The Semantics of Extended SOFL*

Jin Song Dong†, Shaoying Liu‡
†National University of Singapore
Email: dongjs@comp.nus.edu.sg
‡Hosei University, Japan
Email: sliu@k.hosei.ac.jp

Abstract

Recently SOFL (Structured-Object-based-Formal Language) has been extended to a formal object-oriented language and method while keeping its structured features. This extension allows powerful object-oriented reuse mechanisms, such as class inheritance and object composition, to be utilized in the early design phases. This paper presents the semantics for this extended SOFL and further demonstrates the extendibility and reusability of the object-oriented semantic models of SOFL.

Keywords*: Structured Methods, Object-Oriented Methods, Reusable Semantics*

1 Introduction

The design of complex systems requires powerful mechanisms for modeling state, events and interactive behaviour; as well as for visualising and structuring systems in order to control complexity. Usually, many methods are used in the development of such complex systems. Methods can be used in effective combination only if the semantic links between individual methods are clearly established and the semantics of those methods are well integrated. A recent research trend in software specification and design is in methods integration. The semantic complexity of these integrated methods presents further challenges to the traditional semantic approaches.

Like RAISE,SOFL (Structured-Object-based-Formal Language) [1] is an integrated formal method based on VDM. Unlike RAISE, SOFL is a graphical and textual mixed notation that integrates VDM (for modeling system operations over states) with data-flow-diagrams (DFD) and Petri nets (for modeling system interaction and behaviours). SOFL can be a vehicle for introducing formal methods (i.e. VDM) to industries with structured development background (e.g. DFD users).

Recently SOFL has been extended to a formal object-oriented method while keeping its structured features [2]. This extension allows object-oriented reuse mechanisms, such as class inheritance and object composition, to be utilized in the early design phases. The graphical notation of SOFL has also been enriched by introducing the notion of class. In this paper, we extend and restructure our previous formal semantics of SOFL [3] to reflect the new language extensions. Furthermore this paper demonstrates the extendibility and reusability of using object-oriented approaches to the semantics of integrated formal methods.

The semantics work not only is useful as a high level specification for the development of the SOFL case tools, but also aids the language design itself. For example, the complex semantic representation of the input ports for a conditional process in the previous semantic model motivates us to simplify the process construct definition in the new version.

We have used Object-Z [4, 5] notation as the meta-language to define the SOFL semantics in [3]. Object-Z was selected because the semantics of Object-Z itself is well studied [6, 7, 8] and it has been effectively applied to programming language semantics [9, 10]. In this paper, the semantics of the new SOFL is defined by extending the original SOFL semantics [3]. A large part of the semantic model defined in [3] is reused in this work. For example, the semantic models for graphical data flow connectors can be directly included without any modification,

This paper focuses on the newly extended part of the semantics. Some semantic models defined previously [3] are included in order to make this paper self-contained, but some

*This work is supported in part by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research (B) (No. 11694173) and (C) (No. 11680368), and a research grant (Integrated Formal Methods) from National University of Singapore (No. RP3991615)

others are omitted when they are trivial. This paper assumes that the reader is familar with SOFL and Object-Z.

2 The Semantics of Extended SOFL

The major extensions to SOFL are the introduction of class constructs and composite process definition using CDFD in a class. Those extensions lead to a restructured (improved) semantic model for process, class and module constructs, and the introduction of some new graphical CDFD components. Figure 1 illustrates the two SOFL semantic models (the previous one in [3] and the current one). The similarities and differences between SOFL constructs can be more clearly captured in the current semantic model. For example, the notion that a class type is similar to a composite type is captured by the inheritance relationship between the semantic models. The semantic model also clearly captures that both process and module have an interface and module and class share a common structure. The semantic models of processes are presented in an incremental and structured way, in contrast to the previous flat lengthy semantic models.

2.1 Types, Interfaces and Processes

Many semantics definitions in [3] are reusable in this new semantics. We have included the semantic definition of 'variable' (defined in [3]) in the following section which will help to understand the essential semantics modifications and extensions on SOFL types, processes and classes.

2.1.1 Variables

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

$$before : Id \rightarrowtail Id$$

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

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

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

OutVal

$val! : Value$

$val! = \mathcal{V}$

Change

$\Delta(\mathcal{V})$

ToNil

$\Delta(\mathcal{V})$

$\mathcal{V}' \in Nil$

Assign

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

$\mathcal{V}' = val?$

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

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

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

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

The semantics for SOFL variable values and expressions (including predicates) are defined by the class-union construct in [3] as:

$$Value \mathrel{\widehat{=}} VoidVal \cup CompVal \cup IntVal \cup BoolVal \cup Nil$$
$$Exp \mathrel{\widehat{=}} Constant \cup Variable \cup UnaryExp \cup BinaryExp \cup ComponentExp$$
$$Pred == \{e : Exp \mid e.\mathcal{V} \in BoolVal\}$$

where

VoidVal

$\mathcal{V} : Value$

$\mathcal{V} = self$

CompVal

VoidVal

$attris : \text{seq}\ Variable$

The details of other semantics components are directly reusable and unsurprising, and not repeated here. Note that the semantics of SOFL objects (instances of SOFL classes) can be directly modeled as a composite value (instances of *CompVal*).

2.1.2 Type

The first major extension is the notion of class in the SOFL specification language. The type system is extended, as classes can be used as types to instantiate instances.

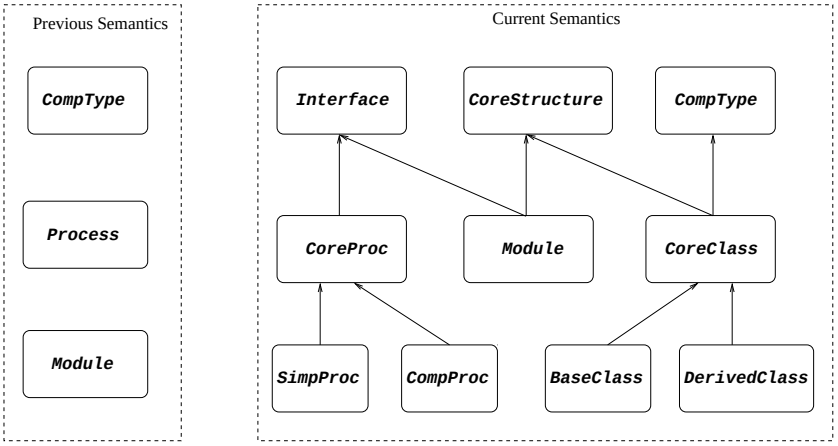

Figure 1. Comparison of the previous and current semantic models

In [3], SOFL value types is modeled as a class-union:

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

(*PreDefType* is defined in [3].)

User-defined types including classes, composite types, etc, are modeled as:

$$UserDefType \mathrel{\widehat{=}} Class \cup CompType \cup ...$$

A composite type has a name and a signature which contains a set of declarations. It is modeled in [3] as:

CompType

$name : Id$
$sig : Id \pfun Type$

Note that *Class* type is similar to *CompType* and will be defined later.

2.1.3 Interface of a process

The SOFL Process is a fundamental construct in the specification language. A process is similar to a 'procedure' or 'function' in programming languages. Input and output variables are partitioned into ports. A port consists of a list of variables, and for each port variable, there is a data flow link (in a CDFD). As a process can be further decomposed into a module, the process and the decomposed module must have the same interface. The interface of a process or a module is defined as:

Interface

$in : \mathrm{seq}\ Variable$ [input port]
$out : Ports$ [output ports]
$outflows : PortLinks$ [output data flow links]
$rd : \mathbb{P}\ Var_{\textcircled{L}}$ [read variables]
$\tilde{}wr\_wr : Var_{\textcircled{L}} \pinj Var_{\textcircled{L}}$ [before, after write variables]
Δ
$interfacevars : \mathbb{P}\ Vars$

$interfacevars = \mathrm{ran}\ in \cup \mathrm{ran}(\bigcup \mathrm{ran}\ out) \cup rd \cup \mathrm{dom}\ \tilde{}wr\_wr \cup \mathrm{ran}\ \tilde{}wr\_wr$
$\tilde{}wr\_wr \subseteq Before\_After$
$outflows \neq \varnothing \Rightarrow$
$\quad \#out = \#outflows \wedge out\ \underline{Connect}\ outflows$

The semantics of the subscript '$_{\textcircled{L}}$ (local objects [11]) appended to the types of the attributes *rd* and $\tilde{}wr\_wr$ captures that all variables referenced by *rd* and $\tilde{}wr\_wr$ are local.

One difference to the previous semantics [3] is that the input port is simplified to a single port (rather than multiple ports).

Each port variable connects a data flow link (in a CDFD).

$Connect : Ports \leftrightarrow PortLinks$

$\forall ports : Ports;\ dataflows : PortLinks \bullet$
$\quad ports\ \underline{Connect}\ dataflows \Leftrightarrow$
$\quad\quad \#ports = \#dataflows$
$\quad\quad \forall i : \mathrm{dom}\ ports \bullet$
$\quad\quad\quad \#ports(i) = \#dataflows(i) \wedge$
$\quad\quad\quad \forall j : \mathrm{dom}\ ports(i) \bullet$
$\quad\quad\quad\quad ports(i)(j) = dataflows(i)(j).left$

where

$$Ports == \text{seq}_1\ \text{seq}_1\ Variable$$
$$PortLinks == \text{seq}_1\ \text{seq}_1\ DataFlow$$

DataFlow will be defined in Section 2.4.

2.1.4 Process

A process can be either a simple process or a composite process: $Proc \mathrel{\widehat=} SimpProc \cup CompProc$

A simple process in a module can be further decomposed into another module, while a process (simple or composite) in a class cannot be further decomposed. Some common core features of a process can be defined as:

CoreProc

Interface

$name : Id$
$within : Class \cup Module$
$decomp : Module \cup Nil$
Δ
$active : \mathbb{B}$

$active = \forall\, i : \text{dom}\, in \bullet in(i).\mathcal{V} \notin Nil$

$Flow \mathrel{\widehat=} (\Box\ i : \text{dom}\, outflows \bullet \wedge j : \text{dom}\, outflows(i) \bullet$
$outflows(i)(j).\mathcal{M})$
$\Box\ [out = \varnothing]$

If a simple process is decomposed into a specification module, the meaning of the process is the meaning of the decomposed module. On the other hand, if a process is not decomposed into a module, the meaning of the process is defined by its pre and postconditions. The Object-Z class *SimpProc* models simple processes in the same way as the previous semantic model.

The behaviour of a composite process, a newly introduced feature, is defined by a CDFD. A composite process in a class can be defined as:

CompProc

CoreProc

$beh : CDFD$

$within \in Class \wedge decomp \in Nil$

$\mathcal{M} \mathrel{\widehat=} beh.\mathcal{M};\ Flow$

Compared to the previous semantic models of processes [3], these semantic models are simple and compact due to the simplification of the input port to be a single port (rather than multiple ports in the previous language). Furthermore the new semantic models of processes are presented in an incremental and structured way, in contrust to the previous flat/lengthy semantic models.

2.2 SOFL Classes and Modules

The two grouping constructs, module and class, have common features, they both include a name, a group of user-defined types and an invariant. Firstly, we model their commonalities as:

CoreStructure

$name : Id$
$definedtypes : \mathbb{P}\, UserDefType$
$inv : Pred_{\textcircled{L}}$

$\forall\, v_1, v_2 : (\mathbf{local} \cap Var) \bullet$
$v_1.id = v_2.id \Rightarrow v_1 = v_2$

The essential features of a class can be defined as:

CoreClass

CoreStructure, *CompType*

$localms : \mathbb{P}\, Proc_{\textcircled{C}}$
$localsig : Id \pfun Type$
$methods : \mathbb{P}\, Proc$

$\#methods = \#\{m : methods \bullet m.id\}$

The subscript '$_{\textcircled{C}}$' appended to the type of the attribute *localms* ensures that every class contains its own set of local methods.

Now, *BaseClass* and *DerivedClass* can be defined by inheriting *CoreClass*:

BaseClass

CoreClass

$sig = localsig \wedge methods = localms$

DerivedClass

CoreClass

$parclass : Class_{\textcircled{L}}$

$sig = localsig \cup parclass.sig$
$methods = localms \cup parclass.methods$

In *DerivedClass*, *parclass* denotes the parent class of a derived class. The subscript '$_{\textcircled{L}}$' (shareable containment) appended to the type of *parclass* precisely specifies that the single inheritance structure is acyclic (i.e. a class cannot directly or indirectly inherit itself). The attribute *localms* (inherited from *CoreClass*) denotes the locally defined additional methods.

SOFL classes can be defined as

$$Class \mathrel{\widehat{=}} BaseClass \cup DerivedClass$$

A SOFL module consists of a name, an interface, a group of user-defined types, a group of state variables (which will be the 'read' or 'write' variables of its process), a module invariant, a group of processes and a CDFD. It can be defined by inheriting the *CoreStructure* and *Interface*.

Module

CoreStructure, *Interface*

[... similar to the previous semantics]

2.3 Semantics of CDFD

As SOFL's CDFD allows non-determinism, the semantics of a CDFD also have non-deterministic behaviour. CDFDs are modeled as:

CDFD

$nodes : \mathbb{P}(Proc \cup InEnv \cup OutEnv)$ [data-flow nodes]

$dfs : \mathbb{P}\ DataFlow$ [data-flow links]

Δ

$active\_nodes : \mathbb{P}(Proc \cup InEnv)$

$nodes \cap Proc \neq \varnothing \wedge nodes \cap InEnv \neq \varnothing$

$(\bigcup\{d : dfs \bullet d.\mathbf{local}\} \cap (Proc \cup InEnv \cup OutEnv)) = nodes$

$active\_nodes = \{c : nodes \mid c.active\}$

$Step \mathrel{\widehat{=}} \Box\ n : active\_nodes \bullet n.\mathcal{M}$

$Activate \mathrel{\widehat{=}}$
$\quad [(active\_nodes \cap Proc) = \varnothing] \bullet Step;\ Activate$
$\quad \Box\ [(active\_nodes \cap Proc) \neq \varnothing]$

$Process \mathrel{\widehat{=}}$
$\quad [(active\_nodes \cap Proc) \neq \varnothing] \bullet Step;\ Process$
$\quad \Box\ [(active\_nodes \cap Proc) = \varnothing]$

$\mathcal{M} \mathrel{\widehat{=}} Activate;\ Process$

A single trace step is to select an active-node and execute the node (captured by the operation *Step*). Initially, 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*).

The input and output environments are modeled as objects of the following classes:

InEnv

$right : Variable$
$outflow : DataFlow$
$active : \mathbb{B}$

$active = true$
$right = outflow.left$

$\mathcal{M} \mathrel{\widehat{=}} right.Assign;\ outflow.\mathcal{M}$

OutEnv

$left : Variable$
$inflow : DataFlow$

$\mathcal{M} \mathrel{\widehat{=}} left.OutVal$

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.

2.4 Data Flows

A data-flow in the original SOFL graphical notation can be a simple data-flow (with only one destination) or a complex data-flow connector. The SOFL graphical constructs are enriched mainly by introducing classes, the semantics of the data-flow is extended as follows:

$$DataFlow \mathrel{\widehat{=}} SimpleDF \cup DFConnector \cup FormObjDF \cup SpreadObjDF \cup RenameDF$$
$$DFConnector \mathrel{\widehat{=}} ConditionDF \cup CaseDF \cup BroadCastDF \cup ChoiceDF$$

where the *DFConnector* with its components defined in [3] are completely reusable in the current semantics and we omit them in this paper. The new object formation data flow and object spreading data flow are defined as:

FormObjDF

$in : \text{seq}\ Variable$ [input port]

$out : CompVal$ [output object]

$\#in = \#out.attris$

$\forall\, i : \text{dom}\ in \bullet in(i).type = out.attris(i).type$

$\mathcal{M} \mathrel{\widehat{=}} \wedge i : \text{dom}\ in \bullet$
$\quad in(i).OutVal \parallel out.attri(i).Assign$

SpreadObjDF

$FormObjDF[Out/in;\ in/Out]$

where *SpreadObjDF* is defined by inheriting *FormObjDF* with appropriate renaming, and exactly captures the similarity and difference between the two. The new renaming data flow is also defined similarly as:

RenameDF

in : seq *Variable* [input port]
out : seq *Variable* [output port]

$\#in = \#out.attris$
$\forall\, i : \mathrm{dom}\, in \bullet\ in(i).type = out(i).type$

$\mathcal{M} \mathrel{\widehat{=}} \wedge i : \mathrm{dom}\, in \bullet$
$in(i).OutVal \parallel out.attri(i).Assign$

This completes the semantics of the extended SOFL.

3 Conclusions

SOFL has been extended to an object-oriented formal language. This paper has presented an extension of the formal semantics for SOFL in Object-Z. The study of the formal semantics also helped us in the language design. For example the complex semantics representation of the input ports for a conditional process in our previous semantics model [3] motivates the simplification of the input ports in the current SOFL process definition.

In the previous SOFL semantics work [3], we predicted that if SOFL is extended, it's formal semantics will be readily reusable and extendible. This paper has confirmed this prediction even with substantial language extensions. A large part of the previous semantic model is reusable in the current SOFL semantics. For example, the semantic models for graphical data flow connectors can be directly included without any modification. Object-Z inheritance is exploited to illustrate the commonalities and the differences between various SOFL textual and graphical constructs and to present incremental and structured semantic models. The two semantic models (the previous one [3] and the current one) help readers, and case tool developers, pinpoint precisely the differences between the two SOFL versions.

Acknowledgements

We would like to thank Hugh Anderson and anonymous referees for many helpful comments. This work is supported by the research grants from JSPS Scientific Exchange Programme, the Ministry of Education of Japan and National University of Singapore.

References

[1] 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.

[2] S. Liu and J. S. Dong. Class and Module in SOFL. In Y. T. Yu and T.Y. Chen, editors, *Asia-Pacific Conference on Quality Software (APAQS'01)*. IEEE Press, December 2001.

[3] J. S. Dong and S. Liu. An Object Semantic Model of SOFL. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*, pages 189–208. Springer-Verlag, June 1999.

[4] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z.* Cornerstones of Computing. Macmillan, March 2000.

[5] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

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

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

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

[9] 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.

[10] W. K. Tan. A Semantic Model of A Small Typed Functional Language using Object-Z. In J. S. Dong, J. He, and M. Purvis, editors, *The 7th Asia-Pacific Software Engineering Conference (APSEC'00)*. IEEE Press, December 2000.

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