An Object Semantic Model of SOFL

Jin Song Dong*
School of Computing, National University of Singapore,
dongjs@comp.nus.edu.sg
Shaoying Liu†
Faculty of Information Sciences Hiroshima City University, Japan,
liu@white.sel.cs.hiroshima-cu.ac.jp

Abstract

SOFL (Structured-Object-based-Formal Language) is recently proposed to combine the advantages of formal methods, structured methods and object-oriented methodology into one method for software development. SOFL integrates modeling techniques from VDM (for specifying system component state) and data flow diagrams with Petri nets (for specifying system behaviours and interaction). To support the standardisation and the tool support for SOFL, a formal semantics of the language is desirable. As SOFL is under further development, the semantic representation of current SOFL needs to be easily reused and extended. In this paper, an object-oriented approach is taken to specify the semantics of SOFL and this approach is proved to be effective in providing a modular and reusable semantic representation. Object modeling techniques, such as inheritance is used to illustrate the common and the differences between various SOFL graphical constructs; object containment is used to capture the acyclic structure of SOFL hierarchical module decomposition.

1 Introduction

One reason for the slow take-up of formal methods in industry is that the existing formal methods are not well integrated with the established industrial software processes. SOFL combines the advantages of formal methods, structured methodology and object-oriented methodology into one method [18]. SOFL is a graphical and textual formal notation that integrates VDM (for modeling system component state), data-flow-diagrams and Petri nets (for modeling system interaction and behaviours). To support the language standardisation and the case tools development (e.g. type checker and reasoning tool development), a formal specification of the semantics of

*Sponsored by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM(A440).

†Supported by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No.90264960) and Hiroshima City University under Hiroshima City University Grant for Special Academic Research (No.9842).

SOFL is desirable. As SOFL is under further development (i.e. current work on real-time [17] and concurrency extensions), the semantic representation of SOFL needs to be easily reused and extended.

A denotational semantics approach is considered as the most effective way to define the semantics of a language. It defines the meaning of the language in terms of mathematical objects [23]. The formal specification language VDM is often used as a meta-language to describe the denotational semantics of programming languages [1] (a VDM model of a language is called a denotational style semantics of the language). The Z notation has also been applied to describe programming language semantics [15, 22] and UML class diagrams [12]. However, the semantic representations in VDM and Z are not readily reusable and extendible. We believe the main reason for this is that VDM or Z notation itself lack a suitable modular structure.

Object-oriented modeling techniques support modularity and reusability, it is appropriate to apply object-oriented extensions of VDM or Z, such as VDM++/Z++ or Object-Z, to specify language semantics. As object-orientation is becoming more acceptable, mature and popular, the object-oriented approach to language semantics is promising. This paper uses the Object-Z notation to give an object-oriented specification of the SOFL language semantics. The reasons for choosing the Object-Z notation as the meta-language to define the SOFL semantics are:

- The semantics of Object-Z itself is well studied. The denotational semantics [14] and axiomatic semantics [20] of Object-Z are closely related to the Z standard work [19, 24]). Object-Z also has a fully abstract semantics [9, 21].

- Object-Z constructs, such as class-union [4] and object containment [5], are particularly useful to define the polymorphic and recursive nature of language constructs. They have been effectively applied to programming language semantics [6]. Although SOFL is a graphical and textual specification notation which is different from programming languages, we hope parts of the programming language semantic model [6] can be reused in the SOFL semantic model.

With this object-oriented approach, the SOFL language constructs such as expressions, condition processes and data-flows are modeled as objects whose attributes and operations capture the role of the construct. A SOFL module is specified in terms of its underlying semantic structures (i.e. the objects associated with the language constructs) and their interaction. The novel part of this paper is that the power of object-orientation is applied to the semantics of a graphical and textual specification notation.

The remainder of the paper is organised as follows. Section 2 presents an overview of the SOFL specification language. Section 3 outlines the object-oriented approach for modeling SOFL. Section 4 presents the Object-Z model of SOFL. Section 5 concludes the paper.

Familiarity with the basic features of the Object-Z notations as given in [11] is assumed.

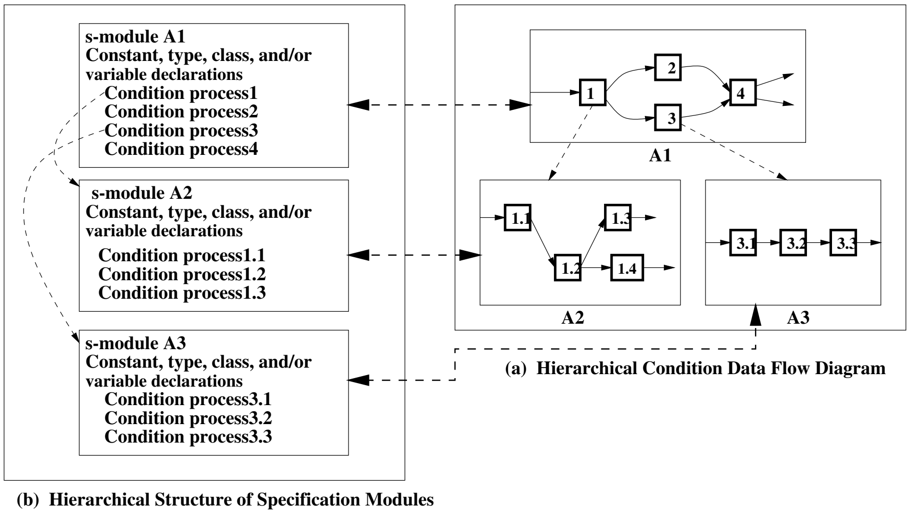

Figure 1: The structure of a SOFL specification

2 An Overview of SOFL

SOFL uses structured methods for requirements analysis and specifications and object-based approach for design and implementation. During both the structured and object-based development of a system, formal methods (a VDM like) can be applied to provide high quality specifications and verifications of various levels of the system. A SOFL specification is a hierarchical condition data flow diagram (DFD) that is linked with a hierarchy of specification modules, as shown in Figure 1. A *condition data flow diagram* (CDFD) is a directed graph consisting of data flows, data stores, and condition processes. A data flow is labeled with a typed variable that represents a packet of data transferred between condition processes. A data store is a variable of a specific type to represent data at rest. A condition process is like a process in Yourdon data flow diagrams, but with pre and postconditions that specify its functionality.

There are four important distinctions between CDFDs and Yourdon data flow diagrams:

1. A CDFD not only describes how components are statically connected, but also how they dynamically interact with data flows. For example, Figure 2(a) shows a classical DFD that describes a connection between the processes Hotel and Customer by data flows reservation and check-out-bill. This says nothing about the order in which the processes handle the data. The CDFD in Figure 2(b) shows the same system, and by the operational semantics the CDFD also indicates that a *firing* of condition process Hotel with an input reservation may either generate a check-out-bill or non-output (indicated by the output port without data flow), and a *firing* of condition process Customer with either input

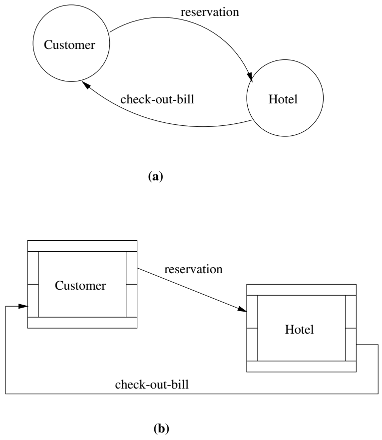

Figure 2: A comparison of CDFD and classical DFD

check-out-bill or non-input generates either non-output or a reservation. The precise definition of its functionality should be given in the associated specification module.

2. A classical DFD represents the static structure of the system and a given process appears only once in a strictly nested hierarchy, while a CDFD represents the dynamic structure of the system and an appearance of a process in the CDFD represents a use of the process. The same process may be used in different parts of a specification to process different data.

3. In classical DFDs, a data flow only transfers data from sources (e.g. process, external process, or data store) to destinations (similar). In CDFDs, a data flow indicates both data transfer and system control.

4. CDFDs may use *conditional nodes* (analogous to *diamond* in flowcharts) to express alternative data flows depending on the values in a data packet, and other diverging data flows, as shown in Figures 3. Classical DFDs have no such nodes.

For each CDFD in the hierarchy, we provide a specification module (s-module). The s-module and the CDFD are complementary in three respects: (1) the CDFD describes the relation between condition processes in terms of data flows while the module describes the precise functionality of the condition processes in terms of their inputs and outputs; (2) the CDFD describes the dynamic structure of the system while the module provides a static definition of all its components; and (3) the CDFD provides a graphical view of the system at the current level while the module supplies the details of the system in a textual form. Here is an example of a simple SOFL module including two condition processes (the purpose of this example is to illustrate syntax only):

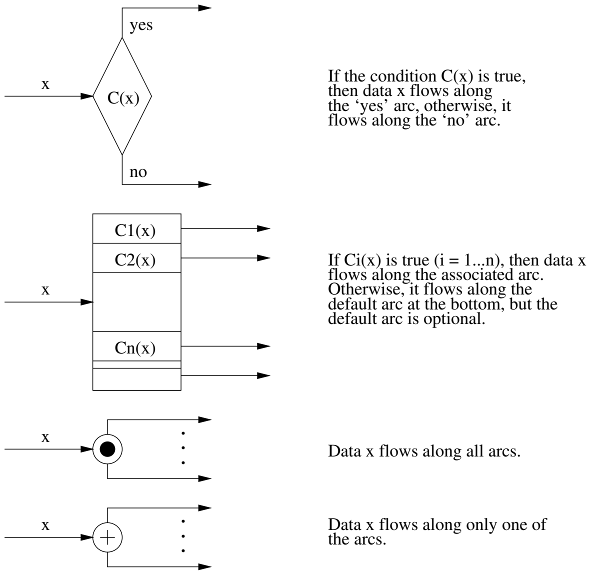

Figure 3: Conditional, Case, Broadcast and Choice data flows

s-module ChangingPoint
type Point = **composed of**

x: Int; y: Int

end
var mypoint: Point;
c-process Compare(point:Point)same: Boolean, thepoint: Point

ext rd mypoint: Point
pre true
post

same = (point.x = mypoint.x **and** point.y = mypoint.y)
and thepoint.x = point.x **and** thepoint.y = point.y

end-process;
c-process MayChange(is-same:boolean, pt:Point)

ext **wr** mypoint: Point
pre true
post

(is-same **implies** (mypoint.x = mypoint.x **and**
mypoint.y = mypoint.y)) **and** ((**not** is-same) **implies**
(mypoint.x = pt.x **and** mypoint.y = pt.y))

end-process
end-module

3 The Object-Oriented Approach to SOFL Semantics

The difference between object-oriented semantics approach and traditional semantics approach is that object-oriented approach maps target language constructs into high-level formal constructs, classes, while traditional approach maps target language constructs into primitive formal constructs, sets and functions. If an object-oriented approach is taken to model SOFL, then a SOFL specification module can be modeled as a composite object of an Object-Z class, say *Module*, which consists of a collection of objects, such as variables, condition processes, data-flow diagrams, etc. Given this, we can use different Object-Z classes to specify the different language constructs of SOFL. For example, if condition processes are modeled by an Object-Z class, say *CP*, then any individual condition process is modeled as an instance of the Object-Z class *CP*.

SOFL allows a condition process to be decomposed into a specification module. A specification module can contain a collection of condition processes. Two crucial properties of the relationship between condition processes and modules are:

p1 the decomposition chain is acyclic

p2 modules partition the condition processes

These properties impose a particular reference structure. Such a structure can be precisely captured in Object-Z by using containment notations as: (Note that *CP* and *Module* are examples in this section only.)

CP

$\ldots$
$decomp : Module_{Ⓛ}$

Module

$\ldots$
$cps : \mathbb{P}\, CP_{©}$

The properties p1 and p2 are implied by the semantics of the subscript `$_{Ⓛ}$' and `$_{©}$' (local objects and object containment [5]) appended to the type of the attribute *decomp* of *CP* and *cps* of *Module*. Local objects (sharably contained objects) ensures that no object directly or indirectly contains itself (a DAG structure); and containment further restricts that no object is directly contained in two distinct objects (a forest structure). For a discussion of local objects and object containment see [5] (note that in [5] local objects are called sharable contained objects.)

When modeling a notation construct, if it needs to be further classified into more specific constructs, then one can use different Object-Z classes to model these constructs. For example, suppose there are different kinds of data-flows, such as simple, condition, case, broadcast and choice data-flow (see Figure 3). Different Object-Z classes, say *SimpleDF*, *ConditionDF*, *CaseDF*, *BroadcastDF* and *ChoiceDF*, can be used to model these data-flow constructs. Any individual data-flow, say condition data-flow, in a CDFD can be modeled as an instance of the Object-Z class *ConditionDF* (and the same for modeling other data-flows). If an entity, say *df*, refers to a general data-flow (either a simple, a condition, a case, a broadcast or a choice), then *df* can be modeled as an element of a class-union [4] $SimpleDF \cup ConditionDF \cup$

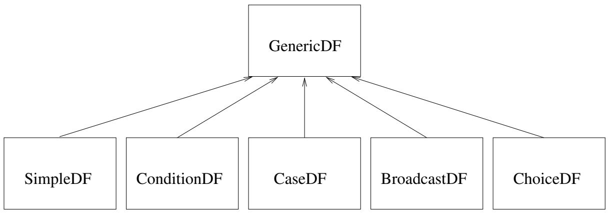

Figure 4: The inheritance structure of the data flow Model

CaseDF $\cup$ *BroadcaseDF* $\cup$ *ChoiceDF* (class-union is a generalised polymorphic type construct in Object-Z, for a discussion of class-union see [3]).

The common and difference of similar notation constructs can be more effectively captured by using inheritance techniques. For example, all data-flows have a similar structure which includes a (left-hand side) source and one or more (right-hand side) destinations and information is passed from source to destinations. In Object-Z, the common part of the data-flows can be modeled by a generic class, say *GenericDF*. Then the specific data-flow model, such as *SimpleDF*, *ConditionDF*, *CaseDF*, *BroadcastDF* and *ChoiceDF*, can be defined by inheriting *GenericDF* (see Figure 4). The difference between those data-flows can be clearly illustrated in their corresponding specific models. This approach is used throughout the formal model of SOFL in Section 4. We believe it is a good way to improve the readability of a semantic model as the common and the difference between the notation constructs are clearly separated. In fact, it becomes an object-oriented design pattern for specifying language semantics.

4 Formal Object Model of SOFL

The main SOFL notation constructs are condition process, condition data flow diagram and module. As the model of these constructs requires the understanding SOFL basic entities, such as types, variables, expressions etc, we will start from the basic constructs.

4.1 SOFL Types, Values and Variables

Let $[Id]$ denote all the possible identifiers (including module names, conditional process names, variable names, etc) and the function:

$$before : Id \rightarrowtail\!\!\!\!\!\rightarrow Id$$

relates the variable names to their pre-state variable names, e.g. $before(x) = \tilde{}x$.

Types

We can model all SOFL constructs as objects. For example, the variable types can be modeled as objects. The pre-defined-types are Integer, Boolean and Void. They are modeled as:

$$PreDefType \mathrel{\widehat=} IntType \cup BoolType \cup VoidType \qquad \text{where}$$

$IntType_{\textcircled{1}}$ $\qquad$ $BoolType_{\textcircled{1}}$ $\qquad$ $VoidType_{\textcircled{1}}$

The subscript '$_{\textcircled{1}}$' on *IntType* specifies that there is only one Integer type. It is a syntactic sugar for a system constraint: $\#IntType = 1$. Similarly, there is only one Boolean type and one Void type.

User defined types can be composite, maps, etc. It can be modeled as :

$$UserDefType \mathrel{\widehat=} CompType \cup \ldots$$

A composite type has a name and a field which contains a set of declarations. It is modeled as:

CompType

$$\begin{array}{l} name : Id \\ field : Id \pfun Type \end{array}$$

In contrast to the pre-defined types, the number of composite types (defined by user) is not restricted.

Other user-defined types can be similarly defined and are not included in this paper. The aim is to demonstrate the approach rather than providing the complete detailed formal description of SOFL which would be too lengthy for this paper. For the same reason, quantification expressions are also omitted. The complete formal model can be found in [7].

SOFL value types can be modeled as a class-union:

$$Type \mathrel{\widehat=} PreDefType \cup UserDefType$$

Values

SOFL variable values correspond to its types. They include Integer, Boolean and Composite values. SOFL follows a VDM style semantics, where variables can be bound to a value of their type, or they can be nil, which means *unavailable* or *non_value*

Let the Object-Z classes:

Nil

VoidVal

$$\begin{array}{l} \mathcal{V} : Value \\ \hline \mathcal{V} = self \end{array}$$

model a nil value and a void value respectively. This distinction between a nil value and a void value is important in SOFL condition process firing mechanism, as a variable that equates to nil means the variable have no value, while a variable that equates to a void value means it has a value but we don't care what it is. Like other defined values, the meaning of a void value is simply just the value itself. When an object of class *VoidVal* is instantiated, the identifier *self* will denote the identity of the object itself (see [10] for a detailed discussion).

The integer and boolean values are modeled as:

$$\begin{array}{l} \underline{\mathit{IntVal}} \\ \mathit{VoidVal} \\ \hline \mathit{val} : \mathbb{Z}_{\textcircled{1}} \\ \hline \end{array} \qquad \begin{array}{l} \underline{\mathit{BoolVal}} \\ \mathit{VoidVal} \\ \hline \mathit{val} : \mathbb{B}_{\textcircled{1}} \\ \hline \end{array}$$

Each instance of the class *IntVal* or *BoolVal* models a particular integer value or a boolean value. The one-to-one correspondence relationship is captured by the subscript `$_{\textcircled{1}}$' on the type of *val*, which is a syntactic sugar for a system constraint (e.g. for *IntVal*):

$$\forall\, i : \mathbb{Z} \bullet \exists_1\, v : \mathit{IntVal} \bullet v.\mathit{val} = i \;\wedge\; \forall\, v : \mathit{IntVal} \bullet \exists_1\, i : \mathbb{Z} \bullet v.\mathit{val} = i$$

A composite value can be modeled as:

$$\begin{array}{l} \underline{\mathit{CompVal}} \\ \mathit{VoidVal} \\ \hline \mathit{attris} : \mathbb{P}\, \mathit{Variable}_{\copyright} \\ \hline \end{array}$$

SOFL variable values are modeled as:

$$\mathit{Value} \mathrel{\widehat=} \mathit{IntVal} \cup \mathit{BoolVal} \cup \mathit{CompVal} \cup \mathit{VoidVal} \cup \mathit{Nil}$$

The function *type_of* captures the relation between expression values and types:

$$\begin{array}{|l} \mathit{type\_of} : \mathit{Value} \rightarrow \mathit{Type} \\ \hline \forall\, v : (\mathit{Value} \setminus \mathit{Nil}) \bullet \\ \quad v \in \mathit{IntVal} \Rightarrow \mathit{type\_of}(v) \in \mathit{IntType} \\ \quad v \in \mathit{BoolVal} \Rightarrow \mathit{type\_of}(v) \in \mathit{BoolType} \\ \quad v \in \mathit{VoidVal} \Rightarrow \mathit{type\_of}(v) \in \mathit{VoidType} \\ \quad v \in \mathit{CompVal} \Rightarrow \\ \qquad \mathit{type\_of}(v) \in \mathit{CompType} \\ \qquad \{a : v.\mathit{attris} \bullet (a.\mathit{id}, a.\mathit{type})\} = \mathit{type\_of}(v).\mathit{field} \end{array}$$

Variables

Variable references are specified by the class *Variable* containing attributes *id* denoting the variable's name (identifier), *type* denoting the declared type, and $\mathcal{V}$ denoting the current value of the variable.

Variable

$id : Id;\ type : Type;\ \mathcal{V} : Value$

$\mathcal{V} \notin Nil \Rightarrow type\_of(\mathcal{V}) = type$

OutVal

$val! : Value$

$val! = \mathcal{V}$

Change

$\Delta(\mathcal{V})$

Assign

$\Delta(\mathcal{V})$
$val? : Value$

$\mathcal{V}' = val?$

ToNil

$\Delta(\mathcal{V})$

$\mathcal{V}' \in Nil$

A variable can output its value (*OutVal*), to reset to a nil value (*ToNil*), to change its value (*Change*) and to be assigned to a new value (*Assign*).

In SOFL, a variable can have a pre-state form. The following function associates variables with their pre-states.

$$Before\_After : Variable \rightarrowtail Variable$$

$$\forall (v_1, v_2) : Before\_After \bullet v_1.type = v_2.type \wedge v_1.id = before(v_2.id)$$

4.2 SOFL Expressions

A SOFL expression is either a constant value (e.g `5' or `true'), a `variable' (e.g. `x' of type integer), an `unary-expression' (e.g. a negation or a bound expression), a `binary-expression' (e.g. an equality or a plus expression), or a `component-expression' (e.g. mypoint.x in the SOFL example of Section 2).

In Object-Z, the general SOFL expression type can be modeled as a class-union

$$Exp \mathrel{\widehat{=}} Constant \cup Variable \cup UnaryExp \cup BinaryExp \cup ComponentExp$$

where $Constant \mathrel{\widehat{=}} IntVal \cup BoolVal$ and *Variable* is defined in the previous section. The *UnaryExp*, *BinaryExp* and *ComponentExp* are specified in the following.

A SOFL predicate (boolean expression) is modeled as

$$Pred == \{e : Exp \mid e.\mathcal{V} \in BoolVal\}$$

If an expression consists of component sub-expression(s), then the meaning (value) of this expression is dependent upon the meaning of its component sub-expression(s). Each SOFL expression can be modeled as an object. For example, an unary (binary) expression is modeled as an object with one (two) sub-expression(s), and a value derived from the value(s) of the sub-expression(s).

The logical `negation' and the `bound' expression (test whether a variable has a non-nil value) of SOFL can be modeled as:

$$
\begin{array}{|l}
\textit{Negation} \\ \hline
\begin{array}{|l}
e : Exp_{\textcircled{\scriptsize L}} \\
\Delta \\
\mathcal{V} : Value \\ \hline
e.\mathcal{V} \in Nil \Rightarrow \mathcal{V} \in Nil \\
e.\mathcal{V} \notin Nil \Rightarrow \\
\quad type\_of(\mathcal{V}) \in BoolType \\
\quad type\_of(e.\mathcal{V}) \in BoolType \\
\quad \mathcal{V}.val = \neg\, e.\mathcal{V}.val
\end{array}
\end{array}
\qquad
\begin{array}{|l}
\textit{Bound} \\ \hline
\begin{array}{|l}
e : Exp_{\textcircled{\scriptsize L}} \\
\Delta \\
\mathcal{V} : Value \\ \hline
type\_of(\mathcal{V}) \in BoolType \\
e \in Variable \\
\mathcal{V}.val = \neg\, (e.\mathcal{V} \in Nil)
\end{array}
\end{array}
$$

The subscript `$\textcircled{\scriptsize L}$' on the type *Exp* specifies that *e* is local references, sharably contained objects, ensuring that the reference structure of expressions is acyclic.

Note that the attribute $\mathcal{V}$ representing the meaning of an expression is a secondary attribute. Syntactically, the declarations of secondary attributes appear below the Δ separator placed in the declaration section of the state schema. The values of the secondary attributes are always subject to change. This is achieved by implicit inclusion of secondary attributes in the Δ-list of every operation in a system. For a detailed discussion on secondary attributes see [8]. The secondary attribute $\mathcal{V}$ captures the way the value of an expression depends on the value of its sub-expressions; i.e. $\mathcal{V}$ captures the meaning (dynamic semantics) of an expression.

SOFL unary expressions can be defined as:

$$UnaryExp \mathrel{\widehat{=}} Negation \cup Bound$$

The *Equality* and *PlusExp* binary expressions can be defined as:

$$
\begin{array}{|l}
\textit{Equality} \\ \hline
\begin{array}{|l}
e_1, e_2 : Exp_{\textcircled{\scriptsize L}} \\
\Delta \\
\mathcal{V} : Value \\ \hline
(e_1.\mathcal{V} \in Nil \lor e_2.\mathcal{V} \in Nil) \Rightarrow \\
\quad \mathcal{V} \in Nil \\
\neg\, (e_1.\mathcal{V} \in Nil \lor e_2.\mathcal{V} \in Nil) \Rightarrow \\
\quad type\_of(\mathcal{V}) \in BoolType \\
\quad type\_of(e_1.\mathcal{V}) = type\_of(e_2.\mathcal{V}) \\
\quad \mathcal{V}.val = (e_1.\mathcal{V}.val = e_2.\mathcal{V}.val)
\end{array}
\end{array}
\qquad
\begin{array}{|l}
\textit{PlusExp} \\ \hline
\begin{array}{|l}
e_1, e_2 : Exp_{\textcircled{\scriptsize L}} \\
\Delta \\
\mathcal{V} : Value \\ \hline
(e_1.\mathcal{V} \in Nil \lor e_2.\mathcal{V} \in Nil) \Rightarrow \\
\quad \mathcal{V} \in Nil \\
\neg\, (e_1.\mathcal{V} \in Nil \lor e_2.\mathcal{V} \in Nil) \Rightarrow \\
\quad type\_of(\mathcal{V}) \in IntType \\
\quad type\_of(e_1.\mathcal{V}) \in IntType \\
\quad type\_of(e_2.\mathcal{V}) \in IntType \\
\quad \mathcal{V}.val = (e_1.\mathcal{V}.val + e_2.\mathcal{V}.val)
\end{array}
\end{array}
$$

Other binary expressions, such as `less-than', `times', `conjunction', `disjunction', etc, can be similarly defined.

The SOFL binary expressions can be defined as

$$BinaryExp \mathrel{\widehat{=}} Equality \cup PlusExp \cup \ldots$$

In this specification, the abstract syntax, static semantics and dynamic semantics for each kind of expression are specified in the one Object-Z class.

Component expression

Syntactically, a component expression is a list of identifiers with dots as the separation marks. Abstractly, a component expression is a list of variable references with a validity constraint (static semantic condition). For example, suppose mypoint.x is a component expression, variable x must be an attribute of the composite value referenced by mypoint. The meaning of a component expression is a value which can be deduced from the expression. The following is a model of component expressions:

$$
\begin{array}{l}
\textit{ComponentExp} \\
\hline
s : \mathrm{seq}\, \textit{Variable} \\
\Delta \\
\mathcal{V} : \textit{Value} \\
\hline
\#s > 1 \land \forall\, i : 1..(\#s - 1) \bullet \\
\quad s(i).\mathcal{V} \in \textit{CompVal} \land s(i+1) \in s(i).\mathcal{V}.\textit{attris} \land \mathcal{V} = \textit{last}(s).\mathcal{V}
\end{array}
$$

4.3 Condition Processes

Condition processes are a fundamental construct in SOFL specification language. A condition process is similar to a `procedure' or `function' in programming languages. Input and output variables are partitioned into ports. Essentially, a port consists of a list of variables. A list of ports is modeled as:

$$\textit{Ports} == \mathrm{seq}_1\, \mathrm{seq}_1\, \textit{Variable}$$

For each port variable, there is a data flow link (in a CDFD). Let

$$\textit{PortLinks} == \mathrm{seq}_1\, \mathrm{seq}_1\, \textit{DataFlow}$$

represent a list of port-links, where *DataFlow* will be defined in Section 4.5.

The following relationship associates the ports and port-links:

$$
\begin{array}{l}
\textit{Connect} : \textit{Ports} \leftrightarrow \textit{PortLinks} \\
\hline
\forall\, \textit{ports} : \textit{Ports};\ \textit{dataflows} : \textit{PortLinks} \bullet \\
\quad \textit{ports}\ \underline{\textit{Connect}}\ \textit{dataflows} \Leftrightarrow \\
\qquad \#\textit{ports} = \#\textit{dataflows} \\
\qquad \forall\, i : \mathrm{dom}\, \textit{ports} \bullet \#\textit{ports}(i) = \#\textit{dataflows}(i) \land \\
\qquad\quad \forall\, j : \mathrm{dom}\, \textit{ports}(i) \bullet \textit{ports}(i)(j) = \textit{dataflows}(i)(j).\textit{left}
\end{array}
$$

For a condition process, if all variables of an input port have non-nil values, then this condition process is *enabled* to fire and this input port is called active port. The following function computes the indexes of active ports from a list of ports:

$$
\begin{array}{l}
\textit{Active\_ports} : \textit{Ports} \rightarrow \mathbb{P}\, \mathbb{N} \\
\hline
\forall\, \textit{ports} : \textit{Ports} \bullet \textit{Active\_ports}(\textit{ports}) = \\
\quad \{ i : \mathrm{dom}\, \textit{ports} \mid \forall\, j : \mathrm{dom}\, \textit{ports}(i) \bullet \textit{ports}(i)(j).\mathcal{V} \notin \textit{Nil} \}
\end{array}
$$

A condition process can be further decomposed into a specification module or implementation module. If a condition process is decomposed into a specification module, then the meaning of the condition process is the meaning of the decomposed module. On the other hand, if a condition process is decomposed into an implementation module, then the condition process should have a totally specified meaning. The following Object-Z class models condition processes:

CP

$name : Id$
$in, out : Ports$ [input and output ports]
$outflows : PortLinks$ [output data flow links]
$valid\_port : \mathbb{N}$ [a port in which all variables have values]
$rd : \mathbb{P}\, Variable_{Ⓛ}$ [read variables]
$\widetilde{wr\_wr} : Variable_{Ⓛ} \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow Variable_{Ⓛ}$ [before, after write variables]
$pre, post : Pred_{Ⓛ}$ [pre and post conditions]
$decomposition : S\_Module \cup I\_Module$ [further decomposition]
Δ
$active : \mathbb{B}$

$\widetilde{wr\_wr} \subseteq Before\_After$
$out\ \underline{Connect}\ outflows$
$active = (Active\_ports(in) \neq \varnothing)$
$active \Rightarrow valid\_port \in Active\_ports(in)$
$(pre.\mathbf{local} \cap Variable) \subseteq (\mathrm{ran}(\bigcup \mathrm{ran}\, in)) \cup rd$
$(post.\mathbf{local} \cap Variable) \subseteq (\mathrm{ran}(\bigcup \mathrm{ran}(in \cup out))) \cup rd \cup \mathrm{dom}\ \widetilde{wr\_wr} \cup \mathrm{ran}\ \widetilde{wr\_wr})$

$PassToBeforeVars \mathrel{\widehat=} \bigwedge (v_1, v_2) : \widetilde{wr\_wr} \bullet v_2.OutVal \parallel v_1.Assign$
$Update \mathrel{\widehat=} \bigwedge v : (\mathrm{ran}(\bigcup \mathrm{ran}\, out) \cup \mathrm{ran}\ \widetilde{wr\_wr}) \bullet v.Change$

ResetValidPort

$\Delta(valid\_port)$

$active' \Rightarrow (valid\_port' \in Active\_ports(in) \setminus \{valid\_port\})$

$ResetPortVars \mathrel{\widehat=} \bigwedge i : \mathrm{dom}\, in(valid\_port) \bullet in(valid\_port)(i).ToNil$
$Flow \mathrel{\widehat=} \Box\ i : \mathrm{dom}\, outflows \bullet \bigwedge j : \mathrm{dom}\, outflows(i) \bullet outflows(i)(j).\mathcal{M}$
$Reset \mathrel{\widehat=} ResetPortVars;\ ResetValidPort$
$\mathcal{M}_{defined} \mathrel{\widehat=} [pre.\mathcal{V}.val] \bullet (PassToBeforeVars;\ Update;\ [post.\mathcal{V}.val];\ Reset)$
$\quad \Box\ [\neg\ pre.\mathcal{V}.val] \bullet (Update;\ Reset)$
$\mathcal{M} \mathrel{\widehat=} [decomposition \in I\_Module] \bullet \mathcal{M}_{defined};\ Flow$
$\quad \Box\ [decomposition \in S\_Module] \bullet decomposition.\mathcal{M}$

If a pre-condition of a condition process is true, then the meaning of the condition process is defined as: firstly, load pre-state (write) variables with current values (*PassToBeforeVars*), updates post-state (write) variables according to the post-condition (*Update*; $[post.\mathcal{V}.val]$), then reset the active port variables to nil and the valid-port (*Reset*), finally activate its output data-flow links (*Flow*).

Note that the term $pre.\mathbf{local}$ (in the class invariant) denotes the directly or indirectly local (sharably contained) objects of the *pre* object [5]. The parallel operator '$\|$' is like conjunction, except that if an output parameter from one component operation has a complementary input parameter in the other component (i.e. both have the same name apart from the ? and ! decorations) these input/output parameters are identified and hidden from the environment, i.e. they are no longer inputs from, or outputs to, the environment. The choice operator '$\sqcap$' defines a system operation in terms of a choice between two given operations. The choice is angelic in that if only one component operation is enabled that operation will be selected; if both are enabled the selection is non-deterministic. The sequential operator '$;$' defines a system operation in terms of the sequential composition of two given operations. The semantics of these operators are defined in [2, 13, 20].

4.4 SOFL Input and Output Environments

The sources or destinations of a data flow in SOFL can be a condition process, input or output environments. We model input and output environments as the following classes:

$$
\begin{array}{|l}
\textit{InEnv} \\
\hline
\begin{array}{|l}
right : Variable \\
outflow : DataFlow \\
active : \mathbb{B} \\
\hline
active = true \\
right = outflow.left
\end{array} \\
\mathcal{M} \mathrel{\widehat=} right.Assign;\ outflow.\mathcal{M} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{OutEnv} \\
\hline
\begin{array}{|l}
left : Variable \\
inflow : DataFlow
\end{array} \\
\mathcal{M} \mathrel{\widehat=} left.OutVal \\
\hline
\end{array}
$$

Note that input environment is always active. The meaning of the input environment object is to receive a value and then pass it through a data-flow link, while the meaning of the output environment object is modeled as a simple output value action.

4.5 Data Flows

A data-flow in SOFL graphical notation can be a simple data-flow (with only one destination) or a complex data-flow connector. It is modeled as:

$$
\begin{array}{ll}
DataFlow \mathrel{\widehat=} SimpleDF \cup DFConnector & \text{where} \\
DFConnector \mathrel{\widehat=} ConditionDF \cup CaseDF \cup BroadCastDF \cup ChoiceDF &
\end{array}
$$

All SOFL data-flows have a similar structure which includes a (left-hand side) source and one or more (right-hand side) destinations and information is passed from

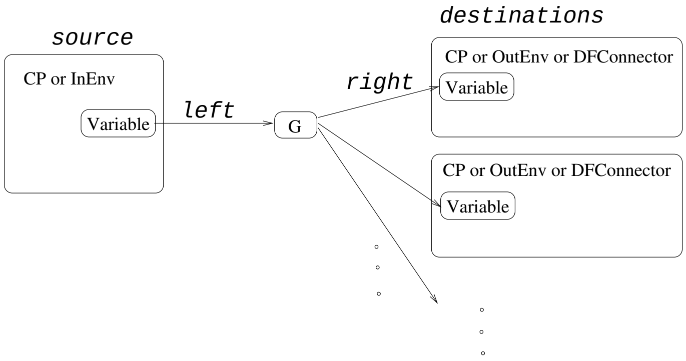

Figure 5: A model of a generic data-flow

source to destinations. In Object-Z, the common part of the data-flows can be modeled by a generic class:

GenericFlow

$$
\begin{array}{l}
source : (CP \cup InEnv)_{\textcircled{L}} \\
left : Variable \\
right : \mathrm{seq}_1\, Variable \\
dests : \mathrm{seq}_1\, (CP \cup OutEnv \cup DFConnectors)_{\textcircled{L}} \\
\hline
source \in CP \Rightarrow left \in \mathrm{ran}(\bigcup \mathrm{ran}\, source.out) \\
source \in InEnv \Rightarrow source.right = left \\
\forall\, i : \mathrm{dom}\, right \bullet right(i).type = left.type \\
\#right = \#dests \\
\forall\, i : \mathrm{dom}\, dests \bullet \\
\quad dests(i) \in CP \Rightarrow right(i) \in (\mathrm{ran}(\bigcup \mathrm{ran}\, dests(i).in)) \\
\quad dests(i) \notin CP \Rightarrow right(i) = dests(i).left
\end{array}
$$

The class invariants captures appropriate linkages according to different kinds of source and destinations. Figure 5 illustrates this model.

A simple data-flow has only one destination. The meaning of a simple data-flow is to pass a value from the left to the right. Simple data-flows are modeled as:

SimpleDF

$$
\begin{array}{l}
GenericFlow \\
\hline
\#right = 1 \\
dests(1) \notin DFConnectors \\
\hline
\mathcal{M} \mathrel{\widehat=} left.OutVal \parallel right(1).Assign
\end{array}
$$

Note that if the destination of the *SimpleDF* is a data-flow connector, then the models of data-flow connectors will include the *SimpleDF* model.

A condition data-flow has two destinations, yes-flow and no-flow, and the value from the left can only be passed to one of the right depending on the truth value of the condition predicate. Condition data-flows are modeled as:

$$
\begin{array}{l}
\textbf{\textit{ConditionDF}} \\
\hline
GenericFlow \\
\quad condition : Pred \\
\quad \#right = 2 \\
YesFlow \mathrel{\widehat{=}} [dests(1) \in CP] \,\Box\, ([dests(1) \notin CP] \bullet dests(1).\mathcal{M}) \\
NoFlow \mathrel{\widehat{=}} [dests(2) \in CP] \,\Box\, ([dests(2) \notin CP] \bullet dests(2).\mathcal{M}) \\
\mathcal{M} \mathrel{\widehat{=}} [condition.\mathcal{V}.val] \bullet (left.OutVal \,\|\, right(1).Assign);\ YesFlow \\
\qquad \Box\ [\neg\ condition.\mathcal{V}.val] \bullet (left.OutVal \,\|\, right(2).Assign);\ NoFlow
\end{array}
$$

Note that meaning of the conditional data-flow is not just to pass value from source to destination, in the case of that the destination is a data-flow connector or an output environment, then the destination is also activated.

The case data-flows *CaseDF* can be similarly modeled (omitted).

Broadcast data-flows and Choice data-flows are modeled as

$$
\begin{array}{l}
\textit{BroadcastDF} \\
\hline
GenericFlow \\
\mathcal{M} \mathrel{\widehat{=}} \wedge i : \mathrm{dom}\, right \bullet \\
\quad ((left.OutVal \,\|\, right(i).Assign); \\
\quad\ [dests(i) \in CP] \,\Box \\
\quad\ [dests(i) \notin CP] \bullet dests(i).\mathcal{M})
\end{array}
$$

$$
\begin{array}{l}
\textit{ChoiceDF} \\
\hline
GenericFlow \\
\mathcal{M} \mathrel{\widehat{=}} \Box\ i : \mathrm{dom}\, right \bullet \\
\quad ((left.OutVal \,\|\, right(i).Assign); \\
\quad\ [dests(i) \in CP] \,\Box \\
\quad\ [dests(i) \notin CP] \bullet dests(i).\mathcal{M})
\end{array}
$$

In the class *BroadcastDF*, the distributed conjunction operator '$\wedge i : \ldots$' captures precisely the nature of the broadcast-dataflow. In the class *ChoiceDF* the distributed choice operator '$\Box\ i : \ldots$' captures precisely the non-deterministic nature of the choice-dataflow.

4.6 Condition Data Flow Diagrams (CDFDs)

A CDFD consists of a group of nodes (condition processes, input and output environments) and a group of data-flow links between those nodes. The minimum requirement for a CDFD is to consist of at least one condition process and one input environment. As SOFL's CDFD allows non-determinism, the semantics of a CDFD also have non-deterministic behaviour. CDFDs are modeled as:

$$
\begin{array}{|l}
\textit{CDFD} \\
\hline
\begin{array}{|ll}
\hline
\textit{nodes} : \mathbb{P}(\textit{CP} \cup \textit{InEnv} \cup \textit{OutEnv}) & \text{[data-flow nodes]} \\
\textit{dfs} : \mathbb{P}\, \textit{DataFlow} & \text{[data-flow links]} \\
\Delta & \\
\textit{active\_nodes} : \mathbb{P}(\textit{CP} \cup \textit{InEnv}) & \\
\hline
\textit{nodes} \cap \textit{CP} \neq \varnothing \wedge \textit{nodes} \cap \textit{InEnv} \neq \varnothing & \\
(\bigcup\{d : \textit{dfs} \bullet d.\mathbf{local}\} \cap (\textit{CP} \cup \textit{InEnv} \cup \textit{OutEnv})) = \textit{nodes} & \\
\textit{active\_nodes} = \{c : \textit{nodes} \mid c.\textit{active}\} & \\
\hline
\end{array} \\
\textit{Step} \mathrel{\widehat{=}} \Box\; n : \textit{active\_nodes} \bullet n.\mathcal{M} \\
\textit{Activate} \mathrel{\widehat{=}} [(\textit{active\_nodes} \cap \textit{CP}) = \varnothing] \bullet \textit{Step};\ \textit{Activate} \\
\qquad\qquad \Box\ [(\textit{active\_nodes} \cap \textit{CP}) \neq \varnothing] \\
\textit{Process} \mathrel{\widehat{=}} [(\textit{active\_nodes} \cap \textit{CP}) \neq \varnothing] \bullet \textit{Step};\ \textit{Process} \\
\qquad\qquad \Box\ [(\textit{active\_nodes} \cap \textit{CP}) = \varnothing] \\
\mathcal{M} \mathrel{\widehat{=}} \textit{Activate};\ \textit{Process} \\
\hline
\end{array}
$$

A single trace step is to select an active-node and execute the node (captured by the operation *Step*). At the start, all active-nodes in a CDFD are input environments. By executing a number of input environments, some condition processes will be activated (*Activate*). Then, a chain of activations continues until there is no active condition process left (*Process*).

4.7 Specification Modules

A SOFL specification module consists of a name, a group of user-defined types, a group of variables (which will be the `read' or `write' variables (data stores) of its condition process), a module invariant, a group of condition processes and a CDFD:

$$
\begin{array}{|l}
\textit{S\_Module} \\
\hline
\begin{array}{|ll}
\hline
\textit{name} : \textit{Id} & \\
\textit{definedtypes} : \mathbb{P}\, \textit{UserDefType} & \\
\textit{vars} : \mathbb{P}\, \textit{Variable}_{Ⓛ} & \\
\textit{inv} : \textit{Pred}_{Ⓛ} & \\
\textit{cps} : \mathbb{P}\, \textit{CP}_{Ⓒ} & \\
\textit{cdfd} : \textit{CDFD} & \\
\hline
\forall\, v : \textit{vars} \bullet v.\textit{type} \in (\textit{PredefinedType} \cup \textit{definedtype}) & \text{[i]} \\
\forall\, c : \textit{cps} \bullet \forall\, v : \operatorname{ran}(\bigcup \operatorname{ran}(c.\textit{in} \cup c.\textit{out})) \bullet & \text{[ii]} \\
\qquad v.\textit{type} \in (\textit{PredefinedType} \cup \textit{definedtype}) & \\
\forall\, c : \textit{cps} \bullet (c.\textit{rd} \cup \operatorname{ran} c.\, {}^{\sim}\textit{wr\_wr}) \subseteq \textit{vars} & \text{[iii]} \\
\textit{cps} = (\textit{cdfd}.\textit{nodes} \cap \textit{CP}) & \text{[iv]} \\
\forall\, v_1, v_2 : (\mathbf{local} \cap \textit{Variable}) \bullet & \text{[v]} \\
\qquad v_1.\textit{id} = v_2.\textit{id} \Rightarrow v_1 = v_2 & \\
\hline
\end{array} \\
\mathcal{M} \mathrel{\widehat{=}} [\textit{inv}.\mathcal{V}.\textit{val}];\ \textit{cdfd}.\mathcal{M};\ [\textit{inv}.\mathcal{V}.\textit{val}] \\
\hline
\end{array}
$$

The class invariants [i,ii,iii] ensure all variables types are defined. The class invariant [iv] ensures the condition process of the CDFD are specified in the s_module. The last class invariant [v] ensures two variables with the same name to be the same object (and hence the same value). The meaning of a specification module is the meaning of its CDFD and holds the module invariants before and after executing the CDFD.

This completes the specification of the semantics of SOFL specification notation. The SOFL implementation language is very close to programming languages. It can be modeled by applying the approach demonstrated in [6]. Basically, a SOFL implementation module can also be modeled as an object of the class:

I_Module

$\ldots$

5 Conclusions

SOFL is a graphical and textual formal notation that integrates VDM (for modeling system component state), data-flow-diagrams and Petri nets (for modeling system interaction and behaviours). This paper has presented an object-oriented approach to specify the semantics for SOFL. The main semantics design idea is to model SOFL constructs, such as condition processes, data-flows, specification modules, etc, as objects. The advantages of this approach is that the abstract syntax and static and dynamic semantics for each the SOFL construct are grouped together and captured in an Object-Z class; hence the language model is structural, concise and easily extendible. SOFL's CDFD is based on Data Flow Diagrams, consequently, this paper includes a formal object model for DFDs.

Various object modeling techniques have been used in the semantic representation of SOFL. For example, inheritance is used effectively to illustrate the common and the differences between the various SOFL data-flow component constructs. The relationships between condition processes and modules, the acyclic decomposition chain and the partition of condition processes by modules, are well captured by the notation of object containment of Object-Z. This paper demonstrated that Object-Z is a candidate for the semantic description of the integrated textual/graphical specification notations. This object-oriented semantics approach promote reusability. For example, the expressions model of programming languages [6] are reused to define the most part of the semantic class model for SOFL expressions. As object-orientation is becoming more acceptable, mature and popular, the object-oriented approach to language semantics is promising.

We believe this specification together with the operational semantic description of SOFL [16] can provide a useful document for developing support tools for SOFL. For example, the class signature and invariants can generate the well-typed rules for a SOFL type-checker; class operations (meanings) can derive the reasoning rules for a proof tool or an animation tool. If the support tool for SOFL will be implemented in object-oriented program languages, say C++ or Java, then we believe parts of the implementation can be readily generated from the semantic model presented in this paper.

Acknowledgements

We would like to thank Liam O' Brien and anonymous referees for many helpful comments on this paper. We would also like to thank the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No.90264960) and Hiroshima City University under Hiroshima City University Grant for Special Academic Research (No.9842 and No.A440) to support this research.

References

[1] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. International Series in Computer Science. Prentice-Hall, 1982.

[2] S. Butler and R. Duke. Defining composition operators for object interaction. *Object Oriented Systems*, 5(1):1–16, 1998.

[3] J.S. Dong. Living with Free Type and Class Union. In *The 1995 Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 304–312. IEEE Computer Society Press, December 1995.

[4] J.S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12 & 9*, pages 181–190. Prentice-Hall, November 1993.

[5] J.S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.

[6] J.S. Dong, R. Duke, and G. Rose. An Object-Oriented Denotational Semantics of A Small Programming Language. *Object-Oriented Systems (OOS)*, 4(1):29–52, Chapman & Hall 1997.

[7] J.S. Dong and S. Liu. An Object-Oriented Denotational Description of the SOFL Semantics. Technical report, Faculty of Information Sciences, Hiroshima City University, Asminami-ku Hiroshima, Japan, 1998.

[8] J.S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. In Alex Stoyenko, editor, *The First IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)*, pages 31–38, Ft. Lauderdale, USA, November 1995. IEEE Computer Society Press.

[9] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lect. Notes in Comput. Sci.*, pages 242–262. Springer-Verlag, 1990.

[10] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[11] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[12] A. S. Evans and A. N. Clark. Foundations of the unified modeling language. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, 1998.

[13] A. Griffiths. A Semantics for Recursive Operations in Object-Z. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, Discrete Mathematics and Theoretical Computer Science, pages 81–102. Springer-Verlag, 1997.

[14] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *Object-Oriented Systems*, 2:195–215, Chapman & Hall 1995.

[15] I. Hayes. A small language definition in Z. Technical Report 94-50, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1994.

[16] C. Ho-Stuart and S. Liu. An Operational Semantics for SOFL. In *Proceedings of The 1997 Asia-Pacific Software Engineering Conference*, pages 52–61, Hong Kong, 1997. IEEE Computer Society Press.

[17] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. An Approach to Specifying and Verifying Safety-Critical Systems with Practical Formal Method SOFL . In Alex Stoyenko, editor, *The 4th IEEE International Conference on Engineering Complex Computer Systems (ICECCS'98)*, pages 100–114, Monterey, California, USA, August 1998. IEEE Computer Society Press.

[18] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1), January 1998.

[19] J. Nicholls. Z Base Standard — Version 1.2. Prepared by members of the Z standards Panel, September 1995.

[20] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

[21] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[22] S. Stepney. *High integrity compilation: A case study*. Prentice-Hall, 1993.

[23] J. E. Stoy. *Denotational semantics : the Scott-Strachey approach to programming language theory*. Cambridge, Mass. : MIT Press, 1977.

[24] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *the Sixth Annual Z User Meeting, York, UK.*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.