INHERITANCE OF BEHAVIOR IN OBJECT-ORIENTED DESIGNS FOR INDUSTRIAL CONTROL SYSTEMS

M. Bonfè * **C. Fantuzzi** ** **C. Secchi** **

* *Dipartimento di Ingegneria - Università di Ferrara*
Via Saragat 1, Ferrara, Italy
** *DISMI - Università di Modena e Reggio Emilia*
Via Allegri 15, Reggio Emilia, Italy

Abstract: The paper presents a feasible approach to introduce object-oriented techniques in the industrial practice of control design. The approach is based on the use of a domain-specific extension of the modeling language UML and on the formalization of design models as transition systems for verification purposes. In particular, the paper shows how to exploit model checking techniques to verify that object classes, designed as subtypes, correctly inherit the behavior of their base classes, according to a notion of substitutability specifically defined for the proposed semantics of object-oriented models.

Keywords: Manufacturing systems, Logic controllers, Discrete-event systems, Verification.

1. INTRODUCTION

The emerging technologies for industrial control systems are putting more and more emphasis on concepts like modularity and reusability of components (both hardware and software), in order to increase efficiency of manufacturing systems design and reduce time spent during the installation of machines and the operational qualification of production lines. For example, the well-known standard for PLC (Programmable Logic Controllers) programming IEC 61131-3 (I.E.C., 2002) and the newer standard IEC 61499-1 (I.E.C., 2000) for distributed control systems, define frameworks for the implementation of modular software architectures, based on program organization units called Function Blocks (FBs). Even though the mentioned technologies incorporate many high-level concepts derived from the most recent Software Engineering principles, practical applications in the manufacturing domain of methods like object-oriented modeling or formal verification are quite rare. Several examples of academic projects that have attempted to fill this gap can be mentioned, for example the references in the review (Frey and Litz, 2000). Nevertheless, there are still important reasons limiting the appeal of formal methods from the point of view of industrial control designers, first of which the difficulties in the interpretation of some theoretical concepts within the peculiarities of the application domain and its day-to-day practice. These difficulties could be overcome adopting an "easy to use" modeling language and customizing it in order to include domain-specific aspects, making at the same time rigorousness of formal approaches transparent for technicians without a background on formal methods. With these remarks as a basic point, the paper presents a domain-specific adaptation and formalization of an object-oriented modeling language, namely UML (O.M.G., 2001), and how to formalize within this modeling framework the concept of *behavioral inheritance* between classes in a design model.

The rest of the paper describes in Section 2 the modeling language considered, in Section 3 its semantical formalization, with particular regard to inheritance of behavior, and in Section 4 the tools that could support designers for model verification. The paper ends with an illustrative example and some concluding remarks.

2. OBJECT-ORIENTED MODELING AND INDUSTRIAL CONTROLLERS

From a software engineering point of view, the features of IEC 61131-3 and IEC 61499-1 can be defined *Object-Based*, since FBs have many similiarities with *objects* as defined in modern programming languages: they are defined as types and used as instances, they encapsulate both private algorithms and data and they communicate with other software modules through well-defined *interfaces*, composed of *input* and *output* signals.Even though FBs allow to develop modular control software, neither of the IEC standards can be considered fully *Object-Oriented* (O-O) in a proper sense, because they do not include the feature of inheritance. However, this lackness should not prevent from the use of O-O design techniques, for example supported by modeling languages like UML and design tools with automatic code generators, provided that a proper interpretation and implementation-level mapping of abstract design concepts is defined. In fact, UML is defined by an extensible *meta-model*, which means that domain-specific concepts can be added to the language by means of *stereotyped* elements and well-formedness rules or constraints (i.e. expressed in *Object Constraint Language* (O.M.G., 2001)). The rest of the section will describe an extension of UML which can be adopted to design industrial control applications, focusing on a subset of the language that allows to completely specify *structure* and *behavior* of a system.

Structural views of an O-O system are described with UML by means of *Class Diagrams*, in which class symbols have compartments to show their properties (i.e. attributes and operations) and graphical links between them represent *simple association*, *aggregation/composition* (part-whole relationship) and *generalization* (a class *derives* from another), which involves inheritance. An important property that can be associated to a class is represented by its *stereotype*, which defines its conceptual role in a domain-specific model. For industrial control applications, it is important to specify structural aspects from a *mechatronic perspective*, which means that software modules must be considered in a tight aggregation with the physical sub-systems that they control. This aggregate, that we call *mechatronic object*, should have a signal-based interface, in order to exchange events with other mechatronic objects of a machine, and the description of its internal structure should highlight relationships with hardware components (i.e. sensors/actuators). The UML stereotypes that we have defined permit to describe classifiers for mechatronic objects as ≪`mechatronic`≫ classes, which have an interface of publicly visible properties, in their turn stereotyped as ≪`input`≫ or ≪`output`≫. A ≪`mechatronic`≫ class cannot have any public operations, while private operations may be used to model complex data-processing activities. The part of a ≪`mechatronic`≫ class related to the connection with physical components is specified with the help of classes stereotyped as ≪`hardware`≫. Classes of this kind are always related by means of a *composition* link with a ≪`mechatronic`≫ class and their ≪`input`≫ and ≪`output`≫ properties represent the hardware I/O ports as a private part of the ≪`mechatronic`≫ class. Figure 1 shows the graphical representation of the proposed stereotypes in a UML Class Diagram. It can be noted

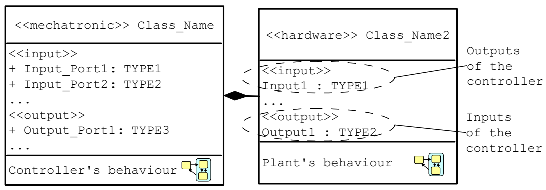

Fig. 1. UML stereotypes for mechatronic models

that mechatronic objects are quite similar to *actors* in the ROOM language (Selic *et al.*, 1994), which communicate with each others through the messages exchanged through well-defined *ports*, consistently with a given *protocol*. However, the concept of interface for mechatronic objects is simpler and more similar to the one proposed by IEC standards: each ≪`input`≫ or ≪`output`≫ signal represent a "port" of a given type (including `EVENT`). A complete structural model for a complex machine would be defined by one or more Class Diagrams, in which *part/whole* relationships between modules are modeled by *composition* links between mechatronic classes. The physical interpretation of objects in this framework suggests the definition of well-formedness rules prescribing that there must always be a single *"top-level"* class (i.e. the machine), that *shareable aggregation* links cannot be used, and that composition links must be qualified with fixed multiplicity, since dynamic creation of objects is not admissible.

The behavioral specification of the system must be specified describing the internal behavior of each class with a UML *State Diagram*, a kind of state model strictly derived from Harel's Statecharts (Harel, 1987). In the proposed UML extension, the Statechart of a ≪`mechatronic`≫ class represents the behavior of the controller, while the Statechart of a ≪`hardware`≫ class models the plant's behavior. With regard to textual expressions in Statecharts (i.e. transition labels and state actions), their specification with an IEC 61131-3 compatible syntax would ease automatic code generation for industrial controllers. In particular, we propose to label transitions with strings having the format:

```
trigger[guard]/actions
```

where events in the `trigger` can be inputs of the stereotyped class or outputs of its contained instances, explicitly typed as `EVENT`. The `guard` must also be a valid boolean expression, and `actions` will follow the same rules of the similar string included in a state action, which is specified by a textual expression like:

```
when / actions IF[guard]
```

Here, `when` is a qualifier that can be `entry` or `exit`, `guard` is an optional boolean expression that may prevent the action from being executed, if it evaluates to false, and `actions` is an ordered list of operations that can: *set* or *reset* a boolean variable, *assign* the value of an expression to a variable of a non-boolean data-type, or *emit* an attribute typed as `EVENT`.

3. FORMALIZATION OF MECHATRONIC MODELS

The underlying semantics of the modeling language described in previous section must be formalized taking into account the peculiarities of the application domain which make inadequate some parts of the UML semantics. In particular, the semantics of Statecharts is defined in UML by a *Run-To-Completion* execution algorithm, based on an event-queue for each object, in which events are processed one at a time. This interpretation is counter-intuitive for the implementation on synchronous devices like PLCs, for which the "original" Statecharts semantics of (Harel and Naamad, 1996) is more suitable. Moreover, the common O-O definitions of inheritance in terms of *structural conformity* (e.g. name consistency of public operations) are not appropriate for the domain of industrial control and for a design methodology in which components behavior is specified with state models. The correct interpretation of the inheritance concept must consider *behavioral conformity* and *substitutability* of state-based behaviors, with definitions similar to those reported in (Harel and Kupferman, 2002). Therefore, we define the instantiation of the *top-level* class in a UML mechatronic model as a **mechatronic system**

$$M_S = (M, t, \Gamma) \qquad (1)$$

where M is a set of instances of *mechatronic classes*, $t \in M$ is the top-level one and $\Gamma : M \rightarrow 2^M$ is a function that retrieves for each instance the ordered set of its components. The composition of M_S is univocally determined by multiplicity of aggregation links in the UML model and each $M_i \in M$ is an univocally referrable instance of a *mechatronic class* C_j. A **mechatronic class** is defined as

$$C = (S, T, P, r, \gamma) \qquad (2)$$

where S is a set of *states*, T is a set of *transitions*, $P = P^I \cup P^O$ is a set of "port" variables, each one of a given data type (including *event*), $r \in S$ is the root state and γ is an ordered set of contained instances of other mechatronic classes. The hierarchical aspects of the Statechart of a class C are defined by typing each $s \in S$ as an AND-state, OR-state or basic, and by the functions $def(s)$, which retrieves the default state of each OR-state, $chldn(s)$, which retrieves the set of immediate substates of s, and $chldn^\star(s)$, the reflexive-transitive closure of $chldn(s)$. A *configuration* is a subset of S which is maximally consistent (i.e. all of its elements can be simultanously active) and $compl(X)$ retrieves a configuration which is the *default completion* of a consistent set X. We also define as $\mathcal{B}$ the set of *boolean expressions* over variables in $P^I \cup \bigcup_{M_i \in \gamma} P_i^O$, as $\mathcal{A}$ the set of *assignments* over variables in $P^O \cup \bigcup_{M_i \in \gamma} P_i^I$ and as $\mathcal{E}$ the set of *event expressions* over variables $P^I \cup \bigcup_{M_i \in \gamma} P_i^O$, which consists of boolean expressions that contain only variables typed as *events*. These definitions permit to associate with each transition $tr \in T$ the following attributes: $src(tr) \in S$, the source state, $trig(tr) \in \mathcal{E}$, the trigger expression, $grd(tr) \in \mathcal{B}$, the guard expression, $act(tr) \in 2^{\mathcal{A}}$, a set of assignment actions, and $targ(tr) \in S$, the target state. The scope of a transition tr is the smallest OR-state containing both $src(tr)$ and $targ(tr)$, while $maxsrc(tr)$ is the unique child of the scope of tr such that $src(tr) \in chldn^\star(maxsrc(tr))$. According to these definitions, when tr is fired the state $maxsrc(tr)$ and all of its descendents $(chldn^\star(maxsrc(tr)))$ are de-activated, while $targ(tr)$ and the states in its default completion are activated. A transition is enabled if the predicate

$$en(tr) = in(src(tr)) \wedge trig(tr) \wedge grd(tr) \qquad (3)$$

is true ($in(src(tr))$ means that the source state is active), but is firable only if an additional predicate $conflict(tr)$ is false, which happens if no other transition with a priority higher than that of tr is enabled. The priority rules we have adopted enforce an explicit order, fixed at design time, between transitions with the same source state and give higher priority to transitions exiting states at a higher level in the hierarchy. To conclude, we assume that states $s \in S$ have an associated list of actions, each defined as a tuple (w, a, g), where

$w \in \{entry, exit\}$, $a \in 2^{\mathcal{A}}$ is a set of assignments and $g \in \mathcal{B}$ is a guard expression.

The reaction of a mechatronic class instance to external stimuli is defined as a **step**, in which the next state configuration and the next value of each variable are computed. Each instance in a mechatronic system M_S performs its step when it is marked as *active*, instead of *idle* by a *scheduling function*, whose formal detail are not specified here. The most simple scheduling function would cyclically mark *active* each $M_i \in M$ according to a fixed sequential order (i.e. the typical PLC scan cycle). Whatever is the scheduling function, we assume that input ports of an instance $M_i \in M$ typed as events retain their truth value until M_i becomes *active* and are immediately set false when M_i has terminated to compute its step, which means that no event can remain undetected. Each instance of a generic mechatronic class C is initialized in the configuration $Sc^0 = compl(r)$, with given initial assignments to variables in $P \cup \bigcup_{M_j \in \gamma} P_j$, and each one of its steps is performed as follows:

(1) compute the set of firable transitions $F_i = (tr \in T_i | en(tr) \wedge \neg conflict(tr))$;
(2) compute the next configuration $Sc'_i = compl((Sc_i - \bigcup_{tr \in F_i} chldn^\star(maxsrc(tr))) \cup \bigcup_{tr \in F_i} targ(tr))$;
(3) execute exit actions related to exited states, actions associated with each $tr \in F_i$ and entry actions related to entered states.

The execution of a step tranforms the *status* of an instance M_i from $\sigma_i = (Sc_i, \mathcal{V}_i)$ to $\sigma'_i = (Sc'_i, \mathcal{V}'_i)$, in which $\mathcal{V}_i$ and $\mathcal{V}'_i$ are current and next values of each variable in $P_i \cup \bigcup_{M_j \in \gamma_i} P_j$. The observable status of an instance is defined as ${}^{obs}\mathcal{V}_i$ and is the value of the variables in P_i. The **global status** of a mechatronic system M_S is given by $\sigma_G = (\sigma_1, .., \sigma_n)$, where n is the cardinality of M, and its behavior, given a scheduling function, is determined by the set $\mathcal{L}_{M_S}$ of all the possible finite and infinite sequences $\sigma_G^0, \sigma_G^1, \sigma_G^2, ...$, in which the change between σ_G^k and σ_G^{k+1} is determined by the step of one instance.

In order to formalize the concept of behavioral conformity between mechatronic classes, we follow the so-called *Liskov Substitution Principle* (Liskov, 1988), which states that a class can be considered a subtype of another one if the behavior of an object-oriented system, defined in terms of the base class, is not affected by the substitution of all the instances of the base class with instances of the derived class. In our interpretation, since the instances of a class can influence the global behavior of a mechatronic system only by means of their input/output ports, we analyze the computational sequences of the system focusing on the value of that kind of variables. Therefore, we define as the **observable behavior** of a mechatronic class C in a mechatronic system M_S, which contains r instances of C with indexes between l and $l + r$, the set $\mathcal{L}^C_{M_S}$ of all the sequences $\sigma_C^0, \sigma_C^1, \sigma_C^2, ...$ that can be extrapolated from $\mathcal{L}_{M_S}$, in which σ_C^i is composed by the observable status of all the instances of C, that is $({}^{obs}\mathcal{V}_l^i, ..., {}^{obs}\mathcal{V}_{l+r}^i)$. Consistently with the previous definitions, we can define that a class C_1 is **substitutable** with another class C_2 having the same interface (i.e. $P_1 = P_2$), if for any mechatronic system M_S, with the same scheduling function,

$$\mathcal{L}^{C1}_{M_S} \subseteq \mathcal{L}^{C2}_{M_S} \tag{4}$$

that is the observable behavior of C2 can extend the observable behavior of C1, without deleting any observable sequence.

4. CHECKING SUBSTITUTABILITY

The substitutability of mechatronic classes as defined in previous section can be checked with the help of specific tools supporting formal verification techniques for finite state systems. In particular, the Cadence version of the tool SMV (McMillan, 1999), originally developed at Carnegie Mellon University, adopts *Symbolic Model Checking* (McMillan, 1993) to verify refinement of components in modular transition systems. Here, we will briefly describe how to translate in the SMV language the behavioral specification of mechatronic classes and how to exploit the tool's feature for refinement verification as a way to prove their substitutability. The SMV language allows to describe a finite state system with constructs to declare modules and data-types, supports both boolean and integer arithmetic and has specific constructs to initialize state variables and to assign them the next value in a computational path. A mechatronic class can be translated as an SMV module as follows:

```
module Mech_Class(Active, I1, I2, O1, O2){
input Active, I1,I2 : boolean;
output O1, O2 : boolean;
Instance1 : Mech_Class1(..);
...
}
```

where `Active` is a boolean input set true according to the scheduling function, the other parameters represent the observable interface and `Instance1` is one of the contained instances of other modules. The Statechart specification will be tranlated encoding the hierarchy of states into variables with enumerated values:

```
Root : {State1, State2, ..., StateN};
SUBState1 : {State11, ..., State1N};
```

and evaluating the configuration and the set of enabled and actually firable transitions with predicates defined as follows:

```
INState1 := (Root = State1);
INState11 := INState1 & (SUBState1 = State11);
ENTrans1 := INStateXX & Trigger & Guard;
CONFLTrans1 := ENTrans2 |  ENTrans3 | ..;
FIRABLETrans1  := ENTrans1 & !CONFLTrans1;
```

Finally, initialization and execution of a step can be translated as follows:

```
init(Root) := State1;       -- default state
init(SUBState1) := State11; -- default state
default{ next(Root) := Root; -- no state change
         next(SUBState1) := SUBState1;
         next(O1) := O1;
         ...}
in case{ Active & FIRABLETrans1 : {
         next(Root):= State2;
         ...
         next(O1) := true;} -- set action
         Active & FIRABLETrans2 : {...}
        ...}
```

which states that if no transition is firable the status of the module remains unchanged (`default` statements), otherwise it is opportunely changed. An SMV program is completed by the declaration of a `main` module (i.e. the top-level) and by the specification of desired properties of the system, to be proved by model checking, expressed with CTL and LTL (Allen Emerson, 1996) temporal logics or in terms of *refinement maps*. In the latter case, SMV will explore the computational paths of the system to prove that the assignments to a given set of variables are compatible with those specified in a so-called *abstract layer*. In practice, SMV can prove that every possible behavior of a system *implementation* is also a possible behavior of the system *specification*. In our case, in order to check that two classes are substitutable, we have to define the base class as the implementation layer and the derived class as the abstract layer. This is translated in SMV as follows:

```
module main(){
I1, I2, .. : boolean;
O1, O2, .. : boolean;
C1 : BaseClass(1, I1, I2, .., O1, O2, ..);
layer derived : {
   C2 : DerivedClass(1, I1, I2, .., O1, O2, ..);}
}
```

Notice that the instances C1 and C2 are always *active*. When SMV opens a similar program, it automatically defines as properties to check formulas written as `Oi//derived`, where `Oi` is an output signal in both C1 and C2. Each one of these properties is verified if the values taken by `Oi` along any computational path of the instance C1 are compatible with those taken along the paths of the instance C2, declared in the layer `derived`. The inputs of both instances are assumed *free* variables, which means that are allowed to range over any possible value of their types. If these properties are all proved, then the observable behavior of any instance of the base class is contained in the observable behavior of any instance of the derived class, for any possible stimulus that they can receive, which proves substitutability of the base class with the derived class in any mechatronic system.

5. EXAMPLE

An example of a manufacturing machine quite common in the packaging industry is schematized in Fig. 2. This kind of machine is called *horizontal packer* and has the following processing principle: products are inserted in an horizontal "tube", which is made wrapping around the film and sealing it along the longitudinal direction, then the film is sealed transversally and cut, in order to release the packed product.

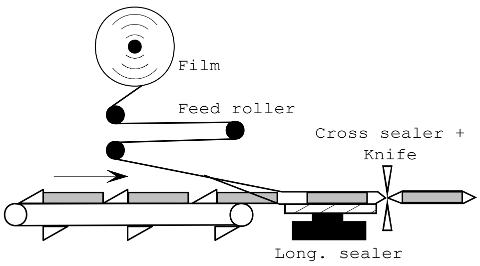

Fig. 2. Packaging machine with horizontal flow

A structural model of this machine, from the perspective of control design, can be described with UML as shown in Fig. 3. The diagram shows that the class `Longitudinal sealer` has a derived version named `Enhanced Longitudinal Sealer`, which has implicitly (because of structural inheritance) the same interface and contained instances (i.e. `Heater` and `Temperature Sensor`) of the base class, plus an additional instance of `Heater`, referred as `ExtraHeater` (name of the composition link).

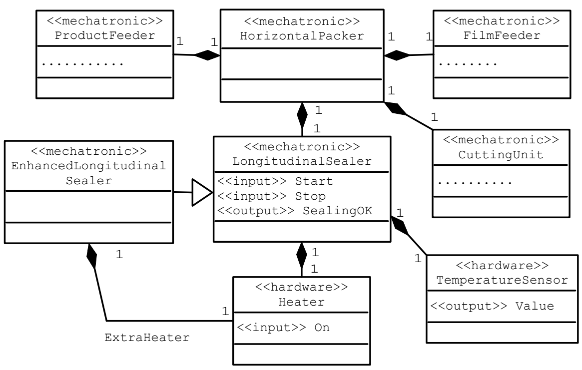

Fig. 3. Class Diagram of the horizontal packer

The behavior of `Enhanced Longitudinal Sealer` must be designed starting from the one inherited by the base class and modifying without breaking substitutability, following the heuristics suggested by some object-oriented methodologists, like those in (Douglass, 1999). In particular, Fig. 4 shows a possible (simplified) Statechart for the base behavior and Fig. 5 an extended Statechart in which the state `Increase Temp.` has been refined in order to include a substate in which the extra heater is powered, to speed up the rise of the measured temperature, and another in which it is switched off, with a transition between them that may be triggered, for example, by a timer.

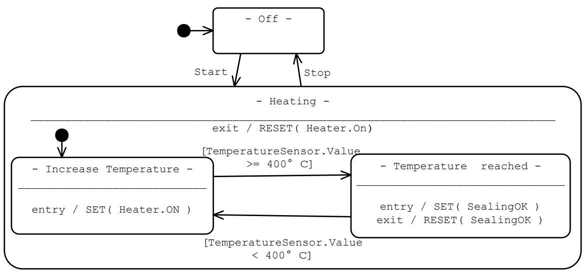

Fig. 4. Basic Statechart for a longitudinal sealer

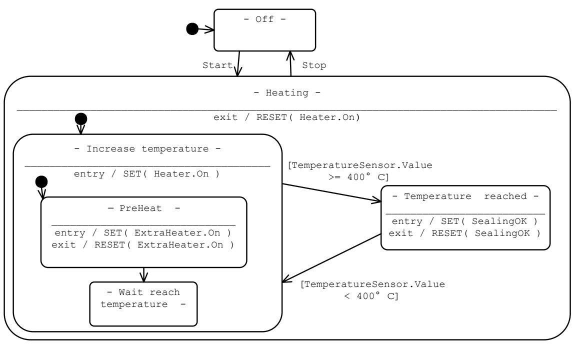

Fig. 5. Refined Statechart for a longitudinal sealer

This refinement do not change the behavior observed from the top-level class `Horizontal Packer`, which sends to the longitudinal sealer `Start` and `Stop` events and reads a high value on the boolean signal `SealingOK` when the measured temperature is above 400^o C. The behavioral conformity of the two classes can be verified writing an SMV program as described in previous section and, in case of positive answer from the model checking tool, the two Statecharts can be translated into the internal behavior of two software components (i.e. FBs) for PLC applications. Of course, both operations can be done automatically with the help of a CASE tool supporting UML and customizable code generation.

6. CONCLUSION AND FUTURE WORK

The paper has described a domain-specific extension of the modeling language UML which can be easily adopted by industrial control engineers to design programs for PLC-based systems. The concept of inheritance, characterizing the object-oriented approach to software and systems design, has been formalized in a definition specifically studied for the application domain. In the future, the definitions contained in the present paper will be extended to consider more complex cases of refinement (i.e. extension of the interface). Moreover, the authors aim to integrate the proposed concepts into a CASE tools that can support industrial control engineers in their design practice.

REFERENCES

Allen Emerson, E. (1996). Automated temporal reasoning about reactive systems. In: *Logics for Concurrency: Structure versus Automata* (F. Moller and G. Birtwistle, Eds.). pp. 111–120. Number 1043 In: *LNCS*. Springer–Verlag.

Douglass, B.P. (1999). *Doing Hard Time: developing Real–Time systems with UML, objects, frameworks, and patterns*. Addison Wesley Longman.

Frey, G. and L. Litz (2000). Formal methods in PLC programming. In: *Proc. IEEE Conf. on Systems Man and Cybernetics (SMC) 2000*. pp. 2431–2436.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8**, 231–274.

Harel, D. and A. Naamad (1996). The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodologies* **5:4**, 293–333.

Harel, D. and O. Kupferman (2002). On object systems and behavioral inheritance. *IEEE Trans. on Software Engineering* **28**(9), 889–903.

I.E.C. (2000). IEC 61499-1. Function Blocks for Industrial Process Measurement and Control - Part 1: Architecture. Public Available Specification (PAS).

I.E.C. (2002). IEC 61131-3. Programmable Controllers - Part 3: Programming Languages (2nd Edition). Final Draft International Standard (FDIS).

Liskov, B. (1988). Data abstraction and hierarchy. *ACM SIGPLAN Notices*.

McMillan, K.L. (1993). *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic Publishers.

McMillan, K.L. (1999). *The SMV language*. Cadence Berkeley Labs.. 2001 Addison St., Berkeley, USA.

O.M.G. (2001). UML, v.1.4, OMG specification. Document N. formal/2001-09-67. www.omg.org/uml.

Selic, B., G. Gullekson and P. Ward (1994). *Real-Time Object-Oriented Modeling*. John Wiley & Sons.