we must consider only how to map structure invariants: **class invariants** are removed by a simple pre-processing of an OO ACT ONE specification.

Structure invariants correspond to boolean conditions of **conditioned parameterised typed state labels** in the O-LSTS model, represented as **state label expressions** of **type** Bool. Invariant properties, in OO ACT ONE, depend on the visibility of a `Bool` class (with members `true` and `false`).

Example 4: An includes (re-use) mechanism

The advantages of having a purely syntactic re-use mechanism is illustrated by the `Floor3s` and `Floor3sa` OO ACT ONE specifications, below.

```
CLASS Floor3s USING Nat13s OPNS
LITERALS: 1,2,3
TRANSFORMERS: up, down
ACCESSORS: where -> Nat13s
EQNS 1.up =2; 2.up=3; 3.up=3; 1.down=1; 2.down=2; 3.down=3;
1..where =1; 2..where=2; 3..where=3
ENDCLASS (* ----------------- Floor3s ----------------- *)
CLASS Floor3sa USING Nat13s OPNS
LITERALS: 1,2,3,off1,off2,off3
TRANSFORMERS: up, down, switch
ACCESSORS: where -> Nat13s
EQNS 1.up =2; 2.up=3; 3.up=3; off1.up=off1; off2.up=off2; off3.up=off3; 1.down=1; 2.down=2;
3.down=3; off1.down=off1; off2.down=off2; off3.down=off3; 1.switch = off1; 2.switch = off2;
3.switch = off3; off1.switch = 1; off2.switch = 2; off3.switch = 3; 1..where =1; 2..where=2;
3..where=3; off1..where =1; off2..where =2; off3..where =3
ENDCLASS(* Floor3sa *)
```

The O-LSTSs corresponding to these OO ACT ONE specifications are defined in the **inclusion** example in figure 3.14. Although there is no subclassing relationship between these two behaviours, it is clear that it is beneficial to be able to define either class in terms of the other. More generally, a mechanism for including some of the operations and equations from one class in a new class definition is required. The `INCLUDES` mechanism is illustrated by the `Floor3s'` specification below.

```
CLASS Floor3s' OPNS INCLUDE FROM Floor3sa
LITERALS: 1,2,3
TRANSFORMERS: up, down
ACCESSORS: where
ENDCLASS (* Floor3s *)
```

This new definition re-uses part of the specification of `Floor3sa`. The operations that are re-used have to be explicitly listed. The equations for these operations do not need to be listed. The equations for the included transformer, accessor and dual operations on the included literals and/or structures are implicit in the new specification. In this example, it is therefore not necessary to define any additional equations. Note that `Floor3s'` is not defined to use `Floor3sa`. The includes mechanism does not copy the classes which are used by the class being included.

The includes mechanism identifies an operation of a class and 'copies' its operation and equation definitions into the new class. (It is not a direct copy since all the occurrences of the old class name have to be replaced by occurrences of the new class name in the included definition part of the new class.) The `includes` mechanism is further sugared to give an `includesall` mechanism which states that all the operations and equations of the specified class are copied into the new specification. The inclusion mechanisms are a sort of MACRO expansion facility. They do not extend the semantics of

the ADT language. OO ACT ONE specifications are preprocessed to remove all include directives.

Example 5: Internal Operations

The specification of internal (or hidden) operations is common to many object oriented programming languages (see chapter 6). It is useful to be able to define attributes of a class which are available only to the class in which they are defined (i.e. attributes which are not part of the external interface). These attributes are then used to help specify the external behaviour. In OO ACT ONE, accessor, dual and transformer operations can be declared as `HIDDEN`. This is illustrated by the specification of a simple store of natural numbers, below.

```
CLASS Store USING Nat, Bool OPNS
LITERALS: empty STRUCTURES: st<Store, Nat>
TRANSFORMERS: add<Nat>
ACCESSORS: average -> Bool, sum -> Nat (* HIDDEN *), size -> Nat (* HIDDEN *)
EQNS
empty..average = ~Nat; empty..sum = 0; empty..size = 0;
st(Store1,Nat1)..average = (st(Store1,Nat1)..sum)..div(st(Store1,Nat1)..size);
st(Store1, Nat1)..sum = Nat1.+(Store1..sum); st(Store1, Nat1)..size = 1.+(Store1..sum)
ENDCLASS (* Store *)
```

In this specification, the `HIDDEN` operations are defined to simplify the definition of the external operation `average`. The `HIDDEN` operations have a direct correspondence with the elements of $HUTT$ and $HVTT$ (the **hidden typed transitions**) in the O-LSTS model. The definition of **state label expressions** constrains hidden operations to being used only inside the class in which they are defined. This constraint must be checked during the static analysis of OO ACT ONE specifications.

Example 6: Preconditions

A parameterised equation definition, defined on a class or a structure of a class, can be preconditioned by a **state label expression** of **type Bool**. This expression must be parameterised on a (non strict) subset of the parameters in the equation definition (Such an expression can be represented in general form as `Pre(p1,..., pn)`, where `p1` to `pn` are the parameters of the parameterised equation.). Preconditions are a powerful mechanism for simplifying specifications and improving their understandibility. The syntax of the precondition mechanism in OO ACT ONE is illustrated by the specification of class `Queue`, below.

The `Queue` example shows the syntax for defining structure preconditions on dual operations. The syntax for defining the results of preconditioned transformer and accessor operations is the same as the first and second parts, respectively, of the dual syntax (the parts separated by `AND`). The mapping between preconditioned equations and the O-LSTS model is straightforward. It is detailed in appendix A.

```
CLASS Queue USING Nat, Bool OPNS
LITERALS: empty STRUCTURES: Q<Queue, Nat>
TRANSFORMERS: add<Nat>
ACCESSORS: is-empty -> Bool (*HIDDEN*)
DUALS: rem -> Nat
EQNS
empty..is-empty = true; Q(Queue1, Nat1)..is-empty = false;
empty.add(Nat1) = Q(empty, Nat1);
Q(Queue1, Nat1).add(Nat2) = Q(Q(Queue1, Nat1),Nat2);
empty.rem = empty AND ~Nat;
Queue1..is-empty =>
Q(Queue1, Nat1).rem = empty AND Nat1 OTHERWISE Q(Queue1.rem, Nat1) AND Queue1..rem
ENDCLASS (*Queue*)
```

Example 7: Generic (Parameterised) Classes

Genericity is a powerful mechanism. A generic class does not specify a behaviour in the sense that it corresponds to one O-LSTS specification: it acts as a template (or structure) from which other classes of behaviour can be constructed. A generic class is parameterised on the classes which it uses. An instance of a generic class is created through an actualisation of the class parameters. The OO ACT ONE generic mechanism (like many aspects of the syntax) is based on the corresponding mechanism in ACT ONE (with a few syntactic differences to reinforce the object oriented flavour of the language). Classes of behaviour which are defined as instances of generic classes can be transformed into classes which are not defined generically.

It should be noted that genericity is not a subclassing mechanism. Users of other object oriented languages, in which subclassing is not formally defined, often argue that a generic class is a superclass of its instances. However, we argue that a generic class is not a class in its own right, it is a template for creating classes. It is possible to define a generic class such that there are subclassing relationships between instances, depending on the actual parameterisation of the instantiated classes. In figure=Sem-Chp3/Figures we believe that genericity and subclassing are two very different concepts and as such they should be kept distinct. Consequently, we choose to define OO ACT ONE so that generic classes cannot be related by 'sets of subclassing' relationships. Thus, we do not allow generic classes to be specialised, generalised, extended or restricted (see examples 9 to 13), although instances of these classes can be used in this way. The genericity syntax is illustrated by the `Pair` class below.

It is not necessary to further expand on the semantics of generic classes: they are handled in the same way as generic types in ACT ONE. In fact, they map directly onto the ACT ONE generic construct when we generate an ACT ONE model from the OO ACT ONE requirements.

Example 8: Renaming

It is often useful to be able to define a new class to exhibit the same semantic behaviour as another but to have a different syntactic representation. In OO ACT ONE we allow one class to be defined by renaming the operation string identifiers of another class. For example, consider the specification

```
GENCLASS Pair USING Bool, GCLASS elementA, GCLASS elementB
GCLASS elementA GOPNS
GACCESSORS: eq<elementA> -> Bool
ENDGCLASS (* elementA *)
GCLASS elementB GOPNS
GACCESSORS: eq<elementB> -> Bool
ENDGCLASS (* elementB *)
OPNS
STRUCTURES: P<elementA, elementB>
ACCESSORS: eq1<elementA> -> Bool, eq2<elementB> -> Bool
TRANSFORMERS: set1<elementA>, set2<elementB>
EQNS
P(elementA1, elementB1)..eq1(elementA2) = elementA1..eq(elementA2);
P(elementA1, elementB1)..eq2(elementB2) = elementB1..eq(elementB2)
ENDCLASS (* Pair *)
```

of the class `TwoNats` given below.

```
CLASS TwoNats RENAMES NatPair
STRUCTURES: P WITH A2Nat
TRANSFORMERS: set1 WITH change1, set2 WITH change2
ENDCLASS (*TwoNats*)
```

The structure operation `P` and the transformer operations `set1` and `set2` are renamed in the new `TwoNats` class specification. By default, all operations which are not renamed retain their original names. The renaming is removed by a simple pre-processing.

Examples 1 to 8 show the object based mechanics of OO ACT ONE (i.e. the object oriented mechanics without class relationships). OO ACT ONE restricted to this syntax is called OB ACT ONE[15]. All valid OB ACT ONE specifications are valid OO ACT ONE specifications. The OB ACT ONE syntax allows for the formal specification of:

- Classes of behaviour which are protected behind strict interfaces.
- The composition of predefined classes into new (more complex) classes of behaviour.
- Invariant properties which all class members must uphold.
- Internal (hidden) operations.

It is necessary to extend the object based mechanisms with subclassing facilities. In particular, we wish to be able to define a new class to be a subclass (or superclass) of an already existing class and be guaranteed that the corresponding O-LSTSs are related by the formal subclassing relationship. Four such mechanisms are provided in OO ACT ONE, namely extension, restriction, specialisation and generalisation. These correspond to the relationships between O-LSTSs which are defined in section 3.3.3. These four mechanisms, together with a mechanism which combines specialisation and extension, provide the only means of explicitly defining class relationships in OO ACT ONE.

[15]Object Based ACT ONE.

Example 9: Extension

In the O-LSTS semantics, a class A is defined to be an extension of a class B when A offers the behaviour of B together with some additional behaviour. Furthermore, all the members of A must be members of B. This notion of **extension** has a corresponding mechanism in OO ACT ONE. Reconsider the O-LSTS specifications of $TwoCount$ and $DoubleCount$ given earlier in figure 3.7. The OO ACT ONE specification of class `TwoCount` is given below.

```
CLASS TwoCount OPNS
LITERALS: 1,2 TRANSFORMERS: inc, dec
EQNS 1.inc =2; 2.inc =2; 1.dec =1; 2.dec =1
ENDCLASS (*TwoCount*)
```

The `EXTENDS` mechanism can be used to explicilty define the class `DoubleCount` as a subclass of `TwoCount`. This new class, `DoubleCount2` say, is defined in OO ACT ONE below. The O-LSTSD which corresponds to `DoubleCount2` also given in figure 3.19; this should be compared with `DoubleCount` in figure 3.7. Note that the environment of the new class records the explicit subclassing relationship between `DoubleCount2` and `TwoCount`.

```
CLASS DoubleCount USING TwoCount
EXTENDS TwoCount WITH OPNS
TRANSFORMERS: switch
EQNS
1.switch = 2; 2.switch = 1
ENDCLASS (* DoubleCount *)
```

Class `TwoCount` corresponds to the O-LSTS specification given in figure 3.7.

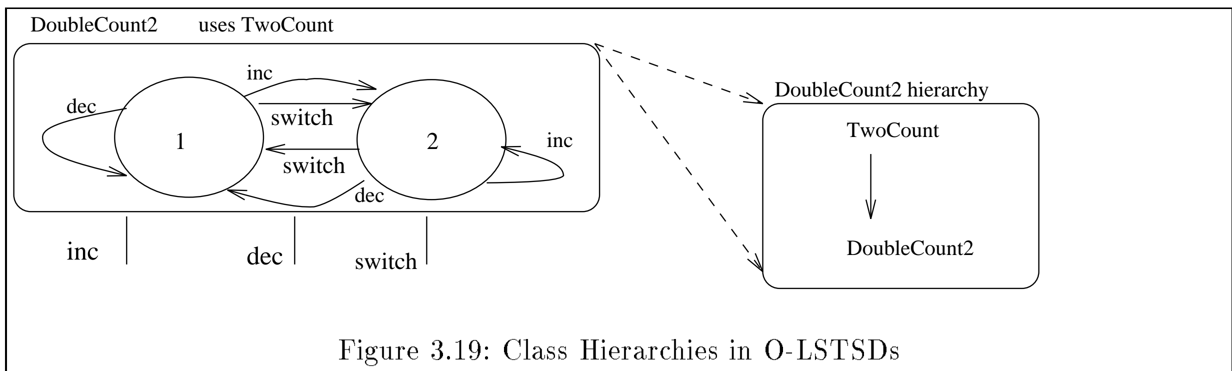


Figure 3.19: Class Hierarchies in O-LSTSDs

The environment of $TwoCount$ is trivial: $Env_{TwoCount} =< \{\}, \{\} >$. `DoubleCount2` corresponds to the O-LSTS specification in the figure 3.19 with a non-trivial environment:
$Env_{DoubleCount2} =< \{TwoCount\}, \{< DoubleCount2, TwoCount >\} >$. Note that the same object based behaviour could have been defined using the `INCLUDESALL` mechanism. In this instance, the O-LSTS 7-tuple corresponding to the OO ACT ONE specification would be the same, but the environment of the system would no longer explicitly acknowledge the subclassing relationship.

The syntactic constraints of the `EXTENSION` mechanism prevent new literals or structures from being defined on a class extension. Furthermore, invariant properties cannot be strengthened or weakened (i.e. the sets of invariants which are defined for the original class are precisely the set of invariants defined for the new extended subclass).

The explicit definition of the subclassing relationship `DoubleCount2` $\sqsubseteq$ `TwoCount` is significant in the semantics of **state label expressions**. Any **state label expression** in which an attribute parameter of class `TwoCount` is expected can accept a parameter of the class `DoubleCount2`.

Example 10: Restriction

Restriction is the inverse relation of extension. Given the original `DoubleCount` specification we would like to be able to define a class `TwoCount` as a superclass (restriction) of `DoubleCount`. A new class can be defined as a restriction of another class by defining a strict subset of the accessor, transformer and dual operations of the class being restricted. Given the following OO ACT ONE specification of class `DoubleCount`, it is possible to define a new class `TwoCount2` as a restriction of `DoubleCount`. This is illustrated below.

```
CLASS DoubleCount
DoubleCount OPNS
LITERALS: 1,2
TRANSFORMERS: inc, dec, switch
EQNS
1.inc =2; 2.inc =2; 1.dec = 1; 2.dec =1; 1.switch = 2; 2.switch = 1;
ENDCLASS (* DoubleCount *)
CLASS TwoCount USING DoubleCount RESTRICTS DoubleCount TO OPNS
TRANSFORMERS: inc, dec
ENDCLASS (* TwoCount *)
```

Example 11: Specialisation

Reconsider the O-LSTSs in figure 3.9. `Nat12s` is defined explicitly (by its environment) to be a subclass of `Nat15s`. These behaviours were only partially specified in the previous section so, for completeness, the actual behaviours are defined below.

The `SPECIALISES` construct requires that the new class must explicitly identify the literals and structures which it has in common with its superclass. (The static analysis of an OO ACT ONE specification must verify that these members form a valid partition of the original class — see 3.4.3).

Example 12: Generalises

Generalisation is the inverse of specialisation. In figure 3.9, `Move12s` **gen** `Move15s`. The two classes of behaviour are defined below.

The `Move12s` class specification illustrates how one class which generalises another offers the complete behaviour of the other class as a partition of itself. We extend the generalises mechanism to

```
CLASS Nat15s USING Nat, Bool OPNS
LITERALS: 1,2,3,4,5
ACCESSORS: eq<Nat> -> Bool
EQNS
1..eq(Nat1) = succ(0)..eq(Nat1); 2..eq(Nat1) = succ(succ(0))..eq(Nat1);
3..eq(Nat1) = succ(succ(succ(0)))..eq(Nat1);
4..eq(Nat1) = succ(succ(succ(succ(0))))..eq(Nat1);
5..eq(Nat1) = succ(succ(succ(succ(succ(0)))))..eq(Nat1)
ENDCLASS (* Nat15s *)
CLASS Nat12s USING Nat15s SPECIALISES Nat15s TO OPNS LITERALS: 1,2
ENDCLASS (* Nat12s *)
```

```
CLASS Move15s USING Bool OPNS
LITERALS: up,down
ACCESSORS: eq<Move15s> -> Bool, curr -> Move15s
EQNS up..eq(up) = true; up..eq(down) = false;
down..eq(up) = false; up..eq(up) = true;
up..curr = up; down..curr = down
ENDCLASS (* ---------------- Move15s ---------------- *)
CLASS Move12s USING Move15s GENERALISES Move15s WITH OPNS
LITERALS: stay
EQNS stay..eq(up) = false; stay..eq(down) = false; stay..curr = ~Move15s
ENDCLASS (* Move12s *)
```

allow one class to be defined as a generalisation of a group of classes. For example, the `stay` member of `Move12s` can be defined as a distinct (though very limited) class of behaviour in its own right. This new class, `JustStay` say, can be generalised with `Move15s` to give the `Move12s` behaviour. This group generalisation is illustrated below.

```
CLASS JustStay USING Move15s OPNS
LITERALS: stay
ACCESSORS: eq<Move15s> -> Bool, curr -> Move15s
EQNS stay..eq(up) = false; stay..eq(down) = false; stay..curr = ~Move15s
ENDCLASS (* ---------------------- JustStay ---------------------- *)
CLASS Move12s USING Move15s, JustStay GENERALISES Move15s, JustStay
ENDCLASS (* Move12s *)
```

Subclassing: contravariance and covariance mechanisms

The classification examples given above do not consider subclassing relationships in which the rules of contravariance and covariance are exploited: all the parameter types are defined by default to be the same in the subclasses as they are in the superclasses. It is necessary to override this default in three different cases:

- i) The class parameters of a structure operation of a subclass may be defined as subclasses of the corresponding parameters in the superclass.

- ii) The result of a valued transition in a subclass may be defined as a subclass of the corresponding result type in the superclass.
- iii) The parameters of a transition in a subclass may be defined as superclasses of the corresponding parameters in the superclass.

These non-default options are illustrated by the O-LSTS specifications of `Lift12s` and `Lift15s` in figure 3.9. All three non-default options are taken by `Lift12s` as a subclass of `Lift15s`. These two classes of behaviour are not connected by an extension or specialisation relationship alone. The behaviour of `Lift12s` is a combination of an extension and specialisation of `Lift15s`. In OO ACT ONE we make this relationship explicit by defining `Lift12s SPECIALISES AND EXTENDS Lift15s`. The complete OO ACT ONE specifications corresponding to the two O-LSTSs are given below.

```
CLASS Lift15s USING Move15s, Nat15s OPNS
STRUCTURES: L<Nat15s>
TRANSFORMERS: M<Move15s>
ACCESSORS: curr -> Nat15s
EQNS L(1).M(up) = L(2); L(2).M(up) = L(2);
L(3).M(up) = L(4); L(4).M(up) = L(5); L(5).M(up) = L(5);
L(1).M(down) = L(1); L(2).M(down) = L(1);
L(3).M(down) = L(3); L(4).M(down) = L(3); L(5).M(down) = L(4);
L(1).curr =1; L(2).curr =2; L(3).curr =3
ENDCLASS (* ------------------ Lift15s ------------------ *)
CLASS Lift12s USING Move12s, Nat12s SPECIALISES AND EXTENDS Lift12s TO OPNS
STRUCTURES: L<Nat12s>
TRANSFORMERS: M<Move12s>
ACCESSORS: Curr<Nat12s>
EQNS L(1).M(stay) = L(1); L(2).M(stay) = L(2); L(3).M(stay) = L(3)
ENDCLASS (* Lift12s *)
```

This example illustrates the only combination of class relationships which can be defined by one mechanism: `SPECIALISES AND EXTENDS` . A `SPECIALISES AND EXTENDS` B states that it is possible to define a class C such that A `EXTENDS` C and C `SPECIALISES` B. Using the combination mechanism means that the class C does not need to be explicitly defined. OO ACT ONE does not define any other 'combination mechanisms': in our experience `SPECIALISES AND EXTENDS` is the only combination mechanism which is used as often as the other singular mechanisms (when provided). Furthermore, it is the only combination mechanism which is straightforward to statically analyse.

A Note On Invariant Properties

It is important to note that all invariant properties are inherited by a subclass from its superclasses.

3.4.3 Static Analysis of OO ACT ONE: Syntax and Semantics

OO ACT ONE specifications are statically analysed as shown in figure 3.20.

Appendix B1 examines the preprocessing of an OO ACT ONE specification for the removal of syntactic sugar. Appendix B2 explains the static analysis of OO ACT ONE specifications which

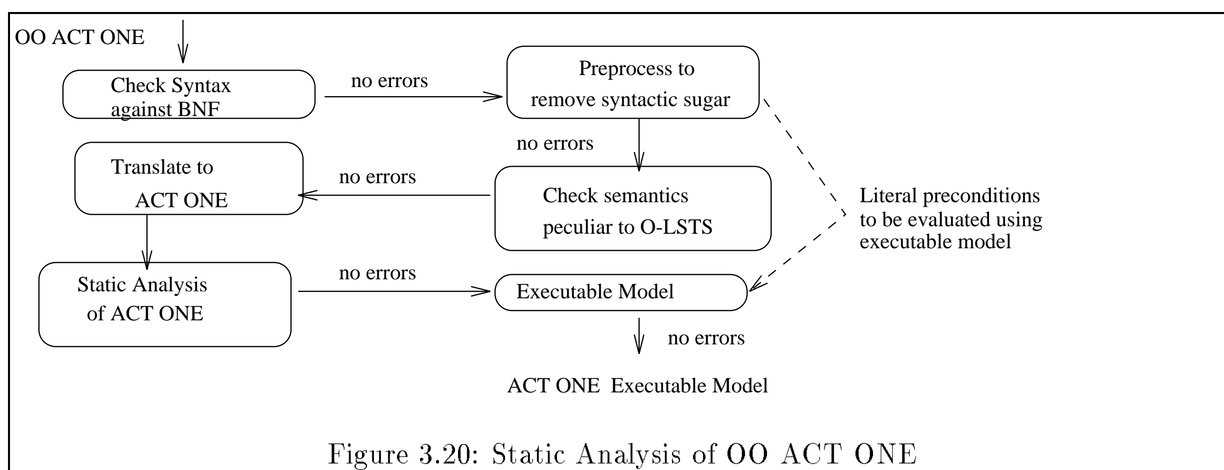


Figure 3.20: Static Analysis of OO ACT ONE

cannot be done by translating the specification to ACT ONE and letting the ACT ONE static analysis tools check the new code.

3.5 An ACT ONE Execution Model for O-LSTS Specifications

3.5.1 The Advantages of Using ACT ONE

Translating an O-LSTS specification, written in OO ACT ONE, into ACT ONE has three advantages:

- It more formally relates the object oriented notions captured in the O-LSTS semantics with the ADT concepts of type, sort, operation, equation, expression and value.
- ACT ONE has a number of associated tools for the static analysis of syntax and semantics and the evaluation of expressions. These tools can be used to complete the static analysis of the OO ACT ONE specification from which the ACT ONE was developed, and to test the dynamic behaviour being modelled.
- ACT ONE forms a part of full LOTOS, which combines the ADT specifications within a process algebra as a single coherent framework. This thesis proposes to use LOTOS as an object oriented design language. The step from formal object oriented analysis to formal object oriented design is made simpler by choosing ACT ONE as the foundation upon which our figure=Sem-Chp3/Figures language is modelled.

3.5.2 Reviewing the ACT ONE Terminology

In ACT ONE, a type specification may define a number of different sorts. Every sort corresponds to a set of terms, with each term representing a particular value of that sort. Equations define equivalences between terms which represent the same value. Each equivalence class of terms represents one value and members of the same equivalence class are identified by that value. The member of the equivalence class which is used to represent the value of the class is normally taken to be the term which all the other terms are 'evaluated to' when the re-writes (as defined by the equations) are applied.

An operation relates 0,1 or more input sorts to a result sort. Operations are term generators. Operations with no input terms are called **literals**. These are commonly used to represent the value of the equivalence class of terms to which they belong. Furthermore, they are the base terms from which other terms can be generated.

3.5.3 An Overview of the OO ACT ONE → ACT ONE Translation

The translation from OO ACT ONE to ACT ONE is best described in stages. Preprocessing has removed OO ACT ONE syntactic sugar and so there are just five object based elements of the syntax to consider:

- `CLASSES`, which are the fundamental behaviour building blocks.
- The `USING` mechanism, which defines a dependency between classes.
- `OPNS`, which define the external interface and set of members for each class.
- `EQNS`, which define the dynamic behaviour of each class member.
- `INVARIANTS`, which define properties fulfilled by members of the classes in which they are defined.

In addition, there are five explicit classification mechanisms to be mapped to ACT ONE. Appendix C examines the semantics of the mapping from OO ACT ONE to ACT ONE.

Additonal Object Based Mechanics

The ACT ONE mechanisms for defining the behaviour of an object which changes state and returns a value in response to a service request are called the **dual mechanics**. Other object based 'mechanics' are required to define the behaviour of the implicit unspecified members of OO ACT ONE classes. Further, additional operations (internal tests) are defined to simplify the specification of the object oriented execution model. The mechanics defined in each of these cases is as follows:

- **Dual Mechanics**
  Values of an ACT ONE sort corresponding to members of an OO ACT ONE class have two types of representation, namely singular and dual. Singular representations are literals or structure expressions. Dual representations of a class `C` are pairs of values whose first element corresponds to a singular representation of the class `C` and whose second element corresponds to any sort value which represents a member of the **result type** of an accessor or dual operation of `C`. For every **result type**, `D` say, of a class `C` there is an operation `dualCD : C,D -> C` which is used to construct the corresponding dual representation. The external attributes (as defined by the accessor, dual and transformer operations) of dual representations of values in `C` are defined as the attribute operations applied to the first element of the dual expresssion.

  Singular representations are generated by transformer operations on singular or dual representations. Dual representations are generated by accessor or dual operations. In OO ACT ONE we represent the newstate of an object, `obj` say, after servicing a request `req` say, by the **state label expression** `obj.req`. Similarly, we represent the value returned by an accessor or dual

operation as `obj..req`. **State label expressions** in OO ACT ONE are mapped into ACT ONE as follows:

- `obj.req` → `.(req(obj))`
- `obj.req(p1,...,pn)` → `.(req(obj,p1,...,pn))`
- `obj..req` → `ClassResult(req(obj))`, where `Class` is the **result type** of `req`.
- `obj..req(p1,...,pn)` → `ClassResult(req(obj,p1,...,pn))`, where `Class` is the sort generated from the **result type** of `req`.

This ACT ONE representation of OO ACT ONE **state label expressions** requires the specification of two additional operations:

- `.:C -> C` is defined for every class `C`.
- `DResult: C -> D` is defined for every **result type** `D` in class `C`.

- **Unspecified Values**
  Implicit in every OO ACT ONE specification are the unspecified literal values of each class. For example, ~C represents the unspecified literal value of class C. In translating to ACT ONE, we generate an operation `unspecC: -> C` for every sort `C`. All operations on this value are defined to return the unspecified value of the appropriate class (by default). Static analysis of an OO ACT ONE specification determines when such defaults are overridden by the specifier.
- **Internal Tests**
  For every class `C` we define an operation `CRep: C -> Bool` which returns true if the input parameter of the `CRep` operation is represented in singular form. This internal test operation is used to simplify the specification of the object oriented 'mechanics'.

3.5.3.2 Example Object Based Behaviours in ACT ONE

The ACT ONE specifications that follow result from translating object based behaviour as specified in OO ACT ONE. (The code that is listed is slightly different from the ACT ONE code that is produced because the object oriented features which allow this class to be defined as a superclass of a new class are not included. We consider such object oriented concerns in section 3.5.3.3.) The three examples are used to illustrate different aspects of the object based properties specified by the ACT ONE code.

Example 1: `Nat` behaviour

Consider the OO ACT ONE `Nat` class specified in example 1 of section 3.4.2 (and its corresponding O-LSTSD in figure 3.18). The ACT ONE code is given below.

There are a number of things to note about this specification:

- Although class `Bool` is not specified as being used by class `Nat` in the OO ACT ONE requirements, a Boolean type with sort Bool is used in the ACT ONE specification of sort `Nat`. Every sort generated from an OO ACT ONE specification is defined in terms of boolean behaviour (it is necessary for defining the internal object oriented mechanisms). Consequently, the type

```
TYPE Nat IS Boolean SORTS Nat OPNS
0: -> Nat (* Literal*)
succ:  Nat -> Nat (* Structure *)
inc:  Nat -> Nat (* Transformer *)
previous:  Nat -> Nat (* Dual Accessor Nat *)
unspecNat:  -> Nat
.:  Nat -> Nat
NatResult:  Nat -> Nat
dualNatNat:  Nat, Nat -> Nat
NatRep:  Nat -> Bool
EQNS FORALL Nat1, Nat2:  Nat
OFSORT Nat
inc(Nat1) = succ(Nat1); inc(unspecNat) = unspecNat;
inc(dualNatNat(Nat1,Nat2)) = inc(Nat1);
previous(0) = dualNatNat(0,unspecNat); previous(unspecNat) = unspecNat;
previous(succ(Nat1)) = dualNatNat(succ(Nat1), Nat1);
previous(dualNatNat(Nat1, Nat2)) = previous(Nat1);
NatRep(Nat1) => .(Nat1) = Nat1; .(dualNatNat(Nat1, Nat2)) = Nat1;
NatResult(dualNatNat(Nat1,Nat2)) = Nat2;
OFSORT Bool
NatRep(0) =true; NatRep(succ(Nat1)) = true; NatRep(unspecNat) = true;
NatRep(dualNatNat(Nat1,Nat2)) = false
ENDTYPE (* Nat *)
```

Boolean with sort Bool is an integral part of the resulting ACT ONE code. This class is specified to exhibit the well understood behaviour of booleans (all the normal operations are available as transformer operations) together with the unspecified value unspecBool.

- The ACT ONE specification has an intuitively object oriented style. Ignoring the additional object oriented mechanics (which are generated in the same way for all behaviours), we have a clear and concise correspondence between the ACT ONE and the OO ACT ONE from which it was generated.
- The list of variables after the forall clause in the ACT ONE code is defined to exactly match the variable parameters used in the equation definitions.
- The translation to ACT ONE produces very inefficient code. However, efficiency is not important at this theoretical stage of development.

Example 2: System behaviour

Consider the OO ACT ONE System class also specified in example 1 of section 3.4.2 (and its corresponding O-LSTSD in figure 3.16). In this example we assume the ACT ONE code for class Stack has been generated in sort Stack defined in the type of the same name. The ACT ONE code which is generated from the class System specification is given below.

This specification shows quite clearly the way in which the components of the System (i.e. the Stacks) are used through their external interfaces alone to provide the external behaviour of their containing object.

```
TYPE System IS Boolean, Stack SORTS System OPNS
sys:  Stack, Stack -> System (* Structure *)
push:  System, Nat -> System (* Transformer *)
move:  System -> System (* Transformer *)
pop:  System -> System (* Dual Nat *)
unspecSystem:  -> System
.:  System -> System
NatResult:  System -> Nat
dualSystemNat:  System, Nat -> System
SystemRep:  System -> Bool
EQNS FORALL System1:  System, Nat1, Nat2:  Nat, Stack1,Stack2:  Stack
OFSORT System
push(sys(Stack1, Stack2), Nat1) = sys(.(push(Stack1, Nat)), Stack2);
push(unspecSystem, Nat1) = unspecSystem;
push(dualSystemSystem(System1,Nat1), Nat2) = push(System1, Nat2);
move(sys(Stack1, Stack2)) =
sys(.(pop(Stack1)), .(push(Stack2, NatResult(pop(Stack1)))));
move(unspecSystem) = unspecSystem;
move(dualSystemSystem(System1,Nat1), Nat2) = move(System1, Nat2);
pop(sys(Stack1, Stack2)) =
dualSystemNat( sys(Stack1, .(pop(Stack2))), NatResult(pop(Stack2)));
pop(unspecSystem) = unspecSystem;
pop(dualSystemSystem(System1,Nat1)) = pop(System1);
SystemRep(System1) => .(System1) = System1;
.(dualSystemNat(System1, Nat1)) = System1;
NatResult(dualSystemNat(System1,Nat1)) = Nat1;
OFSORT Bool
SystemRep(sys(Stack1, Stack2)) = true; SystemRep(unspecSystem) = true;
SystemRep(dualSystemNat(System1,Nat1)) = false
ENDTYPE (* Nat *)
```

Example 3: Preconditions in the specification of `Queue` behaviour

The ACT ONE specification of the sort `Queue` given in Appendix C2 is used to illustrate the mapping of preconditioned expressions to ACT ONE. A less important feature of this example is the mapping of a hidden operation.

3.5.3.3 Translating Object Oriented Requirements

An Overview

To translate object oriented requirements specified in OO ACT ONE to ACT ONE, it is necessary to group together classes of behaviour which are related by subclassing relationships into one type definition in ACT ONE. Consider the OO ACT ONE specifications of `Lift12s` and `Lift15s` defined at the end of section 3.3.3.5 (and their corresponding O-LSTSDs given in figure 3.9). These classes of behaviour are translated into the framework of ACT ONE code given below in the type definition of `Lift12sRoot`.

```
TYPE Lift12sRoot IS Move12sRoot, Nat12sRoot
SORTS Lift12s (* using Move12s, Nat12s, Lift15s *)
Lift15s (* using Move15s, Nat15s:  superclass Lift12s *)
OPNS ...EQNS ...ENDTYPE
TYPE Move12sRoot IS Boolean
SORTS Move12s (* using Bool, Move15s *)
Move15s (* using Bool:  superclass Move12s *)
OPNS ...EQNS ...ENDTYPE
TYPE Nat15sRoot IS Boolean, Nat
SORTS Move12s (* using Bool, Nat *)
Nat12s (* using Bool, Nat:  superclass Nat15s *)
OPNS ...EQNS ...ENDTYPE
```

The three classes of object oriented behaviour that are modelled in this ACT ONE header are `Lift12s`, `Move12s` and `Nat15s`, i.e the root classes of the separate trees in the class hierarchy of `Lift12s`. It is possible to execute a dynamic model of the other non-root classes but there is no guarantee that the environments of the these classes are correctly specified. To guarantee that the environment of a class, `C` say, is correctly modelled in ACT ONE, it is sufficient to restrict the classes listed in the OO ACT ONE code to be only those classes visible to `C`.

Class Hierarchy Mechanisms

There are two important aspects of modelling object oriented behaviour:

- **Polymorphism**:

  In the object oriented paradigm, subclassing means that a member of one class is also a member of each of its superclasses. Consequently, anywhere a member of one of its superclasses is used by an object, all the members of the subclass must also be able to be used. This is inclusion polymorphism. In OO ACT ONE there is one general instance of this rule, namely in a parameterised operation we require that an actual parameter is a member of the parameter class or a member of a subset of the parameter class. The paramaterised operations which need to be considered are: structures, accessors, transformers and duals.

  When generating ACT ONE code from an OO ACT ONE specification we model polymorphism by defining all the parameterised operations on any combination of valid parameter. This is done by defining coercion routines between any two classes related by a subclassing relationship. Subclass parameters are coerced into being the corresponding members of the required superclass. The coercion operations are defined as follows: for every pair of classes, `C1` and `C2` say, such that $C1 \sqsubseteq C2$ in the environment of the class being modelled, then there is an operation `C1toC2:  C1 -> C2` defined in the type of the class to which these classes are rooted. The operation `C1toC2` is defined by a set of equations which equate all literal values of `C1` to the same literal values of `C2`. Further, all subclass `structure` expressions are coerced to superclass `structure` expressions by applying a suitable coercion to the component values.

- **Inheritance:**

  The existence of a subclassing hierarchy suggests that there is a duplication of behaviour between

superclasses and subclasses. This duplication can be taken advantage of when creating an object oriented model of OO ACT ONE requirements in ACT ONE. The behaviour defined by an operation applied to a member of a class is always, whenever possible, inherited from a superclass[16] of that class. In ACT ONE we encode this inheritance by specifying explicit root definitions for operations which are not inherited from superclasses. These root operations (defined as `ClassOpn` where `Class` is the name of the class in which the behaviour is rooted and `Opn` is the name of the operation) are then used in the subclass definitions to avoid duplication. Coercion plays an important part in this inheritance mechanism since it is now also necessary to be able to coerce superclass values into subclass values.

The additional machinery required to model this object oriented behaviour further complicates the ACT ONE code. In general, the ACT ONE code is between four and twenty times larger[17] than the OO ACT ONE from which it was generated. An example of the ACT ONE code arising from the translation of object oriented properties in OO ACT ONE is given in Appendix C3.

3.5.4 Static Analysis of ACT ONE

The static analysis of ACT ONE code generated from OO ACT ONE guarantees certain correctness properties of the underlying O-LSTS model. The ACT ONE static analyser checks the types of all parameters in equation definitions (their visibilty and compatibility). It also checks the correctness of all **state label expressions** in the O-LSTS model specified using OO ACT ONE. Thus, by translating to ACT ONE, the most difficult static analysis is performed by an already existing tool set. One problem is that static analysis errors identified in the ACT ONE code have to be translated back into meaningful OO ACT ONE errors. It should be clear that, although such a mechanism is not formulated in this work, providing an object oriented interpretation of these ACT ONE errors is not a difficult task.

3.5.5 Evaluating Act One Expressions: An Execution Model for OO ACT ONE

To model the behaviour of an object in response to a message request at its interface it is necessary only to evaluate an ACT ONE expression. For example, reconsider the `System` behaviour considered earlier in section 3.5.3.2. To model the effect of a `pop` request at the object represented by `sys(Stack1, Stack2)` we evaluate the ACT ONE expression `pop(sys(Stack1,Stack2))`. The result of this expression evaluation is a dual representation. The first element of this pair of values is the newstate of the object, the second element represents the value returned by the `pop` operation.

To model the dynamic behaviour of an object over a period of time it is necessary to create a feedforward loop of expression evaluations in which the result of an expression evaluation is used as the **server** of the next expression to be evaluated. Such loops of behaviour are represented in **event**

[16] When two superclasses offer the same operation we are guaranteed that the behaviour defined in both superclasses is the same. Consequently, when a choice is available, an arbitrary decision is made as to which superclass a subclass inherits from.

[17] Size in this case is an approximation for the number of operations and equations defined.

diagrams (see below). By taking this simple view of object oriented behaviour, the static ADT model is given a dynamic object oriented interpretation.

3.5.6 Event Diagrams

An event diagram for the behaviour of a `System` object is given in figure 3.21, together with the corresponding **event trace**.

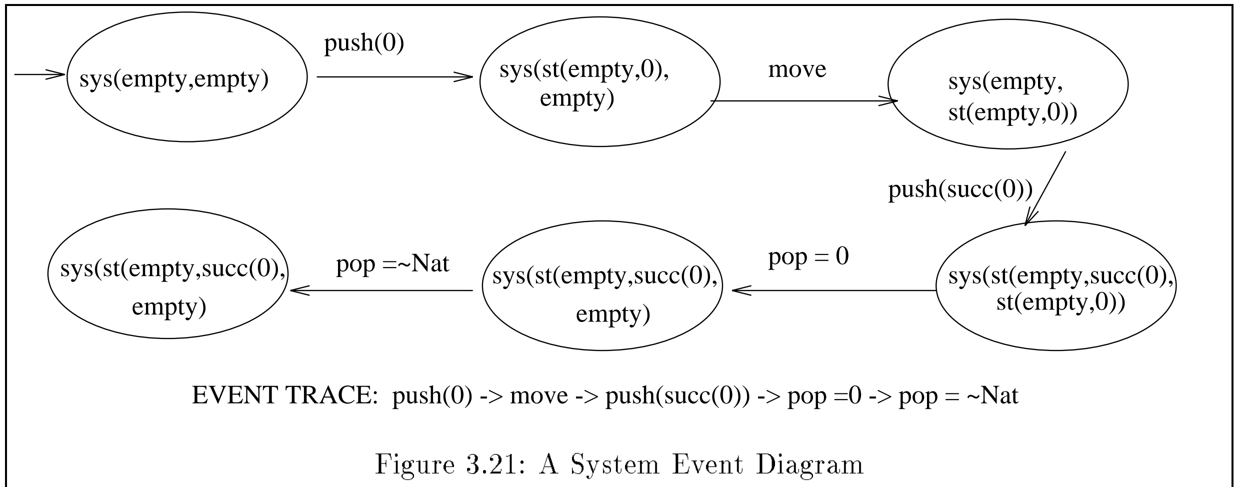


Figure 3.21: A System Event Diagram

An O-LSTS event diagram is simply an an abstraction of the whole O-LSTS(D) in which only a particular set of connected states are represented. The initial state of the system is represented by the only state in the diagram with an incoming arrow which is not rooted in another state. The final state of the system is represented by the only state in the diagram without an outgoing arrow. The sequence of transitions which the system goes through is called an **event trace**. Such traces are common in process algebras: they specify possible behaviours of a process (or system). An O-LSTS gives rise to a peculiar set of **event traces** because of the constraint that an O-LSTS system must always be able to fulfil all of its service requests at all times during its life. The property which distinguishes different instantiations of the same O-LSTS specification are the sequences of values which they return during execution.