

## Appendix A

# Preconditioned Equations: The O-LSTS Model

In this appendix, we define the semantics of preconditions in OO ACT ONE by mapping them to the O-LSTS model. Preconditions are defined for **STRUCTURE** equations, **CLASS** equations and a syntactic sugar defines **total** equations.

### Preconditioned Structure Equations

Preconditioned structure equations are defined for transformer, accessor and dual operation as follows.

- **I) Transformer** preconditions, written as:

$$\begin{aligned} pre_1(P_1, \dots, P_n) &\Rightarrow sc(P_1, \dots, P_{j-1}).tr(P_j, \dots, P_n) = newstate_1 \text{ OTHERWISE } \dots \\ pre_{m-1}(P_1, \dots, P_n) &\Rightarrow newstate_{m-1} \text{ OTHERWISE} \\ newstate_m, &\text{ for some } m, n, j \in \{1, 2, \dots\}, m \geq 2 \end{aligned}$$

correspond to the parameterised set of **unvalued state-to-state** transitions:

- $\langle tr(p_j, \dots, p_n), newstate_1 \rangle \in From_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $pre_1(p_1, \dots, p_n)$
  - $\forall k \in \{2, \dots, m-2\}$ :  
 $\langle tr(p_j, \dots, p_n), newstate_k \rangle \in From_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_1(p_1, \dots, p_n))$  and  $\dots$  and  $not(pre_{k-1}(p_1, \dots, p_n))$  and  $(pre_k(p_1, \dots, p_n))$
  - $\langle tr(p_j, \dots, p_n), newstate_m \rangle \in From_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_i(p_1, \dots, p_n)), \forall i \in \{1, \dots, m-1\}$
- **II) Accessor** preconditions, written as:

$$\begin{aligned} pre_1(P_1, \dots, P_n) &\Rightarrow sc(P_1, \dots, P_{j-1}).acc(P_j, \dots, P_n) = result_1 \text{ OTHERWISE } \dots \\ pre_{m-1}(P_1, \dots, P_n) &\Rightarrow result_{m-1} \text{ OTHERWISE} \\ result_m &\text{ for some } m, n, j \in \{1, 2, \dots\}, m \geq 2 \end{aligned}$$

correspond to the parameterised set of **valued state-to-state** transitions:

- $\langle tr(p_j, \dots, p_n), result_1, sc(p_1, \dots, p_{j-1}) \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $pre_1(p_1, \dots, p_n)$
  - $\forall k \in \{2, \dots, m-2\}$ :  
 $\langle tr(p_j, \dots, p_n), result_k, sc(p_1, \dots, p_{j-1}) \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_1(p_1, \dots, p_n))$  and ... and  $not(pre_{k-1}(p_1, \dots, p_n))$  and  $(pre_k(p_1, \dots, p_n))$
  - $\langle tr(p_j, \dots, p_n), result_m, sc(p_1, \dots, p_{j-1}) \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_i(p_1, \dots, p_n)), \forall i \in \{1, \dots, m-1\}$
- **III) Dual preconditions, written as:**

$$pre_1(P_1, \dots, P_n) \Rightarrow sc(P_1, \dots, P_{j-1}).dual(P_j, \dots, P_n) = newstate_1 \text{ AND } result_1 \text{ OTHERWISE } \dots \\ pre_{m-1}(P_1, \dots, P_n) \Rightarrow newstate_{m-1} \text{ AND } result_{m-1} \text{ OTHERWISE } newstate_m \text{ AND } result_m, \text{ for} \\ \text{some } m, n, j \in \{1, 2, \dots\}, m \geq 2$$

correspond to the parameterised set of **valued state-to-state** transitions:

- $\langle tr(p_j, \dots, p_n), result_1, newstate_1 \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $pre_1(p_1, \dots, p_n)$
- $\forall k \in \{2, \dots, m-2\}$ :  
 $\langle tr(p_j, \dots, p_n), result_k, newstate_k \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_1(p_1, \dots, p_n))$  and ... and  $not(pre_{k-1}(p_1, \dots, p_n))$  and  $(pre_k(p_1, \dots, p_n))$
- $\langle tr(p_j, \dots, p_n), result_m, newstate_k \rangle \in ValFrom_{sc(p_1, \dots, p_{j-1})}, \forall p_1, \dots, p_n$  such that  $not(pre_i(p_1, \dots, p_n)), \forall i \in \{1, \dots, m-1\}$

## Preconditioned Class Equations

Preconditioned class equations are similar to preconditioned structure equations. The only difference is that a set of structure variable parameters are replaced by one parameter which represents all class values. For example, consider **transformer** preconditions, written generally as:

$$pre_1(P_1, \dots, P_n) \Rightarrow Class1.tr(P_j, \dots, P_n) = newstate_1 \text{ OTHERWISE } \dots \text{ OTHERWISE } newstate_m$$

These preconditioned equations correspond to the parameterised set of **unvalued state-to-state** transitions:

- $\forall Class1 \in US(Class), k \in \{1, \dots, m-2\}$ :  
 $\langle tr(p_j, \dots, p_n), newstate_k \rangle \in From_{Class1}, \forall p_1, \dots, p_n$  such that  $not(pre_1(p_1, \dots, p_n))$  and ... and  $not(pre_{k-1}(p_1, \dots, p_n))$  and  $(pre_k(p_1, \dots, p_n))$
- $\forall Class1 \in US(Class), \langle tr(p_j, \dots, p_n), newstate_m \rangle \in From_{Class1}, \forall p_1, \dots, p_n$  such that  $not(pre_i(p_1, \dots, p_n)), \forall i \in \{1, \dots, m-1\}$

Accessor and dual preconditions are defined similarly.

## Total Equations

The behaviour of all the elements of a class in response to an external attribute request can be defined to be equivalent using a preconditioned class equation in which the first precondition is true. For example, `[true] => Class1.tr = sle OTHERWISE ...` specifies that  $\forall c \in US(Class), \langle tr, sle \rangle \in From_c$ . OO ACT ONE provides a more concise way of specifying this behaviour: `Class1.tr = sle`. This is called a **total equation**.

## Appendix B

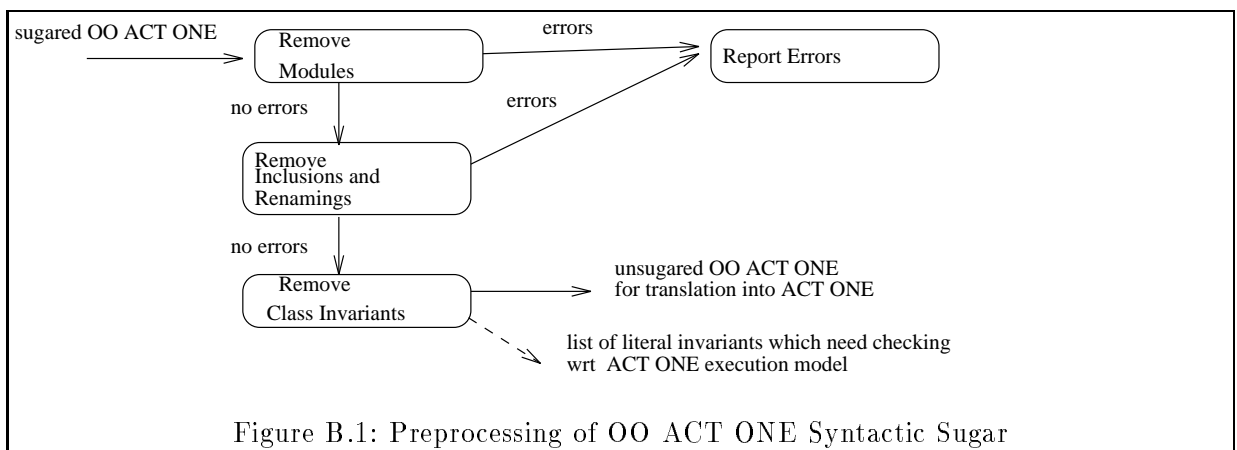
# Static Analysis of OO ACT ONE

### B.1 Preprocessing: Removing Syntactic Sugar

The first step in the static analysis of an OO ACT ONE specification, after the syntax has been checked, is the removal of syntactic sugar. The following syntactic mechanisms have to be removed:

- Modules
- Renaming
- Class Invariants

The diagram in figure B.1 shows the way in which this is achieved.



#### Removing Modules

Removing modules is done in three steps:

- I) Check that all modules are uniquely identified. Return an error otherwise.
- II) For every instance of 'MODULE *Module-name*' in a class definition, 'MODULE *Module-name*' is replaced by the list of classes grouped together by the module definition. If *Module-name* is not defined in the specification then an error is reported.

- III) After changing all module references to references to lists of classes, the module definitions are removed.

### Removing Inclusions and Renamings

After removing generic class definitions and module definitions, the OO ACT ONE specification is made up of a list of class definitions. These class definitions may be mutually dependent, since one class can be defined to: include the operations and equations of another, or to rename the operations and equations of another. It is not possible, in general, to remove these interdependencies by one single pass through the specification. In a specification with  $n$  classes it may take up to  $n$  passes through the specification to remove all the interdependencies. Consequently, we define this preprocessing stage as a loop of passes through the specification. The first pass marks all classes which rename or include part of another class. When one class is marked to depend on another class which isn't marked then this class can be redefined (by a simple syntactic relabelling of the appropriate operation labels) and then unmarked. At the end of every pass through the specification the number of classes marked is checked to see if it has decreased. If not, an error is reported. Once all classes are unmarked, all inclusion and renaming mechanisms have been removed.

### Removing Class Invariants

Translating class invariants into sets of structure invariants is done in two stages.

- First check that all class invariants are true for all literal values in the classes in which the invariants are defined. This check requires a means of evaluating boolean **state label expressions**. The OO ACT ONE execution model (which formalises the meaning of such an evaluation) is defined by a mapping to ACT ONE. Consequently, this pre-processing stage is defined to generate a list of boolean expressions which must evaluate to true (in the ACT ONE framework of evaluation). When such a list is non-empty, a warning is given to say that class invariants for literal values will be checked at a later stage in the analysis (after the preprocessing is complete). Later, if an executable model has been successfully generated in ACT ONE, the literal requirements (expressed as boolean **state label expressions**) are evaluated and an error is returned if any of the expressions are false.
- Secondly, convert class invariants into sets of structure invariants. For every class definition containing a class invariant, represented as:

```
CLASS cname USING ...
STRUCTURES: st1, ..., stn ...
INVARIANTS: cname1 .. sle ... EQNS ... ENDClass,
```

the class is transformed by the preprocessor into:

```
CLASS ..... INVARIANTS: st1 REQUIRES st1..sle, ..., stn REQUIRES stn..sle
EQNS ... ENDClass
```

## B.2 Static Semantic Checks of Unsugared OO ACT ONE

Static semantic checks of O-LSTS behaviour defined in an OO ACT ONE specification fall into two categories:

- Those which are concerned with ‘type checking’ equation definitions, and verifying the visibility of classes used in operation definitions. These checks are performed by a static analysis of the ACT ONE produced from the OO ACT ONE specification.
- Other checks are peculiar to the O-LSTS model and cannot be checked across the mapping to ACT ONE.

The remainder of this appendix examines each of these other requirements in turn and gives an overview of the mechanisms which make these checks.

- **Contravariance, Covariance and Subclassing**

When a subclass is defined to exhibit contravariance and covariance properties with respect to its superclass (or vice versa), additional classification requirements have to be checked:

- Structure parameters in the subclass must be explicitly defined (in the environment of the new class definition) as subclasses of the corresponding structure parameters in the superclass.
- Result parameters in the subclass must be explicitly defined (in the environment of the new class definition) as subclasses of the corresponding result parameters in the superclass.
- Attribute parameters in the subclass must be explicitly defined (in the environment of the new class definition) as superclasses of the corresponding attribute parameters in the superclass.

To make these checks, it is first necessary to create the explicit class hierarchy for each class. Then, the existence of the required subclassing relationships between subclass and superclass parameters is easily verified. An error is returned if the required relationships are not fulfilled.

Note that it may not be possible to generate a class hierarchy if the OO ACT ONE is not well defined. For example, one class may be defined in terms of another class which is defined in terms of the original class. This type of circular dependency is checked for when removing the renaming and inclusion syntactic sugar. It is also tested for during the generation of the class hierarchy (in a similar way). An error is returned if the list of classes being analysed do not have a well defined hierarchical structure with respect to the explicit class relationships specified between them.

- **Checking the use of hidden operations**

As ACT ONE does not facilitate the definition of local operations, it is necessary to check an OO ACT ONE specification to ensure that hidden operations are used only in the class in which they are defined. For every class in an OO ACT ONE specification, the state label expressions

in the equations are analysed to check that operations on classes, other than the one being defined, are not defined as hidden. This analysis is achieved by first producing a list of the hidden operations in each class. An error is returned if a hidden operation of one class is used in the definition of another class.

- **Additional Syntactic Constraints**

Section 3.2.1.1 defines some additional syntactic constraints for O-LSTS specifications. The constraints specify the way in which string identifiers for state constructors and transition names can be defined. Correspondingly, in OO ACT ONE there are syntactic constraints placed on the naming of operations:

All operations must be uniquely categorised (as literal, structure, accessor, transformer or dual) and appear once only in the operation definition.

Another syntactic constraint placed on the O-LSTS model is that the result of a service request and the newstate of an object after servicing a service request must be defined using **state label expressions**. Correspondingly, in OO ACT ONE, we require that:

The right hand side of equation definitions must be expressed as **state label expressions**.

This check is carried out as part of the completeness analysis (see below). An error is returned if either of these conditions are not met.

- **The Definition of the Behaviour of a Class is not Distributed Between Other Classes**

We require that the equations in one class do not specify behaviour for members of another class. Consequently, the left hand side of all equations must be **state label expressions** which have the **server** equal to a member of the class in which the equation is found. This requirement is easily checked by enforcing that all equations have one of the following forms (where **literal** and **structure** are literals or structures respectively of the class in which the equations are defined):

`literal..att = ... or literal.att = ... or structure(...)..att = ... or structure(...)`  
`= ...`

This requirement is verified during the completeness analysis (see below).

- **Completeness Analysis**

The O-LSTS model requires that all states in a class have one, and only one, state transition defined from that state for every attribute of the class. This is called the **completeness condition**. It is more formally defined in section 3.2. Such a requirement cannot be guaranteed through static analysis of the ACT ONE code which is generated from the OO ACT ONE specification.

The completeness analysis of OO ACT ONE specifications depends on the definition of two new concepts: the **Completeness Set** of a class and the **parameterisation** of an operation. These are defined below.

**Definition. Parameterisation:**

The parameterisation of an operation,  $op$  say, written  $Par(op)$ :

$Par(op) = op \Leftrightarrow op$  is unparameterised.

Given a parameterised operation, written  $op < C_1, \dots, C_n >$ ,  $Par(op) = op(C_1x_1, \dots, C_nx_n)$ , where

$x_i \in \{1, 2, \dots\}$  for  $i \in \{1, \dots, n\}$ , and  $x_i = 1 + x_j$  if  $j < i$  and  $C_j = C_i$  and  $\nexists k \in \{j + 1, \dots, i - 1\}$  such that  $C_k = C_i$ . Otherwise,  $x_i = 1$ .

**Definition. Completeness Set:**

The completeness set of a class  $C$ , written  $CS(C)$ , is defined as:

$\{lit.Par(trdl) \mid trdl \text{ is a transformer or dual of } C \text{ and } lit \text{ is a literal operation of } C\} \cup$   
 $\{lit..Par(accdl) \mid accdl \text{ is an accessor or dual of } C \text{ and } lit \text{ is a literal operation of } C\} \cup$   
 $\{Par(str).Par(trdl) \mid trdl \text{ is a transformer or dual of } C \text{ and } str \text{ is a structure operation of } C\} \cup$   
 $\{Par(str)..Par(accdl) \mid accdl \text{ is an accessor or dual of } C \text{ and } str \text{ is a structure operation of } C\}$

We first consider the completeness analysis of classes which are

not defined explicitly as subclasses or superclasses of already existing classes. In a class  $C$  which is not defined using the explicit class relationships we require that:

- Given  $trans$ , a transformer operation of  $C$ , either:
  - a)  $trans$  is defined by a preconditioned class equation
  - b)  $trans$  is partly defined by preconditioned structure equations on a set of structure operations,  $PS$  say, and  $\forall lit \in$  the set of literal operations of  $C$ ,  $lit.Par(trans) \in CS(C)$  and  $\forall st \notin PS$ , where  $st$  is a structure of  $C$ ,  $Par(str).Par(trans) \in CS(C)$ .
- Given  $acc$ , an accessor operation of  $C$ , either:
  - a)  $acc$  is defined by a preconditioned class equation
  - b)  $acc$  is partly defined by preconditioned structure equations on a set of structure operations,  $PS$  say, and  $\forall lit \in$  the set of literal operations of  $C$ ,  $lit..Par(acc) \in CS(C)$  and  $\forall st \notin PS$ , where  $st$  is a structure of  $C$ ,  $Par(str)..Par(acc) \in CS(C)$ .
- Given  $dl$ , a dual operation of  $C$ , either:
  - a)  $dl$  is defined by a preconditioned class equation
  - b)  $dl$  is partly defined by preconditioned structure equations on a set of structure operations,  $PS$  say, and  $\forall lit \in$  the set of literal operations of  $C$ ,  $lit.Par(dl) \in CS(C)$  and  $lit..Par(dl) \in CS(C)$  and  $\forall st \notin PS$ , where  $st$  is a structure of  $C$ ,  $Par(str).Par(dl) \in CS(C)$  and  $Par(str)..Par(dl) \in CS(C)$ .
- The expressions on the left hand sides of equation definitions in  $C$  do not have any repeated entries. In other words, each equation must be uniquely defined.

Completeness analysis for classes defined using the explicit classification mechanisms is based on the analysis above. In an object based specification, the explicit classification mechanisms define only syntactic sugarings of the inclusion mechanisms. Completeness checks are not concerned with subclassing properties in the OO ACT ONE specification. Consequently, to check the completeness of a class defined using an explicit classification mechanism we generate an intermediate class which exhibits the object based behaviour of the original class but does not include the explicit subclassing mechanism. (The means of generating such a class is similar to the mechanism for removing inclusion syntactic sugar.) This intermediate class is then tested for completeness<sup>1</sup> (as above). It plays no further role after completeness checks terminate.

---

<sup>1</sup>We accept that more efficient completeness checks can be formulated for classes defined explicitly to exhibit a class relationship with a class which has already been tested for completeness.

## Appendix C

# Mapping OO ACT ONE to ACT ONE

### C.1 Object Based Requirements

#### I: Classes and Sorts

Every class in an ACT ONE specification is translated into an ACT ONE sort. Each ACT ONE sort is defined inside a type bearing its name. In other words, in the generated ACT ONE code, types are used only as containers for single sorts. Dependencies between classes are mapped into dependencies between the types containing the corresponding sorts. For example,

```
Class USES Class1, ..., Classn
```

is translated into

```
TYPE Class IS Class1, ...Classn SORTS Class OPNS ....
```

The types in the ACT ONE specification are necessary for the modelling of object based dependencies between classes, since in ACT ONE it is not possible to explicitly define dependencies between sorts.

#### II: Operations

There is a direct correspondence between the operations of an OO ACT ONE class and the operations in the generated ACT ONE code.

- All OO ACT ONE LITERALS map to ACT ONE **literal** values. For example, if `lit` is defined as a literal of class `C` then, in the definition of TYPE `C`, there is an operation defined as `lit: -> C`.
- STRUCTURES in an OO ACT ONE class `C`, written `st<c1,...,cn>`, map to ACT ONE operations `st: c1,...,cn -> C`.
- TRANSFORMERS in class `C` map to ACT ONE operations in two different ways:
  - An unparameterised transformer of `C`, `tr` say, maps to an operation `tr:C -> C`.
  - A parameterised transformer of `C`, `tr<C1,...,Cn>` say, maps to the operation `tr:C, C1,...,Cn -> C`.
- ACCESSORS in class `C` also maps to ACT ONE operations in two different ways:

- An unparameterised accessor of  $C$ ,  $\text{acc} \rightarrow C'$  say, maps to an operation  
 $\text{acc}: C \rightarrow C$  (\* dual accessor  $C'$  \*).
- A parameterised accessor of  $C$ ,  $\text{acc}\langle C_1, \dots, C_n \rangle \rightarrow C'$  say, maps to the operation  
 $\text{acc}: C, C_1, \dots, C_n \rightarrow C$  (\* dual accessor  $C'$  \*).
- DUALS in class  $C$  map to ACT ONE operations as follows:
  - An unparameterised dual of  $C$ ,  $\text{dl} \rightarrow C'$  say, maps to an operation  $\text{dl}: C \rightarrow C$  (\* dual  $C'$  \*).
  - A parameterised accessor of  $C$ ,  $\text{dl}\langle C_1, \dots, C_n \rangle \rightarrow C'$  say, maps to the operation  
 $\text{dl}: C, C_1, \dots, C_n \rightarrow C$  (\* dual  $C'$  \*).

### III: Hidden Operations

Internal (hidden) operations are mapped as above except that the hidden operations are commented as such in the ACT ONE code. The static analysis of the OO ACT ONE from which the ACT ONE was developed guarantees that hidden operations are used only in the specification of internal behaviour.

### IV: Equations

Consider the mapping of total equations, literal equations, unpreconditioned structure equations, preconditioned structure equations and preconditioned class equations.

#### • 1) Total Equations

The translation of a total equation from a class definition ( $C$  say) to a sort definition of the same name (with a **result type**  $D$  where appropriate), is given below:

- $C_1.\text{tr} = \text{sle} \rightarrow \text{tr}(C_1) = \text{sle};$
- $C_1.\text{tr}(p_1, \dots, p_n) = \text{sle} \rightarrow \text{tr}(C_1, p_1, \dots, p_n) = \text{sle};$
- $C_1..\text{acc} = \text{sle} \rightarrow \text{tr}(C_1) = \text{dualCD}(C_1, \text{sle});$
- $C_1..\text{acc}(p_1, \dots, p_n) = \text{sle} \rightarrow \text{tr}(C_1, p_1, \dots, p_n) = \text{dualCD}(C_1, \text{sle});$
- $C_1.\text{dl} = \text{sle}_1 \text{ AND } \text{sle}_2 \rightarrow \text{dl}(C_1) = \text{dualCD}(\text{sle}_1, \text{sle}_2);$
- $C_1.\text{dl}(p_1, \dots, p_n) = \text{sle}_1 \text{ AND } \text{sle}_2 \rightarrow$   
 $\text{dl}(C_1, p_1, \dots, p_n) = \text{dualCD}(\text{sle}_1, \text{sle}_2);$

Note that the equations generated from the translation are defined in terms of variable parameters. It is a simple, though vital, part of the translation of this and all other equation types to define these variables in the `forall` clause at the beginning of the ACT ONE equation definitions for each sort. We have shown the mappings for all six forms of total equations. The mappings are very similar and so, for conciseness, we consider only a subset of the equation forms in each of the remaining equation type translations.

#### • 2) Literal Equations

The translation of a parameterised dual equation,  $\text{dl}$ , defined on a literal,  $\text{lit}$ , in a class,  $C$  with **result type**  $D$  is defined below:

$\text{lit}.\text{dl}(p_1, \dots, p_n) = \text{sle}_1 \text{ AND } \text{sle}_2 \rightarrow$   
 $\text{tr}(\text{lit}, p_1, \dots, p_n) = \text{dualCD}(\text{sle}_1, \text{sle}_2);$

The other forms are similarly defined.

- **3) Unpreconditioned Structure Equations**

The translation of an unpreconditioned unparameterised accessor equation `acc` defined on `str(p1,...,pn)` a structure of class `C` is as follows:

`str(p1,...,pn)..acc = sle → acc(str(p1,...,pn)) = sle;`

The other forms are similarly defined in an appropriate manner.

- **4) Preconditioned Structure Equations**

Consider the generic representation of a preconditioned structure equation:

$$\begin{aligned} pre_1 \Rightarrow sc(P_1, \dots, P_{j-1}).dl(P_j, \dots, P_n) = newstate_1 \text{ AND } result_1 \\ \text{OTHERWISE} \dots \text{OTHERWISE} \\ pre_{m-1} \Rightarrow newstate_{m-1} \text{ AND } result_{m-1} \text{ OTHERWISE} \\ newstate_m \text{ AND } result_m, \text{ for some } m, n, j \in \{1, 2, \dots\}, m \geq 2 \end{aligned}$$

This translates into the following set of ACT ONE preconditioned equations:

$$\begin{aligned} \{MAP(pre_1) \Rightarrow dl(sc(P_1, \dots, P_{j-1}), P_j, \dots, P_n) = dualCD(newstate_1, result_1)\} \cup \\ \{MAP(not(pre_1) \text{ and } \dots \text{ and } not(pre_{k-1}) \text{ and } (pre_k)) \Rightarrow \\ dl(sc(P_1, \dots, P_{j-1}), P_j, \dots, P_n) = dualCD(newstate_k, result_k) \mid k \in \{2, \dots, m-2\}\} \cup \\ \{MAP(not(pre_1) \text{ and } \dots \text{ and } not(pre_{m-1})) \Rightarrow \\ dl(sc(P_1, \dots, P_{j-1}), P_j, \dots, P_n) = dualCD(newstate_m, result_m)\} \end{aligned}$$

The meta-operation `MAP` defined on the boolean preconditions represents the mapping of the OO ACT ONE **state label expressions** of type `Bool` to ACT ONE expressions of sort `Bool`.

- **5) Preconditioned Class Equations**

Consider the following unparameterised transformer equation defined in class `C` (expressed in generic form):

$$\begin{aligned} pre_1 \Rightarrow C1.tr = newstate_1 \text{ AND } result_1 \text{ OTHERWISE} \dots \text{OTHERWISE} \\ pre_{m-1} \Rightarrow newstate_{m-1} \text{ AND } result_{m-1} \text{ OTHERWISE } newstate_m \text{ AND } result_m, \text{ for} \\ \text{some } m \in \{2, \dots\} \end{aligned}$$

This translates into the following set of ACT ONE preconditioned equations:

$$\begin{aligned} \{MAP(pre_1) \Rightarrow tr(C1) = newstate_1\} \cup \\ \{MAP(not(pre_1) \text{ and } \dots \text{ and } not(pre_{k-1}) \text{ and } (pre_k)) \Rightarrow tr(C1) = newstate_k \mid k \in \\ \{2, \dots, m-2\}\} \cup \\ \{MAP(not(pre_1) \text{ and } \dots \text{ and } not(pre_{m-1})) \Rightarrow tr(C1) = newstate_m\} \end{aligned}$$

**V: Structure Invariants** Structure invariants generate ACT ONE preconditions which precede every equation defining the behaviour corresponding to the appropriate structure. Consequently, operations on structured objects are defined only when the components of the objects fulfil the precondition property specified by the invariant which generated it.

Additional work is required to map invariant properties in combination with preconditioned equation definitions. Structured preconditions from OO ACT ONE must be coded in ACT ONE as the

boolean conjunction of the corresponding ACT ONE preconditions and the preconditions generated by any invariant properties. Class preconditions pose an even bigger problem than structured preconditions. Static analysis of the ACT ONE code flags every case in which these two mechanisms ‘overlap’. The generation of ACT ONE must then include an internal operation which tests an object to see if it is represented as a particular structure expression. All class preconditions are then separated into sets of precondition equations (one for each structure invariant, and one for the remaining objects). Appendix C2, following, illustrates the mapping of preconditions in the `Queue` class.

## C.2 Example Queue Behaviour

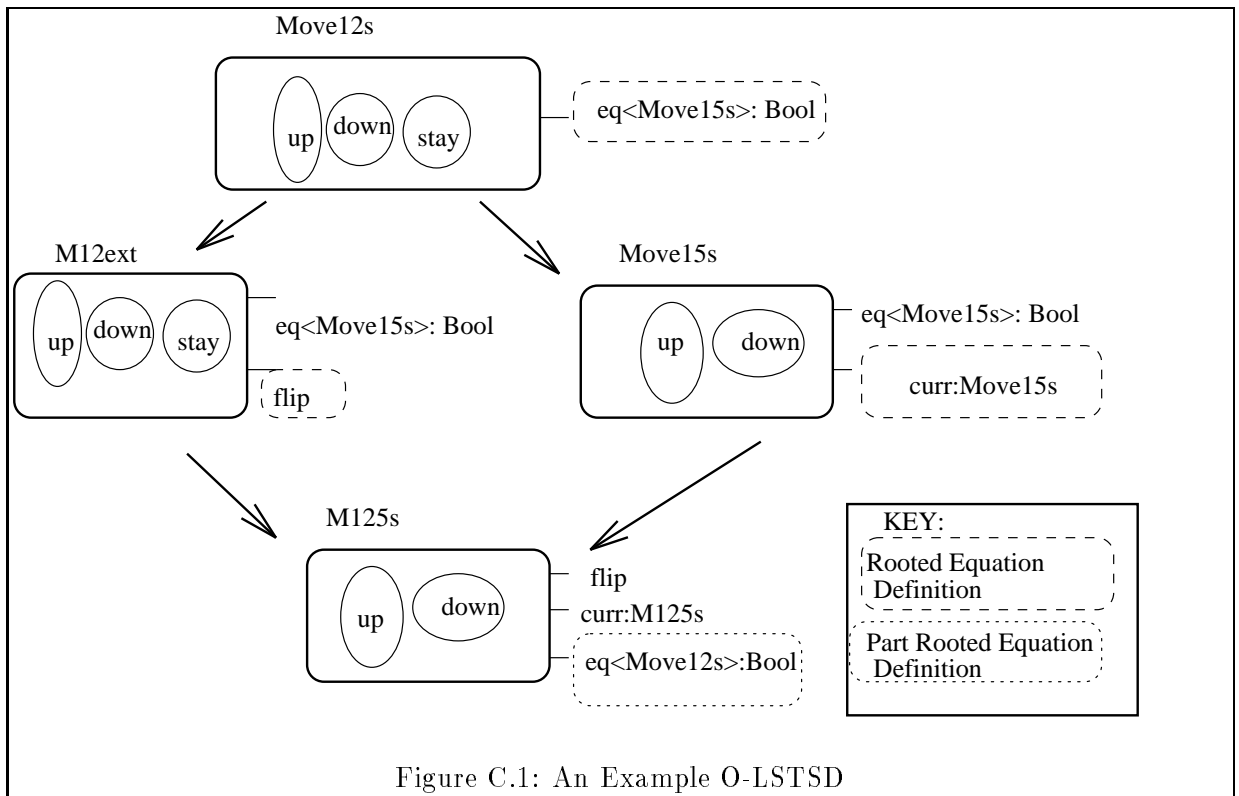
```

TYPE Queue IS Nat, Boolean
SORTS Queue OPNS
empty: -> Queue (* Literal *)
Q: Queue, Nat-> Queue (* Structure *)
add: Queue, Nat -> Queue (* Transformer *)
is-empty: Queue -> Queue (* Dual accessor Bool HIDDEN *)
rem: Queue -> Queue (* Dual Nat *)
unspecQueue: -> Queue
.: Queue -> Queue
NatResult: Queue -> Nat
BoolResult: Queue -> Bool
dualQueueNat: Queue, Nat -> Queue
dualQueueBool: Queue, Bool -> Queue
QueueRep: Queue -> Bool
EQNS FORALL Queue1: Queue, Nat1, Nat2: Nat, Bool1: Bool
OFSORT Queue
add(Queue1, Nat1) = Q(Queue1, Nat1);
add(unspecQueue, Nat1) = unspecQueue;
add(dualQueueNat(Queue1, Nat1), Nat2) = add(Queue1, Nat2);
add(dualQueueBool(Queue1, Nat1), Bool1) = add(Queue1, Nat2);
is-empty(empty) = dualQueueBool(empty, true);
is-empty(Q(Queue1, Nat1)) = dualQueueBool(Q(Queue1, Nat1), false);
rem(empty) = dualQueueNat(empty, unspecNat);
BoolResult(is-empty(Queue1)) =>
rem(Q(Queue1, Nat1)) = dualQueueNat(empty, Nat1);
not(BoolResult(is-empty(Queue1))) =>
rem(Q(Queue1, Nat1)) = dualQueueNat(Q.(rem(Queue1)), Nat1), BoolResult(rem(Queue1)));
QueueRep(Queue1) => .(Queue1) = Queue1;
.(dualQueueNat(Queue1, Nat1)) = Queue1;
OFSORT Bool
QueueRep(empty) = true; QueueRep(Q(Queue1, Nat1)) = true;
QueueRep(unspecQueue) = true; QueueRep(dualQueueNat(Queue1, Nat1)) = false
BoolResult(dualQueueBool(Queue1, Bool1)) = Bool1;
OFSORT Nat
NatResult(dualQueueNat(Queue1, Nat1)) = Nat1;
ENDTYPE (* Queue *)

```

### C.3 Translating Object Oriented Requirements: An Example

The mapping of object oriented properties to ACT ONE is best illustrated by the following example. The diagram in figure C.1 shows the hierarchy of behaviour which we wish to model in ACT ONE.



This class hierarchy illustrates two interesting features of object oriented specifications:

- **M125s** has got two direct superclasses (parents). It inherits the `flip` behaviour from **M12ext**, the `curr` behaviour from **Move15s** and the `eq` behaviour partly from **Move12s** (through either of its two parents).
- **M125s** illustrates the rules of contravariance and covariance between subclasses and superclasses. It is defined to return an **M125s** result in response to a `curr` request whilst its superclass **Move15s** is defined to return a superclass of that result class, namely **Move15s**. Furthermore, **M125s** can accept parameter values which are superclasses of the parameter values its superclasses can accept. For example, **M125s** can respond to the request `eq(stay)` but this service is not offered by **Move15s**.

## ACT ONE Requirements Model of Class Move12sRoot

SPECIFICATION Move12sRoot:noexit

LIBRARY

Boolean

ENDLIB

TYPE Move12sRoot is Boolean

SORTS Move12s (\* using Bool \*), Move15s (\* superclass Move12s\*),  
 M12ext (\* superclass Move12s \*),  
 M125s (\* superclass M12ext, Move15s \*)

OPNS

```
(* M125s ----- *)
  up,down: -> M125s                (* literals *)
  flip: M125s -> M125s             (* transformer from M12ext *)
  eq: M125s, Move12s -> M125s      (* dual Bool part from M12ext *)
      eq: M125s, Move15s -> M125s   (* parameter subclass *)
      eq: M125s, M12ext -> M125s   (* parameter subclass *)
      eq: M125s, M125s -> M125s    (* parameter subclass *)
      M125seq: M125s, Move12s -> M125s (* eq root definition part *)
  curr: M125s -> M125s             (* dual M125s from Move15s *)
  unspecM125s: -> M125s           (* Unspecified Machinery *)
(* Dual Machinery *)
  .: M125s -> M125s
  M125sResult: M125s -> M125s
  BoolResult: M125s -> Bool
  dualM125sM125s: M125s, M125s -> M125s
  dualM125sBool: M125s, Bool -> M125s
(* Subclass machinery *)
  M125stoM12ext: M125s -> M12ext
  M12exttoM125s: M12ext -> M125s
  M125stoMove15s: M125s -> Move15s
  Move15stoM125s: Move15s -> M125s
  M125stoMove12s: M125s -> Move12s
  Move12stoM125s: Move12s -> M125s
(* Internal Test *)
  M125sRep: M125s -> Bool
(* Move15s ----- *)
  up,down: -> Move15s              (* literals *)
  eq: Move15s, Move15s -> Move15s  (* dual Bool from Move12s *)
```

```

        eq: Move15s, M125s -> Move15s          (* parameter subclass *)
    curr: Move15s -> Move15s                    (* dual Move15s *)
        Move15scurr: Move15s -> Move15s        (* curr root definition *)
    unspecMove15s: -> Move15s                  (* Unspecified Machinery *)
(* Dual Machinery *)
    .: Move15s -> Move15s
    BoolResult: Move15s -> Bool
    Move15sResult: Move15s -> Move15s
    dualMove15sBool: Move15s, Bool -> Move15s
    dualMove15sMove15s: Move15s, Move15s-> Move15s
(* Subclass Machinery *)
    Move15stoMove12s: Move15s -> Move12s
    Move12stoMove15s: Move12s -> Move15s
(* Internal Test *)
    Move15sRep: Move15s -> Bool
(* M12ext ----- *)
    up, down, stay: -> M12ext          (* literals *)
    flip: M12ext -> M12ext             (* transformer *)
    eq: M12ext, Move15s -> M12ext      (* dual accessor Bool from Move12s *)
        eq: M12ext, M125s -> M12ext    (* parameter subclass *)
    unspecM12ext: -> M12ext           (* Unspecified Machinery *)
(* Dual Machinery *)
    .: M12ext -> M12ext
    BoolResult: M12ext -> Bool
    dualM12extBool: M12ext, Bool -> M12ext
(* Subclass Machinery *)
    M12extttoMove12s: M12ext -> Move12s
    Move12stoM12ext: Move12s -> M12ext
(* Internal Test *)
    M12extRep: M12ext -> Bool
(* Move12s ----- *)
    up, down, stay: -> Move12s          (* literals *)
    eq: Move12s, Move15s -> Move12s      (* dual accessor Bool *)
        eq: Move12s, M125s -> Move12s    (* parameter subclass *)
        Move12seq: Move12s, Move15s -> Move12s (* eq definition root *)
    unspecMove12s: -> Move12s          (* Unspecified Machinery *)
(* Dual Machinery *)
    .: Move12s -> Move12s
    BoolResult: Move12s -> Bool
    dualMove12sBool: Move12s, Bool -> Move12s

```

```

(* Internal Test *)
  Move12sRep: Move12s -> Bool

(* Additional 0-LSTS Machinery for Booleans *)
  unspecBool: -> Bool

EQNS FORALL Move15s1, Move15s2: Move15s, Move12s1: Move12s,
  M12ext1: M12ext, M125s1, M125s2, M125s3: M125s, Bool1: Bool
(* M125s ----- *)
(* Inherited from M12ext *)
OFSORT M125s
M125sRep(M125s1) =>
  flip(M125s1) = M12exttoM125s(flip(M125stoM12ext(M125s1)));
  flip(dualM125sBool(M125s1, Bool1)) = flip(M125s1);
  flip(dualM125sM125s(M125s1, M125s2)) = flip(M125s1);
(* Part inherited from M12ext --- contravariance on parameter 1 *)
M125sRep(M125s1) =>
  eq(M125s1, Move12s1) = M125seq(M125s1, Move12s1);
  M125seq(dualM125sM125s(M125s1, M125s2), Move12s1) = M125seq(M125s1, Move12s1);
  M125seq(dualM125sBool(M125s1, Bool1), Move12s1) = M125seq(M125s1, Move12s1);
  M125seq(up, stay) = dualM125sBool(up, false);
  M125seq(down, stay) = dualM125sBool(down, false);
  M125seq(unspecM125s, stay) = unspecM125s;
  Move15sRep(Move12stoMove15s(Move12s1)) =>
  M125seq(M125s1, Move12s1) =
    eq(M125s1, Move12stoMove15s(Move12s1));
  M125sRep(M125s1) =>
  eq(M125s1, Move15s1) =
    dualM125sBool(
      M12exttoM125s(. (eq(M125stoM12ext(M125s1), Move15s1))),
      BoolResult(eq(M125stoM12ext(M125s1), Move15s1)) );
  eq(M125s1, Move15s1) = eq(M125s1, Move15stoMove12s(Move15s1));
  eq(M125s1, M12ext1) = eq(M125s1, M12exttoMove12s(M12ext1));
  eq(M125s1, M125s2) = eq(M125s1, M125stoMove12s(M125s2));
(* Inherited from Move15s *)
OFSORT M125s
M125sRep(M125s1) =>
  curr(M125s1) =
    dualM125sM125s(
      Move15stoM125s(. (curr(M125stoMove15s(M125s1)))),

```

```

    Move15stoM125s(Move15sResult(curr(M125stoMove15s(M125s1)))) );
    curr(dualM125sBool(M125s1, Bool1))= curr(M125s1);
    curr(dualM125sM125s(M125s1, M125s2))= curr(M125s1);
(* Dual machinery *)
OFSORT M125s
M125sRep(M125s1) =>
    .(M125s1) = M125s1;
    .(dualM125sBool(M125s1, Bool1)) = M125s1;
    .(dualM125sM125s(M125s1, M125s2)) = M125s1;
M125sRep(M125s1) =>
    M125sResult(M125s1) = unspecM125s;
    M125sResult(dualM125sBool(M125s1, Bool1)) = unspecM125s;
    M125sResult(dualM125sM125s(M125s1, M125s2)) = M125s2;
OFSORT Bool
M125sRep(M125s1) =>
    BoolResult(M125s1) = unspecBool;
    BoolResult(dualM125sBool(M125s1, Bool1)) = Bool1;
    BoolResult(dualM125sM125s(M125s1, M125s2)) = unspecBool;
(* Subclass machinery *)
OFSORT M125s
Move12stoM125s(up) = up; Move12stoM125s(down) = down;
Move12stoM125s(unspecMove12s) = unspecM125s;
M12exttoM125s(up) = up; M12exttoM125s(down) = down;
M12exttoM125s(unspecM12ext) = unspecM125s;
Move15stoM125s(up) = up; Move15stoM125s(down) = down;
Move15stoM125s(unspecMove15s) = unspecM125s;
OFSORT Move15s
M125stoMove15s(up) = up; M125stoMove15s(down) = down;
M125stoMove15s(unspecM125s) = unspecMove15s;
OFSORT M12ext
M125stoM12ext(up) = up; M125stoM12ext(down) = down;
M125stoM12ext(unspecM125s) = unspecM12ext;
OFSORT Move12s
M125stoMove12s(up) = up; M125stoMove12s(down) = down;
M125stoMove12s(unspecM125s) = unspecMove12s;
(* Internal Test *)
OFSORT Bool
M125sRep(up) = true; M125sRep(down) = true;
M125sRep(unspecM125s) = true;
M125sRep(dualM125sBool(M125s1, Bool1)) = false;

```

```

M125sRep(dualM125sM125s(M125s1, M125s2)) = false;
(* Move 15s ----- *)
(* Root definitions *)
OFSORT Move15s
curr(Move15s1) = Move15sCurr(Move15s1);
Move15sCurr(up) = dualMove15sMove15s(up,up);
Move15sCurr(down) = dualMove15sMove15s(down,down);
Move15sCurr(unspecMove15s) = unspecMove15s;
Move15sCurr(dualMove15sBool(Move15s1, Bool1)) = Move15sCurr(Move15s1);
Move15sCurr(dualMove15sMove15s(Move15s1, Move15s2)) =
  Move15sCurr(Move15s1);
(* Inherited from Move12s *)
OFSORT Move15s
Move15sRep(Move15s1) =>
  eq(Move15s1, Move15s2) =
    dualMove15sBool(
      Move12stoMove15s(. (eq(Move15stoMove12s(Move15s1), Move15s2))),
      BoolResult(eq(Move15stoMove12s(Move15s1), Move15s2)) );
    eq(dualMove15sBool(Move15s1, Bool1), Move15s2) =
      eq(Move15s1, Move15s2);
    eq(dualMove15sMove15s(Move15s1, Move15s2), Move15s2) =
      eq(Move15s1, Move15s2);
    eq(Move15s1, M125s1) = eq(Move15s1, M125stoMove15s(M125s1));
(* Dual machinery *)
OFSORT Move15s
Move15sRep(Move15s1) =>
  .(Move15s1) = Move15s1;
  .(dualMove15sBool(Move15s1, Bool1)) = Move15s1;
  .(dualMove15sMove15s(Move15s1, Move15s2)) = Move15s1;
Move15sRep(Move15s1) =>
  Move15sResult(Move15s1) = unspecMove15s;
  Move15sResult(dualMove15sMove15s(Move15s1, Move15s2)) = Move15s2;
  Move15sResult(dualMove15sBool(Move15s1, Bool1)) = unspecMove15s;
OFSORT Bool
Move15sRep(Move15s1) =>
  BoolResult(Move15s1) = unspecBool;
  BoolResult(dualMove15sMove15s(Move15s1, Move15s2)) = unspecBool;
  BoolResult(dualMove15sBool(Move15s1, Bool1)) = Bool1;
(* Subclass machinery *)
OFSORT Move15s

```

```

    Move12stoMove15s(up) = up; Move12stoMove15s(down) = down;
    Move12stoMove15s(unspecMove12s) = unspecMove15s;
OFSORT Move12s
    Move15stoMove12s(up) = up; Move15stoMove12s(down) = down;
    Move15stoMove12s(unspecMove15s) = unspecMove12s;
(* Internal Test *)
OFSORT Bool
    Move15sRep(up) = true; Move15sRep(down) = true;
    Move15sRep(unspecMove15s) = true;
    Move15sRep(dualMove15sBool(Move15s1, Bool1)) = false;
(* M12ext ----- *)
(* Root definition *)
OFSORT M12ext
    flip(unspecM12ext) = unspecM12ext;
    flip(dualM12extBool(M12ext1, Bool1)) = flip(M12ext1);
    flip(up) = down; flip(down) = up; flip(stay) = stay;
(* Inherited from Move12s *)
OFSORT M12ext
M12extRep(M12ext1)=>
    eq(M12ext1, Move15s1) =
        dualM12extBool(
            Move12stoM12ext(. (eq(M12exttoMove12s(M12ext1), Move15s1))),
            BoolResult(eq(M12exttoMove12s(M12ext1), Move15s1)) );
    eq(dualM12extBool(M12ext1, Bool1), Move15s1) =
        eq(M12ext1, Move15s1);
    eq(M12ext1, M125s1) = eq(M12ext1, M125stoMove15s(M125s1));
(* Dual machinery *)
OFSORT M12ext
M12extRep(M12ext1)=>
    .(M12ext1) = M12ext1;
    .(dualM12extBool(M12ext1, Bool1)) = M12ext1;
OFSORT Bool
M12extRep(M12ext1)=>
    BoolResult(M12ext1) = unspecBool;
    BoolResult(dualM12extBool(M12ext1, Bool1)) = Bool1;
(* Subclass machinery *)
OFSORT M12ext
    Move12stoM12ext(up) = up; Move12stoM12ext(down) = down;
    Move12stoM12ext(stay) = stay;
    Move12stoM12ext(unspecMove12s) = unspecM12ext;

```

```

OFSORT Move12s
  M12extttoMove12s(up) = up;  M12extttoMove12s(down) = down;
  M12extttoMove12s(stay) = stay;
  M12extttoMove12s(unspecM12ext) = unspecMove12s;
(* Internal Test *)
OFSORT Bool
  M12extRep(up) = true; M12extRep(down) = true; m12extrep(stay) = true;
  M12extRep(unspecM12ext) = true;
  M12extRep(dualM12extBool(M12ext1, Bool1)) = false;
(* Move12s ----- *)
(* Root definitions *)
OFSORT Move12s
  eq(Move12s1, Move15s1) = Move12seq(Move12s1, Move15s1);
  eq(Move12s1, M125s1) = eq(Move12s1, M125stoMove15s(M125s1));
  Move12seq(up, up) = dualMove12sBool(up, true);
  Move12seq(up, down) = dualMove12sBool(up, false);
  Move12seq(down, up) = dualMove12sBool(down, false);
  Move12seq(down, down) = dualMove12sBool(down, true);
  Move12seq(stay, up) = dualMove12sBool(stay, false);
  Move12seq(stay, down) = dualMove12sBool(stay, false);
  Move12seq(unspecMove12s, Move15s1) = unspecMove12s;
  Move12seq(Move12s1, unspecMove15s) = unspecMove12s;
  Move12seq(dualMove12sBool(Move12s1, Bool1), Move15s1) =
    Move12seq(Move12s1, Move15s1);
(* Dual machinery *)
OFSORT Move12s
  Move12sRep(Move12s1) =>
    .(Move12s1) = Move12s1;
    .(dualMove12sBool(Move12s1, Bool1)) = Move12s1;
OFSORT Bool
  Move12sRep(Move12s1) =>
    BoolResult(Move12s1) = unspecBool;
    BoolResult(dualMove12sBool(Move12s1, Bool1)) = Bool1;
(* Internal Test *)
OFSORT Bool
  Move12sRep(up) = true; Move12sRep(down) = true; Move12sRep(stay) = true;
  Move12sRep(unspecMove12s) = true;
  Move12sRep(dualMove12sBool(Move12s1, Bool1)) = false;
ENDTYPE (* MovesRoot *)
(* EXAMPLE State-label expression evaluations*)

```

```

(*)
a = eq( up of Move12s, up of Move15s);
b = eq( a, down of Move15s);
c = eq(down of Move12s, down of M125s);
d = eq(c, up of M125s);
e = flip(stay);
f = eq(e, down of Move15s);
g = eq(up of M12ext, up of M125s);
h = flip(g);
i = eq(up of Move15s, up of Move15s);
j = eq(i, .(i));
k = curr(j);
l = eq(j, up of M125s);
m = flip(up of M125s);
n = eq(m, .(m));
o = eq(flip(m), .(m));
p = curr(n);
p = curr(n);
q = curr(o);
-----
a = dualMove12sBool(up, true);
c = dualMove12sBool(down, true);
e = stay;
g = dualM12extBool(up, true);
i = dualMove15sBool(up, true);
m = down;
b = dualMove12sBool(up, false);
d = dualMove12sBool(down, false);
f = dualM12extBool(stay, false);
h = down;
j = dualMove15sBool(up, true);
n = dualM125sBool(down, true);
o = dualM125sBool(up, false);
k = dualMove15sMove15s(up, up);
l = dualMove15sBool(up, true);
p = dualM125sM125s(down, down);
q = dualM125sM125s(up, up)
*)

```

## Appendix D

# An OO ACT ONE Interpretation of Interaction

### D.1 Interaction

Objects which are configured are able to interact (in some as yet unspecified way) for their separate behaviours to be combined in the fulfilment of a service request in their containing object. There are two different types of interaction:

- **Master-Slave Relationships.**

These are modelled, in OO ACT ONE, when a containing object requests its component objects to fulfil services. It is this interaction which is given a formal definition in the O-LSTS semantics in terms of **state label expression** evaluations.

- **Peer-Peer Relationships.**

When two components of the same containing object interact such that each can request services of the other they are called peer objects. An OO ACT ONE specification can be implemented to exhibit this type of relationship at the code level, but this is not specified in the requirements model.

A master-slave relationship implies a control flow from master to slave in all interactions. Peer-to-peer interactions imply that control flow can occur in either direction. Control flow is a dynamic property of an object oriented system which is an important aspect of design and implementation but does not have a fundamental role in analysis. The same can be said of data flow. Both these terms are widespread in structured methods but are not central to object oriented analysis.

## D.2 Data and Control Flow

Data and control flow, which on the surface seem quite different, are very difficult to distinguish without a formal semantics. One of the main difficulties in applying structured analysis techniques is in distinguishing the two concepts, even though they are modelled in different ways. Data flow diagrams and control flow models are prolific in structured analysis methods but are not explicit in our object oriented model. To understand the reason for this it is necessary to consider some examples.

### Data and Control Flow Example: A Two Stack System

Consider the OO ACT ONE `System` specification given below.

```

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

```

Specifications with static structure (like `System`) are amenable to three structural interpretations:

- 1) **Configuration**

This specification can be interpreted as saying that the two `Stack` components are configured by the `move` attribute. Chapter 4, section 3, formally defines configuration in terms of dependency and so it is not necessary to consider it in any more detail as part of this example.

- 2) **Data Flow**

An accessor operation on a component object, `comp` say, written `comp..acc ...`, on the right hand side of an equation definition can be interpreted as modelling data flow from `comp` to the containing object. In other words, the `ACCESSOR` (or `DUAL`) service requests model data flow from client to server (in the form of the result returned by the service). A parameterised attribute on a component object `comp` can be interpreted as modelling data flow into `comp`, i.e. an input parameter. Now, if the data flow into one component matches the data flow out of another this can be interpreted as saying that data flows between the two peer components. More formally, a **state label expression** of the form

$$\text{obj1.att1}(\dots, \text{obj2.att2}(\dots), \dots) \text{ or } \text{obj1..att1}(\dots, \text{obj2.att2}(\dots), \dots)$$

can be interpreted as data flowing internally from `obj2` to `obj1`.

In the `System` specification a high level interpretation can lead to the statement that that data flows from `Stack1` to `Stack2` in response to a `move` request. Note that we do not say how the information is transferred.

- **3) Control Flow**

The simplest interpretation of control flow is from client to server (and back again) when the client requests some service from the server. In the O-LSTS model we do not formulate an interpretation of control flow between peer components. In a **System** class we may implement the first **Stack** to be subordinate to the second **Stack**: ‘the **move** request is passed on to the second component which requests a **pop** from the first **Stack**’. Contrastingly, the first **Stack** can be implemented to request a **push** operation of the second component. A third option is to have some additional controlling process (object) which mediates between the two **Stacks**. Such decisions are not the realm of analysis and as such OO ACT ONE does not provide an explicit mechanism for defining such properties. Designers and implementers are free to choose which ‘less-abstract’ interpretation of control flow they take from an OO ACT ONE specification.

# Appendix E

## Design Issues

### E.1 The ParXStack Process Definition

The Par Specification of the extended Stack behaviour (XStack) is defined as follows.

```
process ParXStack[push,pop,size](SStack: Stack): noexit:=
hide request, response in
StackIn[push, pop, size, request] (0) |[request]|
StackBody [request, response](SStack) |[response]|
StackOut [pop, size, response](0)
where
process StackIn[push, pop, size, request] (ID: Nat): noexit :=
Reqs[push,pop,size,request](ID) |[request]| ReqController[request](ID) where
process Reqs[push,pop,size,request](IDsStackIn:Nat): noexit:=
(push? Nat1:Nat;
 ( Reqs[push, pop, size, request] (.inc(IDsStackIn)))
 |||
 (request!push!Nat1!IDsStackIn; exit)))
[]
(pop;
 (Reqs[push,pop, size, request] (.inc(IDsStackIn)))
 |||
 (request!pop!IDsStackIn; exit)))
(size;
 (Reqs[push,pop, size, request] (.inc(IDsStackIn)))
 |||
 (request!size!IDsStackIn; exit)))
endproc (*Reqs*)
process ReqController[request](ServeID:Nat):noexit:=
(request!push?Nat1:Nat!ServeID; ReqController[request] (.inc(ServeID))))
```

```

[]
(request!pop!ServeID; ReqController[request](.(inc(ServeID))))
[]
(request!size!ServeID; ReqController[request](.(inc(ServeID))))
endproc (* ReqController *)
endproc (* StackIn *)
process StackBody[request, response](SStack: Stack): noexit:=
( request!push? Nat1: Nat?ID:Nat;
(StackBody[request, response](.(push(SStack, Nat1)))
|||
(response!push!ID; exit)))
[]
( request!pop?ID:Nat;
(StackBody[request, response](.(pop(SStack)))
|||
(response!pop!NatResult(pop(SStack))!ID; exit)))
( request!size?ID:Nat;
(StackBody[request, response](.(pop(SStack)))
|||
(response!size!NatResult(size(SStack))!ID; exit)))
endproc (* StackBody *)
process StackOut[pop, size, response](CountStackOut: Nat): noexit:=
(response!pop?NatStackOut:Nat!CountStackOut;
pop!NatStackOut; StackOut[pop, response](.(inc(CountStackOut))))
[]
(response!size?NatStackOut:Nat!CountStackOut;
size!NatStackOut; StackOut[pop, response](.(inc(CountStackOut))))
[]
(response!push!CountStackOut;
StackOut[pop, response](.(inc(CountStackOut))))
endproc (* StackOut *)
endproc (* ParStack *)

```

## E.2 Two Mappings from OO ACT ONE to an Initial Full LOTOS Design

Given an OO ACT ONE class, **CName** say, with operation definitions:

LITERALS:  $lit_1, \dots, lit_l$   
 Unhidden external TRANSFORMERS:  
 $tr_1 < \dots >, \dots, tr_n < Ptr_{n_1}, \dots, Ptr_{n_m} >$   
 Unhidden internal TRANSFORMERS:  
 $itr_1 < \dots >, \dots, itr_o < Pit_{o_1}, \dots, Pit_{o_p} >$   
 Unhidden ACCESSORS:  
 $acc_1 < \dots > - > AResult_1, \dots, acc_p < Pac_{p_1}, \dots, Pac_{p_q} > \rightarrow AResult_p$   
 Unhidden DUALS:  
 $dl_1 < \dots > \rightarrow DResult_1, \dots, dl_r < Pdl_{r_1}, \dots, Pdl_{r_s} > \rightarrow DResult_r$

we can define the result of applying *MakeRPC* and *MakePar* to **CName** (in E.2.1 and E.2.2, below).

First, some notation is useful:

- $Req_{CName} ? p_1 : P_1 ? \dots ? p_n : P_n$  represents a parameterised event, where *Req* is an unhidden attribute of the class **CName**, and  $P_1, \dots, P_n$  are the input parameter types of *Req*.
- $Req_{CName} ! p_1 ! \dots ! p_n$  represents an event, where *Req* is an unhidden attribute of the class **CName** and  $(p_1, \dots, p_n)$  are values of the appropriate sorts.
- $\square_{Req_{CName}}$  represents a parameterised choice of behaviours over the *Req* attributes of **CName**.
- $\square_{AD_{CName}}$  represents a parameterised choice over the accessor and dual *AD* attributes of **CName**.
- $\square_{Tr_{CName}}$  represents a parameterised choice over the transformer *Tr* attributes of **CName**.
- $Result_{AD_{CName}}$  is the ACT ONE sort corresponding to the result class of the *AD* accessor or dual attribute of **CName**.

### E.2.1 The MakePar Mapping

*MakePar*(**CName**) =

```

process ParCName[ $tr_1, \dots, tr_n, acc_1, \dots, acc_p, dl_1, \dots, dl_r$ ](SCName): noexit =
hide request, response,  $itr_1, \dots, itr_o$  in
  CNameIn[ $tr_1, \dots, tr_n, acc_1, \dots, acc_p, dl_1, \dots, dl_r, request, itr_1, \dots, itr_o$ ](0) | [request] |
  CNameBody[request, response](SCName) | [response] |
  CNameOut[ $acc_1, \dots, acc_p, dl_1, \dots, dl_r, response$ ](0)
where ...

```

Process **CNameIn** is defined as follows:

```

process CNameIN[ $tr_1, \dots, tr_n, acc_1, \dots, acc_p, dl_1, \dots, dl_r, itr_1, \dots, itr_o, request$ ](ID:Nat):
noexit:=
Reqs[ $tr_1, \dots, tr_n, acc_1, \dots, acc_p, dl_1, \dots, dl_r, itr_1, \dots, itr_o, request$ ](ID)
| [request] |
ReqControl[request](ID)
where
process Reqs[...](ID):noexit:=
 $\square_{ReqCName}$ 
( $Req?p_1 : P_1, ?, \dots, ?p_n : P_n; ( Reqs[...](. (inc(ID))) ||| request!Req!p_1! \dots !P_n!ID$ )
endproc (* Reqs *)
process ReqControl[request](ID):noexit :=
 $\square_{ReqCName}$ 
(request! $Req?p_1 : P_1, ?, \dots, ?p_n : P_n !ID; ReqControl[...](. (inc(ID)))$ )
endproc (* ReqControl *)

```

Process CNameOut is defined as follows:

```

process CNameOut[ $acc_1, \dots, acc_p, dl_1, \dots, dl_r, response$ ] (ID: Nat): noexit:=
 $\square_{TrCName}$ 
(response! $TrCName!ID; CNameOut[...](. (inc(ID)))$ )
 $\square_{ADCName}$ 
(response! $ADCName?Result: Result_{ADCName}!ID;$ 
 $ADCName! Result; CNameOut[...](. (inc(ID)))$ )
endproc (* CNameOut *)

```

Process CNameBody is defined as follows:

```

process CNameBody [ request, response ](SCName: CName): noexit:=
 $\square_{TrCName}$ 
(request! $TrCName?p_1 : P_1, \dots, p_n : P_n?ID:Nat;$ 
(CNameBody[...](. ( $TrCName$ (SCName,  $p_1, \dots, p_n$ ))))
||| (response! $TrCName!ID; exit$ ))
 $\square_{ADCName}$ 
(request! $ADCName?p_1 : P_1, \dots, p_n : P_n?ID:Nat;$ 
(CNameBody[...](. ( $ADCName$ (SCName,  $p_1, \dots, p_n$ ))))
||| (response! $ADCName! Result_{ADCName} Result( ADCName$ (SCName,  $p_1, \dots, p_n$ ))!ID; exit))
endproc (* CNameBody *)

```

## E.2.2 The MakeRPC Mapping

$MakeRPC(CName) =$

```

process RPCCName[ $tr_1, \dots, tr_n, acc_1, \dots, acc_p, dl_1, \dots, dl_r$ ](SCName): noexit :=
 $\square_{TrCName}$ 

```

```

(TrCName?p1 : P1, ..., pn : Pn?ID:Nat;
(RPCCName[...](. (TrCName(SCName, p1, ..., pn)))
))
[]AD_CName
(AD_CName?p1 : P1, ..., pn : Pn?ID:Nat;
(RPCCNameBody[...](. (AD_CName(SCName, p1, ..., pn)));
AD_CName!ResultAD_CNameResult( AD_CName(SCName, p1, ..., pn)!ID);
(RPCCName[...](. (TrCName(SCName, p1, ..., pn)))
endproc (* RPCCName *)

```