SUBTYPE AND INHERITANCE IN OBJECT ORIENTED HIGH LEVEL PETRI NETS

Marius Brezovan * **Eugen Ganea** *

* *University of Craiova, Software Engineering Department*

Abstract: It is well-known that both object-oriented paradigm and Petri net theory are two powerful frameworks used to specify complex systems, each of them having specific advantages. In recent years, several proposals tried to associate them into a single framework which combine the expressive power of both approaches. This paper presents a Petri net formalism called *Object Oriented High Level Petri Nets* (OOHLPN), and its connection with object-oriented methodologies. One important feature of the OOHLPN formalism is treatment of the inheritance and subtyping notions. OOHLPNs use distinct hierarchies for subtyping and subclassing, the connection between a type hierarchy and an associated inheritance hierarchy being a conformance relation.

Keywords: Petri nets, object-oriented Petri nets, system modelling, formal methods, algebraic specifications

1. INTRODUCTION

During recent years, several formalisms have been developed based on merging Petri net formalisms and the object-oriented paradigm, which preserve the advantages of both paradigms.

Battiston and De Cindio (Battiston *et al.*, 1988) include the algebraic data types into Petri net formalism. OBJSA does not include the notion of inheritance, but this issue is addressed in the derivative language CLOWN (Batiston *et al.*, 1996) where inheritance is supported by adding attributes, by redefinition of methods and by requiring an ST-preorder relationship between the corresponding state machine nets.

Another formalisms proposed by Sibertin-Blanc and Bastide (Bastide *et al.*, 1993) are Cooperative Objects, and their successor Cooperative Nets. A Cooperative Object posses a type, an identity and a state, while a class of consists of an interface which declares the services provided by the class, and an implementation which details how services are realized.

Biberstein (Biberstein and Buchs, 1995) proposed a formalism called CO-OPN that extends the use of algebraic data types to include the communication structures. An extended version of CO-OPN, called CO-OPN/2 (Biberstein *et al.*, 2001) adds new features, like a synchronization mechanism and the definition of sub-typing based on bisimulation.

Another approach is taken by Lakos and Keen in LOOPN (Lakos, 1994), which is a textual language based on the Object Petri Net formalism (OPN). OPN is an extension of Coloured Petri nets, representing a single unified class hierarchy which allows inheritance and polymorphism.

In this paper we define the Object Oriented High Level Petri Nets (OOHLPN) and their connections with object-oriented methodologies. The formalism of OOHLPN is based on the forthcoming ISO High-Level Petri Net (HLPN) standard (ISO/IEC, 2002) and on concepts from algebraic specifications (Ehrig and Mahr, 1985; Wirsing, 1990).

2. INTERFACES AND TYPES

In the OOHLPN formalism we make a difference between data types, which are non object-oriented types (either simple types as integers, strings, etc., or composed types as lists, etc.), and object types, which are specific to object-oriented applications.

Throughout this paper, we assume a universe U, which includes several disjoints sets:

$$U = SORT \cup OID \cup VAR \cup METH$$

where $SORT$ is the set of all sorts, OID is the set of all identifiers associated to objects, VAR is the set of variable names, and $METH$ is the set of public method names of object types. The set OID contains the constant $undef$, $undef \in OID$, but $undef$ cannot be associated to any object. The set of sorts, $SORT$, is partitioned in three disjoint sets, $SORT_D$, $SORT_{Ob}$ and $SORT_{Ref}$, corresponding to the names of non object-oriented data types, the names of object types and the names of reference types respectively.

It is widely recognized that subtyping inheritance are orthogonal mechanisms, concerned with the manipulation of types and implementations, respectively (Cook *et al.*, 1990; Taivalsaari, 1996). Nevertheless, a majority of the modern object-oriented programming languages and formalisms have opted for the unification of the notions of type and implementation into a single concept.

In the OOHLPN formalism we separate the notion of a class into its two distinct elements:

- *interface*, which is used to define an object type;
- *implementation module*, which is used to specify an object type implementation.

In order to specify the methods of objects, the set $METH$ is considered as a sorted set:

$$(METH_{\hat{c}w,s})_{\hat{c}\in SORT_{Ref},w\in SORT^*,s\in SORT\cup\{\epsilon\}}$$

For each method name $m \in METH_{\hat{c}w,s}$, $\hat{c}$ represents the reference type to the object containing the method, w is the sequence of sorts representing the input values, and s is the sort of the returned value, with the property that $ws \neq \varepsilon$ (a method must have at least an input value or a returned value).

In order to safely integrate the concepts of covariant or contravariant specialization of inherited methods, OOHLPN uses *encapsulated multi-methods* and a multi-dispatching mechanism for messages (Bruce *et al.*, 1996), (Castagna, 1995), instead of ordinary methods and a single dispatching mechanism. An *encapsulated multi-method* is an overloaded method associated with an object type, which contains several methods (branches) having the same name. We use some notions and definitions taken from (Bruce *et al.*, 1996), (Castagna, 1995), adjusted to our formalism.

Given an object type c, m is the name of an encapsulated multi-method of c containing n branches, if there exists $w_1, \ldots, w_n \in SORT^*$ and $s_1, \ldots, s_n \in SORT \cup \{\epsilon\}$ such that:

$$m \in \bigcap_{i=1}^{n} METH_{\hat{c}w_i,s_i}$$

In order to allow OOHLPN using a subtyping hierarchy, a type must be associated to each encapsulated multi-method. Let c be an object type and $m \in \bigcap_{i=1}^{n} METH_{\hat{c}w_i,s_i}$ an encapsulated multi-method name of c with n branches. The type of m can be denoted as

$$\{\hat{c}w_1 \rightarrow s_1, \ldots, \hat{c}w_n \rightarrow s_n\}$$

where $\hat{c}w_i \rightarrow s_i$ is the function type of the i^{th} branch of m.

The creation of objects can be viewed as calling a special method, *create*, of the corresponding object type. An object type may contain several *create* methods, but these methods do not represent an encapsulated multi-method. The main difference between a branch from an encapsulated multi-method and a *create* method concerns the moment when a called method is determined: in the case of multi-methods the dynamic (or late) binding is used, whereas in the case of *create* methods the static binding is used. From syntactical point of view, the *create* methods of an object type have a form similar to an encapsulated multi-method:

$$(create, t_{create})$$

where t_{create} represent the type of the set of methods.

Remark. There are no *destroy* methods in the OOHLPN formalism, because a garbage-collector mechanism is used.

In the OOHLPN formalism the weak form of substitutability principle is used because interfaces are defined at syntactic level. Also, OOHLPNs use the notion on named subtyping instead of structural subtyping because of its simplicity. In this case the subtype hierarchy is declared explicitly: an interface associated to a object type must declare all its supertypes.

Definition 2.1. Let c be an object type. An *interface*, $Intf$, defining c is a tuple

$$Intf = (c, \leq^S, Mt, Create)$$

where:

- c is the name of the interface;
- $\leq^S \subseteq \{(c, s) \mid s \in SORT_{Ob}\}$ is a partial order specifying the subtype relation associated to c;

- $Mt \subseteq MMETH(c)$ is a finite set of encapsulated multi-methods of c.
- $Create = (create, t_{create})$ is the set *create* methods of c.

The set of all interfaces is denoted by $INTF$.

The map $ObjType$ is used to associate to each interface its object type:

$$ObjType : INTF \rightarrow SORT_{Ob}$$
$$ObjType(Intf) = c, \text{ if } Intf = (c, \leq^S, Mt, Create)$$

The *subtype* relation is related to the notion of type and it relies on the substitution principle (Wegner and Zdonik, 1988).

In order to compare the sorts of function types the subtyping relation for function types is used. This relation require a contravariant rule (Cardelli, 1988). Let $u, v \in SORT^*$, and $s, t \in SORT$. If $v \leq u$ and $s \leq t$, then for the function type the following relation on sorts holds:

$$u \rightarrow s \leq v \rightarrow t$$

Subtyping relation for function types allows to define a similar subtyping relation for multi-methods.

A type of an encapsulated multi-method, $t_1 = \{\hat{c_1}w_1 \rightarrow s_1, \ldots, \hat{c_1}w_m \rightarrow s_k\}$, is a *subtype* of another encapsulated multi-method type, $t_2 = \{\hat{c_2}v_1 \rightarrow r_1, \ldots, \hat{c_2}v_n \rightarrow r_n\}$, if every encapsulated multi-method of type t_1 also has type t_2. In other words, $t_1 \leq t_2$ if for every function type $\hat{c_2}v_j \rightarrow r_j$ from t_2 there exists a function type $\hat{c_1}w_i \rightarrow s_i$ from t_1 such that:

$$\hat{c_1}w_i \rightarrow s_i \leq \hat{c_2}v_j \rightarrow r_j$$

The OOHLPN formalism uses the notion of deep subtyping (Castagna, 1995) and the weak principle of substitution to define the subtype relation between object types.

Definition 2.2. Let $Intf_1 = (c_1, \leq_1^S, Mt_1, Create_1)$ and $Intf_2 = (c_2, \leq_2^S, Mt_2, Create_2)$ be two interfaces defining the object types c_1 and c_2, with

$$Mt_1 = \{(a_1, p_1), \ldots, (a_n, p_n)\}$$
$$Mt_2 = \{(b_1, q_1), \ldots, (b_k, q_k)\} \ .$$

The object type c_2 is a *subtype* of c_1, or c_1 is a *supertype* of c_2, if the following relations hold:

(a) the subtype relation must be explicitly defined in the interface $Intf_2$: $(c_2, c_1) \in \leq_2^S$;
(b) the subtype t_2 has at least all the messages of the supertype t_1: $\{a_1, \ldots, a_n\} \subseteq \{b_1, \ldots, b_k\}$;
(c) multi-methods of t_2 may have subtypes of the corresponding multi-methods of t_1 with the same name: $a_i = b_j$ implies $p_i \leq q_j$.

Additionally, some restrictions on a set of interfaces have to be imposed in order to define a coherent set of object types. Let IT be a set of interfaces. The first requirement prevents two distinct interfaces to be associated with the same object type:

$$Intf_i, Intf_j \in IT \text{ with } i \neq j \text{ implies } ObjType(c_i) \neq ObjType(c_j)$$

Because OOHLPN don't use suctructural subtyping, the second requirement prevents an object type to be defined as a subtype of itself. Let $\leq_{IT}^S$ representing the set of all subtype relations from IT:

$$\leq_{IT}^S = \bigcup_{(c, \leq^S, Mt, Create) \in IT} \leq^S$$

The set $\leq_{IT}^S$ determines the subtype graph associated to IT, and the second requirement constraints this directed graph to be acyclic. A set of interfaces IT is *well-defined* if IT verifies the above two requirements.

A *root* of a well-defined set of interfaces IT is an interface, $Intf = (c, \leq^S, Mt, Create)$, with no ancestors, so that $\leq^S = \emptyset$.

3. MODULES AND IMPLEMENTATIONS

An *implementation* provides the realization of an object type behavior, and it defines a set (or class) of objects having the same internal structure and behavior. Because we separated the notions of type and implementation, we use the term *implementation module* instead of *class*. An implementation module $Impl$ implements an object type c if every instance of $Impl$ belongs to the object type c.

The implementation module of an object type is realized in the OOHLPN formalism by using a class of Petri nets called *Extended High-Level Petri Net with Objects* (EHLPNO), which represent high level Petri nets enriched with some object orientation concepts, such as creating new objects inside transitions when they fire, and calling public methods of objects inside transitions.

In OOHLPNs to each method or branch from an encapsulated multi-method corresponds a subnet contained in its associated Petri net, having an input place which stores the input values of the method, and an output place where the returned values are presented.

All input places of the branches oh an encapsulated multi-method (m, t) are connected to a unique input place of the multi-method denoted by $\#m$, and all output places are connected to a unique output place denoted by $m\#$. The input

place of m has no input arcs, ${}^\bullet\#m = \emptyset$, while its output place has no output arcs, $\#m^\bullet = \emptyset$.

In order to keep atomic actions inside transitions, two types of actions concerning method (or service) calls will be considered. Let t be a transition and $x \in Var(t)$ a variable associated to an object *ob*, which bound to its object identifier. A *send action* is a syntactical construction having the following form:

$$x.m(a_1, \ldots, a_n)$$

and a *retrieve action* is a syntactical construction having the following form:

$$b \leftarrow x.m$$

where m is the name of a method of the object *ob*, $a_1, \ldots, a_n \in TERM(O \cup V)$ are expressions containing input variables of t, and b is an output variable of t. The set of all method calls is denoted by MC.

A general assignment action of the form

$$b \leftarrow x.m(a_1, \ldots, a_n)$$

must be divided in two actions associated with two distinct transitions: a first transition containing a send action, an intermediate place waiting for the result, and a second transition containing a retrieve action, as presented in Figure 1.

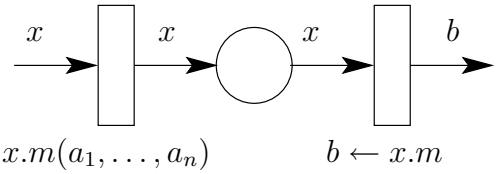

Fig. 1. Subnet representing an assignment action with a method call

In order to specify the creation of objects inside of a transition t, the following syntactical form representing an *instruction for creating objects* can be used:

$$\langle variable \rangle = new\ \langle sort \rangle (\langle params \rangle)$$

where $\langle variable \rangle \in VarOut(t) - VarIn(t)$ represents a variable associated to the newly created object, $\langle params \rangle$ is a list of expressions containing input variables of t, and $sort \in SORT_{Ob}$ is the object type of the created object. Because objects in the OOHLPN formalism are used through references, the variable associated with a creation instruction must have a reference type, $\langle variable \rangle \in SORT_{Ref}$.

The set of all object creation instructions is denoted by $CREATE$.

The creation of an object can be viewed as calling the special method, *create*, of the corresponding object type. The subnet associated to a *create* method plays an important role when a new object is created. Because the initial marking the places of the Petri net associated to an implementation module may contain references to some objects, the *create* methods must create the objects used in these initial markings. Unlike encapsulated multi-methods, each *create* method from an implementation module has its own input and output places.

Another action which can be associated to a transition is an assignment. Let t be a transition from a High-Level Petri net with an associated signature $Sig = (S, \leq, O)$ and a set, V, of variables. An *assignment action* for t is a syntactical construction having the following form:

$$v \leftarrow e$$

where $v \in VarOut(t)$ is a variable having a sort $s \in SORT_D$, and e is an expression with the same sort, $e \in TERM(O \cup V)_s$. The set of all assignments is denoted by ASS.

In order to define *Extended High-Level Petri Net with Objects*, each EHLPNO is associated with a signature, but some elements from an EHLPNO such as method calls and object creation can use sorts not locally defined in the signature. For this reason the profiles of method calls, object creation and inheritance relations are defined over a set of signatures and a set of class interfaces.

Definition 3.1. Let SG be a set of order-sorted signatures, IT a set of interfaces, $Sig \in SG$, $Sig = (S, \leq, O)$ a Boolean order-sorted signature, and $H = (S_H, \leq, O_H)$ an order-sorted Sig-algebra. An *Extended High-Level Petri Net with Objects* is a tuple:

$$Ehlpno = (NG, Sig, V, H, Sort, An, Ac, FP, m_0),$$

where:

(i) $NG = (P, T; F)$ is the net graph with;
(ii) V is an S-indexed set of variables, disjoint from O;
(iii) $Sort : P \rightarrow S$ is a function which assigns sorts to places;
(iv) $An = (a, TC)$ is the net annotation:
  - $a : F \rightarrow TERM(O \cup V)$ is a function that annotates each arc with a term of the same sort as that of the associated place;
  - $TC : T \rightarrow TERM(O \cup V)_{Bool}$ is a guard function that annotates each transition with a Boolean term expression;

(v) $Ac : T \rightarrow CREATE \cup MC \cup ASS \cup \{undef\}$ is the action net annotation;
(vi) $FP \in VAR^*$, $FP = \langle FP_1, \ldots, FP_n \rangle$, with $n \geq 0$, is a sequence of S-indexed variables, disjoint from V, such that if $n \geq 0$, then $\overline{FP} = \bigcup_{i=1}^{n} FP_i$ represents the set of for-

mal arguments used in the initial parametric marking m_0; if $n = 0$, then $\overline{FP} = \emptyset$.

(vii) $m_0 : P \to TERM(O \cup \overline{FP})$ such that $\forall p \in P$, $m_0(p) \in TERM(O \cup \overline{FP})_{Sort(p)}$ is the initial parametric marking having the same sort as the place.

The set of all Extended High-Level Petri Nets with Objects is denoted by $EHLPNO$.

The notion of the implementation module offers support for the definition of the inheritance concept. In the OOHLPN formalism only single inheritance relation is supported. In the following, $IMPL$ will denote the set of all implementation modules.

Definition 3.2. Let SG be a set of order-sorted signatures, $Sig \in SG$ a Boolean order-sorted signature, $Sig = (S, \leq, O)$, $H = (S_H, \leq, O_H)$ an order-sorted Sig-algebra, IT a set of well-defined interfaces, $Intf \in IT$ an interface, $Intf = (c, \leq^S, Mt, Create)$, defining the object type c. An *implementation module* of the interface $Intf$ is a pair:

$$Impl = (Ehlpno, Intf, Inh)$$

where:

(i) $Ehlpno = (NG, Sig, V, H, Sort, An, Ac, FP, m_0)$ is an Extended High-Level Petri Net with Objects;

(ii) $Inh \in IMPL \cup \{undef\}$ specify implementation inheritance relation of $Impl$.

Because methods in OOHLPNs are represented by subnets from EHLPNOs, the inheritance relation concerns incremental modifications of Extended High-Level Petri Nets with Objects. In order to specify the inclusion concept related to two EHLPNOs, the two order-sorted signatures associated to the two petri nets must define a hierarchical signature.

Definition 3.3. Let $\Sigma = (S, \leq, O)$ and $\Sigma' = (S', \leq', O')$ be two order-sorted signatures. The signatures Σ and Σ' define in this order a *hierarchical signature*, and it is denoted $\Sigma \leq_{Sig} \Sigma'$, if there exists an order-sorted signature morphism $\sigma : \Sigma \to \Sigma'$ such that $\sigma(\Sigma)$ is a subsignature of Σ'.

In the following definition of implementation inheritance, the notion of function restriction is used. Given a function, $f : A \to B$, and a subset $A' \subseteq A$, the restriction of f to A' is denoted by $f|_{A'}$.

Definition 3.4. Let IT be a set of well-defined interfaces, IM a set of well-defined modules that implement interfaces from IT, SG a set of order-sorted signatures used in the implementation modules of IM, and $Impl, Impl' \in IM$, $Impl = (Ehlpno, Intf, Inh)$ and $Impl' = (Ehlpno', Intf', Inh')$ two implementation modules, with:

$Ehlpno = (NG, Sig, V, H, Sort, An, Ac, FP, m_0)$
$Ehlpno' = (NG', Sig', V', H', Sort', An', Ac', FP', m_0')$

The implementation $Impl'$ *inherits* $Impl$ and it is denoted $Impl' \leq_I Impl$ iff the following relations hold:

(a) $Inh' = Intf$;
(b) $Sig \leq_{Sig} Sig'$;
(c) $P \subseteq P',\ T \subseteq T',\ F \subseteq F'$;
(d) $V \subseteq V'$;
(e) $Sort'|_P = Sort$;
(f) $a'|_F = a$, $TC'|_T = TC$, provided that $An = (a, TC)$ and $An' = (a', TC')$;
(g) $Ac'|_T = Ac$;
(h) $\overline{FP} \subseteq \overline{FP'}$, where $\overline{FP}$ represents the set constructed from the elements of the sequence FP, as in Definition 3.1;
(i) $m_0'|_P = m_0$.

4. OBJECT ORIENTED HIGH LEVEL PETRI NETS

The definition of OOHLPN uses the notions of subtype and inheritance hierarchy.

Definition 4.1. Let IT be a set of well-defined interfaces, IM a set of well-defined modules that implement interfaces from IT. An *object-oriented system* associated to IT and IM is a triple:

$$OS = (IT, IM, Inst)$$

where $Inst : IT \to \wp(OID)$ is a function which associates a set of object identifiers to each object type, such that if $Intf_i, Intf_j \in IT$, $ObjType(Intf_i) \neq ObjType(Intf_j)$, then $Inst(Inst_i) \cap Inst(Inst_j) = undef$, or $Inst(Inst_i) \subseteq Inst(Inst_j)$ if $ObjType(Inst_i) \leq ObjType(Inst_j)$.

Disregarding the constant $undef$, $Inst(Intf_i), i = 1, \ldots, n$ are disjoint sets in the case of unrelated implementation, in order to prevent two instances of unrelated object types to have the same object identifier. In order to prevent two instances of the same object type to have the same object identifier, the set of all object identifiers associated to a object-oriented system has to be structured as an algebra of object identifiers.

Definition 4.2. Let $OS = (IT, IM, Inst)$ be a object-oriented system as in the above definition. An *Object Oriented High Level Petri Net* associated to OS is a triple:

$$Oohlpn = (OS, Intf_0, oid_0)$$

where $Intf_0 \in IT$ is a root interface of the object-oriented system, called the initial object type of *Oohlpn*, and $oid_0 \in Inst(Intf_0)$ is the object identifier associated to the initial object of *Oohlpn*.

The initial interface $Intf_0$ of an OOHLPN is important when defining the dynamic semantics of these nets. It represents the higher level of abstraction for a modelled system, and its initial object is the unique instance of $Intf_0$, which exists at the beginning of the dynamic system evolution.

5. CONCLUSION

In this paper, we presented a new class of Petri nets, called Object Oriented High Level Petri Nets and their connections with object-oriented methodologies.

The OOHLPN formalism has been proposed for the need to encapsulate the object-oriented methodology into the Petri net formalism. The main features of the OOHLPN formalism are the followings:

- Unlike other approaches, the OOHLPN formalism is based on the forthcoming ISO High-Level Petri Net standard and it can be easily integrated in projects concerning concurrent and object-oriented systems;
- OOHLPNs preserve the property of Predicate/Transition nets to be viewed as the rule-based systems (Murata and Zhang, 1988). For a rule-based system a behaviorally equivalent OOHLPN can be determined. Because of this property OOHLPNs can also be used to specify rule-based systems;
- The OOHLPN formalism allows two types of relations: the inheritance relation, which is syntactically defined, and the subtyping relation, which is defined based on the substitutability principle.

REFERENCES

Bastide, R., C. Sibertin-Blanc and P. Palanque (1993). Cooperative objects: A concurrent, petri-net based, object-oriented language. In: *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*. pp. 286–292.

Batiston, E., A. Chizzoni and F. DeCindio (1996). Modeling a cooperative environment with clown. In: *Proc. of the 16th International Conference on Application and Theory of Petri Nets*. Workshop on Object-Oriented Programming and Models of Concurrency. pp. 12–24.

Battiston, E., F. DeCindio and G. Mauri (1988). Objsa nets: A class of high level nets having objects as domain. *Lecture Notes in Computer Science* **340**, 20–43.

Biberstein, O. and D. Buchs (1995). Structured algebraic nets with object-orientation. In: *Workshop on Object-Oriented Programming and Models of Concurrency'95*. pp. 131–145.

Biberstein, O., D. Buchs and N. Guelfi (2001). Object-oriented nets with algebraic specifications: The co-opn/2 formalism. *Lecture Notes in Computer Science* **2001**, 70–127.

Bruce, K., L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens and B. Pierce (1996). On binary methods. *Theory and Practice of Object Systems* **l(3)**, 221–242.

Cardelli, L. (1988). A semantics of multiple inheritance. *Information and Computation* **76**, 138–164.

Castagna, G. (1995). Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **17(3)**, 431–447.

Cook, W., W. Hill and P. Canning (1990). Inheritance is not subtyping. In: *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*. pp. 125–135.

Ehrig, H. and B. Mahr (1985). Fundamentals of algebraic specifications. *EATCS Monograph in Theoretical Computer Science*.

ISO/IEC (2002). *High-level Petri Nets - Concepts, Definition and Graphical Notation. Final Draft*. International Standard ISO/IEC 15909.

Lakos, C. A. (1994). Object petri nets, definition and relationship to colored nets. Technical Report TR94-3. University of Tasmania.

Murata, T. and Du Zhang (1988). A predicate-transition net model for parallel interpret ation of logic programs. Vol. 14(4). pp. 481–497.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys* **28(3)**, 438–479.

Wegner, P. and S. Zdonik (1988). Inheritance as an incremental modification mechanism or what like is and isn't like. *Lecture Notes in Computer Science* **322**, 55–77. Proceedings of the European Conference on Object-Oriented Programming (ECOOP).

Wirsing, M. (1990). Algebraic specification. In: *Handbook of Theoretical Computer Science* (J. van Leewen, Ed.). Vol. B: Formal Models and Semantics. pp. 675–788. The MIT Press.