Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Verification of Fault Tolerance of Discrete-Event Object-Oriented Models using Model Checking

Marcello Bonfè * Cesare Fantuzzi ** Cristian Secchi **

* *ENDIF, Università di Ferrara, via Saragat 1, Ferrara, Italy*
** *DISMI, Università di Modena e Reggio Emilia, via Amendola 2, Reggio Emilia, Italy.*

Abstract: The Object-Oriented (O-O) approach has been recently used in the industrial automation to design logic control systems, thanks to the features of specification languages (e.g. UML) that can help to describe event-based behavioral requirements. In this paper, we aim to formalise an O-O framework for the design of modular logic controllers, in which faults occurring in the plant can alter the behavior of closed-loop system. Given the formal model of the system in terms of Kripke structures, it is possible to verify with model checking that even in case of faults the system do not violate given safety and liveness properties. Moreover, we will consider the case in which an O-O logic controller is refined applying the so-called "design-by-extension" mechanism, in which case it is important to verify that the fault tolerance property is inherited by the refined system.

1. INTRODUCTION

Object-Oriented (O-O) Modeling and Design methods are more and more used in several application domains, not only related to business software development, which is their original birthplace. Standard tools for modeling requirements and design specifications, like for example the graphical notations defined by the Unified Modeling Language (UML, O.M.G. (2005)), include many features that can be related to Discrete Event Systems (DES) Theory. For example, behavioral specifications for Object-Oriented Systems are often given by means of Statecharts (Harel (1987)), an enriched kind of hierarchical state machines, or Message Sequence Diagrams, a graphical representation of system traces. For these reasons, UML and similar languages have been recently used also in the industrial automation to design logic control systems. Translating UML design models into software code for Programmable Logic Controllers (PLCs) is not a difficult task with modern development tools (i.e. IEC 61131-3 languages and IEC 61499-1 Function Blocks). Morevoer, even if UML is not a formal language, in a proper sense, it is always possibile to formalize at least a subset of the language, with a domain-specific semantics, in order to apply formal verification methods, i.e. model checking and theorem proving (Clarke and Wing (1996)), and analyse the correctness of the control design against some given requirements specification.

The application of formal methods to logic control problems for industrial automation is a relevant research topic, as shown in the review of Frey and Litz (2000). The most recent results are related to fault diagnosis and fault tolerant design (Paoli and Lafortune (2005); Wen et al. (2007)). In this paper, we address the problem of fault tolerance by means of a domain-specific adaptation of the object-oriented modeling language UML and its formalization in order to apply model checking algorithms. Moreover, the concept of *inheritance* between classes, which is central in object-oriented design, is studied in detail and formalised at the behavioral level. By means of inheritance, it is possible to apply *design by extension* mechanisms, which means that new classes of objects are defined as refinements of existing ones. With this approach, we aim to address also the problem of fault tolerance, considering it as a behavioral property that must be preserved (i.e. inherited) after refinement.

2. BACKGROUND ON FAULT TOLERANCE

The concepts of dependability for automated and computing systems have been investigated since the very first generation of calculating systems (e.g. Babbage's Engine) (Avizienis et al. (2001)). Dependability is even more critical when humans may be directly threatened by incorrect services of computing systems, as is commonly the case in industrial automation and embedded control. In these applications, which constitute the so-called *mechatronic domain*, a *fault* (i.e. change in a system that may cause an *error* and, consequently, a *failure*) can rise up from electronics hardware, from software (i.e. "bugs") or from the controlled plant (e.g. a mechanical damage). Whatever is the nature of a sub-system considered, the design of fault diagnosis or fault tolerance techniques is mostly based on redundancy, either physical or analytical, of operating parts and some kind of (reliable) supervisor. For example, hardware or software for safety-critical systems often includes design patterns like watchdog timing, multiple computation with final voting, etc. (Douglass (1999)), while mechanical systems may have dual actuators, parallel structured modules or redundant manipulators (Sreevijayan et al. (1994)). The level of fault tolerance achieved with these solutions is generally classified as *fail-safe*, *nonmasking* or *masking*, on the basis of the preservation of safety and/or liveness properties.

In mechatronics, analysis and design tools based on DES models are very useful, especially considering diagnosis and faults handling. In fact, DES allow to represent quite complex systems by abstracting their main operating modes and driving stimuli. Faults can be represented as (uncontrollable and unpredictable) events, wherever they may be generated. Moreover, DES tools are quite adequate to design and verify logic control systems (i.e. PLC programs), whose primary function in industrial automation is actually to detect faulty situations and to guarantee safety of plants and human operators. As said before, fault tolerant design or fault tolerance verification has been addressed in a DES framework by many authors (Dal Cin (1997); Wen et al. (2007)), but the impact of these works on the industrial community is currently quite limited. On the other hand, O-O methods are instead quite appreciated by control engineers, thanks to the advantages they provide in terms of modularity and reusability of design patterns and software implementation. Fault tolerance can be easily incorporated in an O-O design framework exploiting concepts like incremental design, refinement and inheritance, as shown for example by Johnsen et al. (2001). In fact, fault tolerance in either DES or O-O models is generally based on some definition of bisimulation between faulty and non-faulty systems. Similar notions of refinement (see Harel and Kupferman (2002)) can be defined to formalize the concept of inheritance between classes in O-O languages. Therefore, in the following sections we adopt as a unifying approach the O-O one, describing first a modeling language based on UML, whose application is primarily thought as logic control design for industrial automation, then we give a formal definition of inheritance, with the aim to introduce model checking as a means to verify behavioral conformity between classes inheriting from others and checking their tolerance to faults.

3. OBJECT-ORIENTED MODELING AND INDUSTRIAL CONTROLLERS

The use of UML for logic control design is not a novelty (see for example Thramboulidis (2004). Here, we briefly recall the UML extension proposed by Bonfé and Fantuzzi (2004), which has been successfully applied in several industrial projects to design complex software programs for PLCs. The language is tailored to ease code generation for IEC 61131-3 and IEC 61499-1 targets, therefore structural specifications, described by UML *Class Diagrams*, emphasize class interfaces defined in terms of *Input/Output signal ports*, eventually distinguished into *data* I/Os (the only supported by IEC 61131-3) and *events* I/Os (supported by IEC 61499-1), while behavioral specifications are given by UML *State Diagrams*, whose textual expressions are made syntactically compatible with IEC 61131-3 languages.

The extension mechanism that allows UML to include domain-specific concepts is called *stereotyping*: the elements of UML's *meta-model*, (i.e. classes, relations, etc.) can be stereotyped by addying properties and constraints, which are peculiar of a given application domain. A consistent set of stereotypes is called a *profile*. In order to define an industrial control profile, 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, making this aggregate a *mechatronic object*. The UML stereotypes that we have defined permit to describe classifiers for mechatronic objects as ≪`mechatronic`≫ classes, which have an interface of public properties, in their turn stereotyped as ≪`input`≫ or ≪`output`≫. 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.

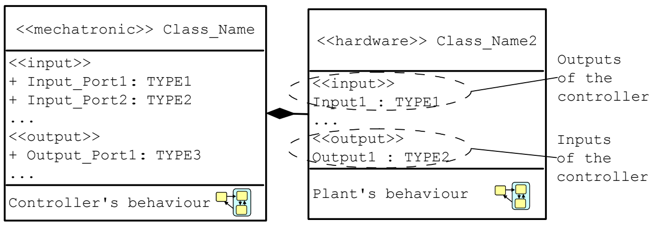

Fig. 1. UML stereotypes for mechatronic models

The behavioral specification of a mechatronic system is described by associating each class with a UML *State Diagram* (derived from Statecharts of Harel (1987)). Of course, the statechart of a ≪`mechatronic`≫ class represents the behavior of the controller, while the statechart of a ≪`hardware`≫ class models the plant's behavior. In the proposed UML extension, transitions are labeled 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:

`entry` (or `exit`) `/ actions [guard]`

where `guard` is an optional boolean expression that must be verified to enable actions' execution, 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 event.

With regard to inheritance, we assume that a class that is related to another one by a UML *generalization* link inherits the statechart of the base class. This statechart may be refined by design with the addition of transitions or elaborating substates in inherited states, but inherited states or transitions should not be deleted, in order to preserve behavioral substitutability between the base class and the derived one.

4. FORMALIZATION OF MECHATRONIC MODELS

The semantics of the modeling language described in previous section has been formalized taking into account the peculiarities of the application domain, which make the (informal) UML semantics inadequate. In particular, O.M.G. (2005) describes a *Run-To-Completion* execution algorithm for Statecharts, based on an event-queue for each object, in which events are processed one at a time. This interpretation is not consistent with both IEC 61131-3 and IEC 61499-1, which instead allow simultaneous management of multiple events. Moreover, we aim to include a definition of inheritance in terms of *behavioral conformity* and *substitutability* of state-based behaviors, with an approach similar to that of Harel and Kupferman (2002).

For this aim, we define the instantiation of the *top-level* class in a UML mechatronic model as a **mechatronic system**

$$M_S = (M, t, \Gamma) \tag{1}$$

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

$$C = (S, T, P, r, \gamma) \tag{2}$$

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. Here, we do not introduce distinctions between ≪`mechatronic`≫ or ≪`hardware`≫ classes. In next sections, hardware components will be assumed the only ones prone to faults and nondeterministic behavior.

To define the hierarchical structure of a statechart, we assume that each $s \in S$ is typed as an AND-state, OR-state or basic, and that $def(s)$ and $chldn(s)$ are functions retrieving, respectively, the default state of each OR-state and the set of immediate substates of s, while $chldn^\star(s)$ is 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. In order to formalize the set of states which are exited when a transition tr is fired, we define $scope(tr)$ as the smallest OR-state containing both $src(tr)$ and $targ(tr)$, $maxsrc(tr)$ as the unique child of $scope(tr)$ such that $src(tr) \in chldn^\star(maxsrc(tr))$. In this way, 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

$$en(tr) = in(src(tr)) \wedge trig(tr) \wedge grd(tr) \tag{3}$$

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. We assume higher priority for transitions exiting higher level states and explicit ordering (fixed by design) of transitions with the same source state. 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 given *scheduling function*. 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). In any case, 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. 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 composed by the values of variables in P_i. The **global status** of a mechatronic system M_S is given by:

$$\sigma_G = (\sigma_1, .., \sigma_n) \tag{4}$$

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 sequences

$$\sigma_G^0, \sigma_G^1, \sigma_G^2, ... \tag{5}$$

in which the change between σ_G^k and σ_G^{k+1} is determined by the step of one of the instance in M.

In order to formalize the concept of behavioral conformity between mechatronic classes, we must consider their behavior within a mechatronic system that contains instances of those classes. In fact, according to the *Liskov*

Substitution Principle Liskov (1988), one should be allowed to say that a class is 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. Since the instances of a class can influence the global behavior of a mechatronic system only by means of their input/output ports, it is necessary to analyze the traces of the system focusing on 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

$$\sigma^0_C, \sigma^1_C, \sigma^2_C, ... \tag{6}$$

that can be extrapolated from $\mathcal{L}_{M_S}$, in which σ^i_C is composed by the observable status of all the instances of C, that is $({}^{obs}\mathcal{V}^i_l, ..., {}^{obs}\mathcal{V}^i_{l+r})$.

It is now possible to give the following:

Definition: A mechatronic class C_1 is **substitutable** with another class C_2 having a compatible external interface (i.e. $P_1 \leq P_2$), if for any mechatronic system M_S, with the same scheduling function,

$$\mathcal{L}^{C1}_{M_S} \subseteq \mathcal{L}^{C2}_{M_S} \tag{7}$$

that is the observable behavior of C_2 can extend the observable behavior of C_1, without deleting any observable sequence.

This definition is related to inheritance as described in Section 3: the refinement of an inherited statechart must always be checked against behavioral substitutability, by using formal verification tools.

5. TOOLS FOR CHECKING SUBSTITUTABILITY

Formal verification can be used to analyze the substitutability of mechatronic classes, thanks to specific tools like Cadence SMV (CaSMV, McMillan (1999)), that adopts *Symbolic Model Checking* (McMillan (1993)) to verify refinement of components in modular transition systems. In fact, the underlying formal model which is actually model checked (i.e. the Kripke structure), is described in CaSMV using a high-level textual language supporting modularity, reusability, boolean and integer arithmetic, so that automatic code generation from UML tools can be easily implemented. In this section, we describe how to translate in the input language of the CaSMV tool the behavioral specification of mechatronic classes, in order to exploit the tool's feature for refinement verification as a way to test their behavioral conformity. A mechatronic class can be translated as a CaSMV 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 translated 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 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 changes accordingly. As mentioned before, the difference between a ≪`mechatronic`≫ and a ≪`hardware`≫ class of the UML model, is that the latter is allowed for nondeterminism, related to faulty behavior. In CaSMV, nondeterministic assignments are written as follows: `next(Var) := {Value1, Value2}`, which means that `Var` can take either `Value1` or `Value2`. Moreover, we assume that the `Active` input of hardware components is always true.

A CaSMV 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, expressed with CTL or LTL temporal logics (Allen Emerson (1996)) or in terms of *refinement maps*. In the latter case, CaSMV will explore the computational paths of the system to check that the assignments to a given set of variables are compatible with those specified in a so-called *abstract layer*. In practice, CaSMV can verify that every possible behavior of a model representing a system's *implementation* is also a possible behavior of a model representing the system's *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 specification layer, since we admit that the derived class can extend the behavior of the base class, but not restrict it. This is translated in CaSMV as follows:

```
module main(){
```

```
I1, I2, .. : boolean;
O1, O2, .. : boolean;

C1 : Base(1, I1, I2, .., O1, O2); --Active = 1

layer derived : {
   C2 : Derived(1, I1, I2, .., O1, O2);}
}
```

When CaSMV 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 they are allowed to range over any possible value of their types. If these properties are all verified, 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 external stimulus that they can receive, which ensures substitutability of the base class with the derived class in any mechatronic system.

6. EXAMPLE

In this section, we present an example to illustrate the conceptual approach. Let us consider a generic parts loading system for a manufacturing machine, whose main actuator is a pneumatic cylinder with a dual solenoid valve and two endstroke sensors. This kind of actuators is typically prone to mechanical faults, due to dirt and other environmental conditions, in which case they may get stuck in a given position.

In order to develop a full UML model of the mechatronic system, it is first necessary to specify a model of the basic components of the plant. In this case, we can represent the behavior of the pneumatic cylinder considered, in response to control system's command on the solenoids, with the simple statechart shown in Fig. 2. In order to keep the picture clear, several textual labels have been omitted. As can be seen, we assume that if the cylinder get stuck, it is always possible to return to the initial state. This transition may, for example, be fired because a human action that forces the cylinder to move. Of course, the model admits only one direction for this manual movement. In any case, nondeterminism is due only to the transitions exiting from "automatic" movement states (i.e. `Moving1` and `Moving2`).

The logic controller for this kind of actuators, that will be designed as a ≪`mechatronic`≫ class in UML, should include a fault detection mechanism based on a watchdog timer: if the cylinder is actuated in one direction, it must reach the endstroke before the watchdog timer elapse, otherwise the controller will raise an alarm and stop the operation. If this happens, human operators are required to manually inspect the status of the pneumatic system, which in case of a parts loading mechanism may get stuck also because of parts jamming, remove the cause of cylinder's fault, reset the alarm on the control system and restart the machine.

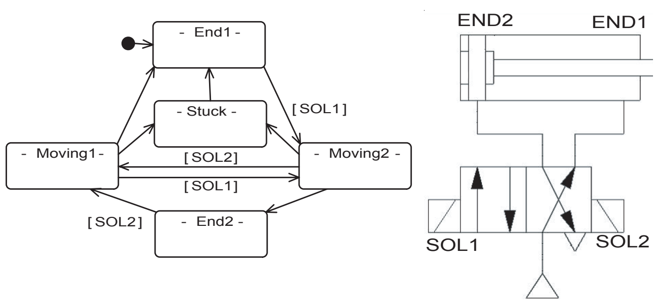

Fig. 2. Model of a pneumatic cylinder

Watchdog timing can be considered as a design pattern which allows to reach fail-safe fault tolerance. In fact, the watchdog timer allows the logic controller to maintain safety properties, but not liveness: if manual reset is not considered, the system is deadlocked and cannot perform any other (automatic) operation. However, watchdog timer must be correctly managed by the logic controller, in order to avoid false alarms and to deadlock the system in a safe state. This means that the timer must be activated only when really necessary, at the beginning of potentially dangerous operations, and reset when the operations are completed without errors.

A schematic statechart model of the logic controller for the mechatronic system considered is shown in Fig. 3.. For the same reason mentioned before, several textual labels have been omitted. We assume that the mechatronic class has an external interface composed by the input events `Start, Stop, LoadPart, AlarmReset` and the output events `Operating, PartLoaded, Alarm`.

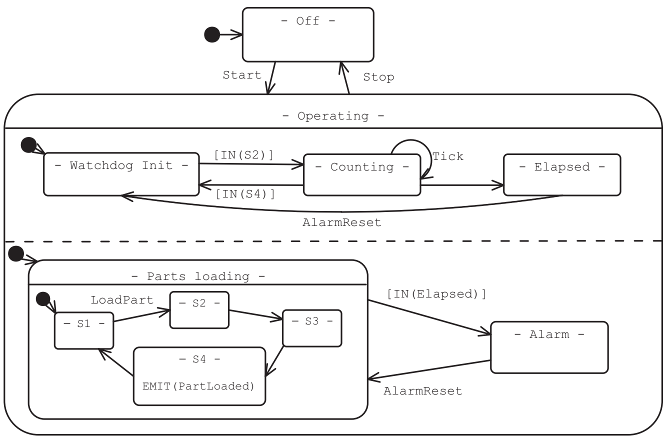

Fig. 3. Logic control of the loading mechanism

According to the graphical syntax of UML State Diagrams, the design model adopts concurrency (AND-states) within the `Operating` state, to manage the watchdog timer and the operations of the pneumatic cylinder. The timer must be enabled to count when the system starts a part loading operations and actuates a command on the cylinder. Therefore, the transition from `WatchdogReset` to `Counting` is fired by the condition `IN(S2)` (true when `S2` is) active). The timer increments at each `Tick` event (we assume that there is a system's clock provinding this event) and if it is not reset before a given deadline, an alarm is raised. If this happens, human operators are required to acknowledge the alarm, manually remove the cause of fault and reset the alarm, a common procedure in industrial automation.

In order to verify fault tolerance, it is necessary to guarantee that whenever the pneumatic cylinder gets stuck, the controller raises the alarm, therefore the following CTL formula must be verified with CaSMV:

```
AG (INStuck -> AF (INAlarm))
```

since `AG` means "for all paths, globally", while `AF` means "for all paths, in a future state". Similarly, it is required that the deadlock states can always be exited thanks to the manual recovery procedure described before, checking the formula:

```
!EG (AG (INAlarm))
```

in which `EG` means "exists a path, (in which) globally".

If the the class modeling the control logic is refined, by inheriting the base statechart and modifying it, for example in the way shown in Figure 4, the refinement verification procedure described in previous section must be applied in order to guarantee:

- first, that the behavior of the derived class is compatible with the behavior of the base class;
- then, that fault tolerance is inherited by the derived class.

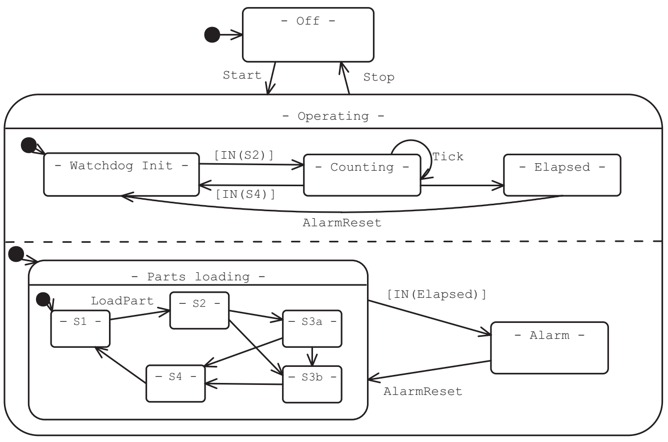

Fig. 4. Refined control logic of the inherited statechart

7. 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 mechatronic 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. The model checking tool CaSMV has been adopted in the design framework to verify both behavioral substitutability between classes in a design model and fault tolerance of the mechatronic system. In the future, we aim to address more general kind of fault tolerance (i.e. masking fault tolerance), with fully developed cases of study taken from real industrial problems.

REFERENCES

E. Allen Emerson. Automated temporal reasoning about reactive systems. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, number 1043 in LNCS, pages 111–120. Springer–Verlag, 1996.

A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report N01145, LAAS-CNRS, April 2001.

M. Bonfé and C. Fantuzzi. An application of object-oriented modeling tools to design the logic control system of a packaging machine. In *Proc. IEEE Int. Conf. on Industrial Informatics (INDIN'04)*, Berlin, Germany, June 2004.

E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Also available as Carnegie Mellon University's Technical Report CMU-CS-96-178.

M. Dal Cin. Verifying fault-tolerant behavior of state machines. In *Proc. Second IEEE High-Assurance Systems Engineering HASE 97*, Bethesda, Maryland, August 1997.

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

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

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

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

E.B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental fault-tolerant design in an object-oriented setting. In *Proc. 2nd Asian Pacific Conference on Quality Software, APAQS*, pages 223–230, 2001.

B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), May 1988.

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

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

O.M.G. UML, v. 2.0, superstructure specification. Document N. formal/05-07-04, 2005. www.omg.org/uml.

A. Paoli and S. Lafortune. Safe diagnosability for fault-tolerant supervision of discrete-event systems. *Automatica*, 41:1335–1347, 2005.

D. Sreevijayan, S. Tosunoglu, and D. Tesar. Architectures for fault-tolerant mechanical systems. In *Proc. of 7th Mediterranean Electrotechnical Conference*, volume 3, pages 1029–1033, Antalya, Turkey, April 1994. IEEE.

K. Thramboulidis. Using UML in control and automation: a model driven approach. In *Proc. of INDIN'04*, Berlin, Germany, June 2004.

Q. Wen, R. Kumar, J. Huang, and H. Liu. Fault-tolerant supervisory control of discrete event systems: Formulation and existence results. In *Proc. of Dependable Control of Discrete Systems*, Paris, France, June 2007.