Behavioral inheritance in object-oriented models for mechatronic system

Marcello Bonfè
Department of Engineering,
University of Ferrara, 44100 Ferrara, Italy
E-mail: marcello.bonfe@unife.it

Cesare Fantuzzi* and Cristian Secchi
Department of Science and Methods for Engineering,
University of Modena and Reggio Emilia,
41100 Reggio Emilia, Italy
E-mail: {fantuzzi.cesare,secchi.cristian}@unimore.it
*Corresponding author

Abstract:
The paper describes a formal framework for object-oriented modeling of mechatronic systems, whose main contribution is on one hand to unify the modeling approaches for dynamical systems and for industrial control software and, on the other hand, to provide a definition of inheritance, a cardinal concept in object-orientation, that emphasize the behavioral conformity of base and derived classes of objects. The proposed framework exploits the coalgebraic description of software artifacts to provide a connection between the behavioral approach for modeling dynamical systems and the object-oriented approach for software modeling and design. In particular, our definition of inheritance aims to allow control engineers to apply the design by extension methodology, widely used in software engineering, to the development of mechatronic components for manufacturing systems.

Keywords: Object-Oriented Modeling, Behavioral approach, Category Theory, Mechatronics, Industrial Automation

Reference to this paper should be made as follows: Bonfè, M., Fantuzzi, C. and Secchi, C. ‘Behavioral inheritance in object-oriented models for mechatronic system’, Int. J. Manufacturing Research, Vol. x, No. x, pp.xxx–xxx.

Biographical Notes: Marcello Bonfè received his Ph.D. in Information Engineering in 2003 from the University of Modena and Reggio Emilia. He is currently Assistant Professor in Automatic Control at the Department of Engineering of University of Ferrara. His main research areas are formal methods for manufacturing systems control, automatic fault diagnosis, modeling and control of mechatronic systems.

Cesare Fantuzzi is a ......

Cristian Secchi is ...
1 Introduction

The object-oriented (o-o) approach is renowned as a driving idea for many modern software design methods, supported not only by several programming languages of common use, but also by modeling and specification techniques based on graphical notations, of which the Unified Modeling Language (UML) [17] is the most well-known. The appealing reasons for the success of object-orientation in the software domain are related to the benefits that it guarantees in terms of modularity of programs and reusability of code. In particular, o-o methods allow to design by extension, which means to define new software components on the basis of previously developed ones, either by simply aggregating basic heterogeneous blocks or by adding new (specific) functionalities to a more generic component. In the latter case, the o-o feature that comes in hand is called inheritance. Thanks to the inheritance mechanism, given a basic class of software objects, characterized by a set of attributes (data) and methods (operations), it is possible to build a new class which inherits all the characteristics (i.e. attributes and methods) of the basic one and which can be further endowed with additional attributes and methods. However, while inheritance, as supported by o-o programming languages, is useful for code reuse, it may also become an issue in terms of behavioral conformity between basic and derived classes. Theoretically, the Liskov Substitution Principle (LSP) [13] states that an object of a derived class can be asked to do anything an object of the basic class can do, which means conformity of functional interface, but also that it should do it in the very same way and with the same results, which is of course a stronger requirement.

If we consider object-orientation as a conceptual approach to network-based modeling of dynamical and physical systems, as supported by languages like Bond-Graphs [19, 10], Modelica [25] and the extension of UML defined in [2, 22], the advantage of inheritance is evident: a complex system composed of a network of objects exchanging physical information (i.e. energy) can be extended by replacing one of its parts with a derived component which is able to realize the basic behavior, but, in addition, also to perform further functionalities. Clearly, in this domain it is necessary to define inheritance in a way that allows to formally fulfill the LSP or, in other words, that guarantees behavioral conformity between basic and derived classes of dynamical systems.

This paper presents an approach to formalize inheritance in an unified o-o framework for modeling and specification of manufacturing systems, in which both control software and physical subsystems contribute to the overall behavior. The major contribution of the proposed approach consists of allowing designers of mechatronic systems to adopt the principle of design by extension, so that they can formally specify new mechatronic components with additional features and use them to replace parts of a more complex system, while preserving the possibility of generating the same global behavior. In this way, all the tasks that can be performed by the overall original system can still be done exactly in the same way using the modified system but, thanks to the new features introduced in the new components, it is
also possible to implement some new and/or improve some of the existing working processes. The framework presented in the paper combines the coalgebraic view of object-orientation presented in [9], where category theory [14] is exploited to build a formal foundation for the object-oriented paradigm in software programming and to formally define the concept of inheritance, and the behavioral approach [27, 20]. The paper is organized as follows: in Sec. 2 some background on the use of UML for modeling mechatronic systems, on the behavioral approach and on category theory is given. In Sec. 3 and in Sec. 4 we describe the proposed o-o approach to Dynamical Systems modeling and we define the concept of inheritance for this domain. In Sec. 5 we propose two examples taken from industrial automation to validate the proposed framework. The paper ends with some ideas for future enhancements and applications of the proposed modeling framework.

2 Background

2.1 The Unified Modeling Language

The object-oriented approach to software design, successfully applied for long time in the “business” domain to develop complex systems based on reusable and reliable components, is now common also in many other fields, including those of real-time systems and control applications. This spreading has been certainly helped by software modeling methods based on domain-independent specification languages, which are essential tools to focus on conceptual design solutions, rather than implementation details (necessarily domain-dependent). Currently, the dominant modeling language for o-o software is UML [17], which is supported by several commercially available development tools (generating code for the most common o-o programming languages) and has been applied also for real-time and safety critical systems design [3]. More recent are instead the applications of UML and o-o principles to the design of control software for manufacturing systems, mainly because of the limitations of programming languages for industrial control devices. Nevertheless, it is worth to mention that the tools for modular software development (i.e. Function Blocks) described in the IEC 61131-3 [8] standard for PLC (Programmable Logic Controllers) programming and in the IEC 61499-1 [7] for distributed control systems have motivated the research works described in [1, 24, 6, 11], all of which present different methodologies for industrial software design based on UML.

Using UML, a (software) system can be described by means of various kind of diagrams, which permit to specify:

- the structural properties of the system, modeling with Class Diagrams the object classes, with their attributes and operations, and the relationships among them (association, aggregation, composition, generalization and dependency).

- the behavior of objects, either individual (i.e. “internal”) behavior, which can be modeled by means of hierarchical state diagrams based on Harel’s Statecharts [4], or interactive behavior (the exchange of requests for operations and responses), which can be described by means of Sequence Diagrams and Collaboration Diagrams.
It is important to note that behavioral specifications given by state machines require particular care when using inheritance mechanisms. Formal rules or, at least, some heuristics are necessary to guide designers in modifying a state diagram without losing its original behavior. For example, Douglass in [3] suggests that inherited state behaviors can be modified adding transitions or elaborating substates in inherited states, but inherited states or transitions should not be deleted. More formal approaches can be found in [21, 5].

Moreover, with regard to the specification of complex collaborative behaviors, the basic tools of UML have been acknowledged by many real-time methodologists quite weak, especially when considering real-time applications and event-based interactions. In fact, semantical extensions to the language, originally proposed in the UML Real-Time Profile of [23], have been included in the newer version of the UML Standard 2.0 [18]. The UML Real-Time Profile introduced particular kind of classes and Collaboration Diagrams, namely “capsules”, “ports” and “protocols”, in order to model signal-based collaborations and architectural structures supporting complex interaction between systems’ components. We can briefly describe the elements of the UML Real-Time profile as follows:

- “capsule” classes are active entities that can communicate with their surroundings only through signal-based connective objects called “ports”;
- “port” entities are aggregated to a capsule, each one defining a pass-through means for events, and
- each port plays a role in a well defined “protocol”, the definition of valid signal flows for a port;
- finally, “connectors” join the ports of capsules participating in a protocol.

As it can be seen, event-based collaboration between objects is greatly emphasized in Real-Time UML. From the behavioral point of view, this means that not only each capsule would have a state-based behavior, as is common in UML for reactive objects, but also the protocol itself may be specified by a state machine.

In [2, 22] the Real Time UML Profile has been extended and the concept of mechatronic class has been introduced. In this way, it is possible to use UML for modeling both physical and logical components of manufacturing system. For further details see [22].

2.2 The behavioral approach

The behavioral approach is a mathematical framework that aims to formalize a general dynamical system by focusing on its behavior, rather than on the equations that characterize it.

Consider a dynamical system characterized by an I/O interface \((u, y) \in U \times Y\) through which it can interact with the rest of the world. The interface outcomes space, namely the space on which inputs and outputs of the system live, is defined as

\[ W = U \times Y \] (1)
It is then possible to define the universe of interface outcomes $\mathcal{U}$ as the set of ALL possible trajectories that can be described in $W$, namely as

$$\mathcal{U} = \{(u, y) \in W^T\}$$

where $W^T$ is the set of maps from $T$, an ordered time set, to $W$. An interface behavior is defined as a certain $\mathcal{B} \subseteq \mathcal{U}$ of compatible interface outcomes, namely a particular subset of trajectories that the system is allowed to describe in $W$.

The interface behavior can be described by means of differential equations using the concept of jet space [16]. Let $\mathcal{G}$ be a sufficiently smooth manifold and $g(\cdot) \in \mathcal{G}^T$, a sufficiently smooth function. Let $\mathcal{G}_i$ denote the set of all possible instantaneous $i$th temporal derivatives for any possible $g(\cdot)$. The set

$$\mathcal{G}^{(n)} = \mathcal{G} \times \mathcal{G}_1 \times \cdots \times \mathcal{G}_n$$

is well defined and points in $\mathcal{G}^{(n)}$ are indicated as $g^{(n)}$. Consequently, for any sufficiently smooth function $g(\cdot) \in \mathcal{G}^T$, there is an induced function $g^{(n)}(t) = pr^{(n)} g(t)$ called the $n$th prolongation of $g(\cdot)$ which is defined by the following relation:

$$g^{(n)}(t) = (g(t), g_1(t), \ldots, g_n(t))$$

where

$$g_i(t) = \frac{d^i g}{dt^i} \bigg|_t$$

Thus $pr^{(n)} g(\cdot)$ is a function from the time set $T$ to $\mathcal{G}^{(n)}$. The extended $n$th-order outcome jet space is defined to be the space $T \times \mathcal{W}^{(n)}$. Under suitable mild conditions (see [27]), an outcome behavior $\mathcal{B}$ can be described by a continuous function $\Delta^\mathcal{B} : T \times \mathcal{W}^{(n)} \to \mathbb{R}^v$. The behavior is equivalent to the set:

$$S_{\Delta^\mathcal{B}} = \{(t, w) \mid \Delta^\mathcal{B}(t, pr^{(n)} w) = 0\}$$

namely to the kernel of the operator $\Delta^\mathcal{B}(\cdot, \cdot)$.

The concepts of interface outcome and of interface behavior are very general and they can be used for the study of general dynamical systems. For an extensive description of this approach see [20].

2.3 Basics on Category theory

Category theory is a branch of mathematics that has been developed to describe various structural concepts from different mathematical fields in a uniform way, providing an abstraction of many specific concepts in diverse branches of mathematics, including computer science and system theory. Intuitively category theory discusses a class of objects and their relationship to each other and it is formally defined as:

**Definition 1** A category consists of the following data

- A collection of objects $(A, B, C, \ldots)$
- A collection of morphisms $(f, g, h, \ldots)$; each morphism relates two objects and it is usually written $f : A \mapsto B$. 
• For each morphism $f$ there are two objects $\text{dom}(f)$ and $\text{cod}(f)$ called the domain and the codomain of $f$. Usually it is written $f : A \mapsto B$ to indicate that $\text{dom}(f) = A$ and $\text{cod}(f) = B$.

• Given morphisms $f : A \mapsto B$ and $g : B \mapsto C$ there is a morphism $g \circ f : A \mapsto C$ called the composite of $f$ and $g$.

• For each object $A$ there is given a morphism $1_A : A \mapsto A$ called the identity morphism of $A$.

These data are required to satisfy the associativity law (i.e. $h \circ (g \circ f) = (h \circ g) \circ f$ for all $f : A \mapsto B$, $g : B \mapsto C$, $h : C \mapsto D$) and the unit law (i.e. $f \circ 1_A = f = 1_B \circ f$ for all $f : A \mapsto B$).

A functor is a mapping from one category to another that preserves the categorical structure; it maps objects into objects and morphisms into morphisms and it preserves the associativity and the unit property. Formally we can give the following:

**Definition 2** Let $\mathcal{A}$ and $\mathcal{B}$ be categories. A functor from $\mathcal{A}$ to $\mathcal{B}$ is a mapping $F$ that sends objects of $\mathcal{A}$ into objects of $\mathcal{B}$ and morphisms of $\mathcal{A}$ into morphisms of $\mathcal{B}$ in such a way that:

• $F(f) : F(A) \mapsto F(B)$ whenever $f : A \mapsto B$

• $F(g \circ f) = F(g) \circ F(f)$ for each objects $A$ in $\mathcal{A}$

• $F(1_A) = 1_{F(A)}$

A functor that maps a category into itself is called endofunctor. Let $\mathcal{A}$ and $\mathcal{B}$ be categories such that each object $A \in \mathcal{A}$ can be regarded as an object of $\mathcal{B}$ by suitably ignoring structures that $A$ may have as an object of $\mathcal{A}$ but not as an object of $\mathcal{B}$. A functor $F : \mathcal{A} \mapsto \mathcal{B}$ which operates on objects of $\mathcal{A}$ by “forgetting” any imposed mathematical structure is called a forgetful functor. Loosely speaking a forgetful functor is a kind of generalized canonical projection which cuts away all the information which does not fit in the target space. For a more formal definition see [14].

3 A categorical view of classes and of dynamical systems

The coalgebraic framework proposed in [9] aims to describe a software program in terms of generated behavior instead of data processing or symbolic manipulation. We can interact with software artifacts by means of inputs and we observe the results of computations by means of outputs and, therefore, what we perceive is the I/O behavior and NOT the functional implementation of a program. Coalgebras formalize this black-box modeling approach in a suitable level of abstraction, based on category theory, and they are defined by:

**Definition 3** Let $\mathcal{A}$ be an arbitrary category and let $F : \mathcal{A} \mapsto \mathcal{A}$ be an endofunctor. A coalgebra consists of an object $X \in \mathcal{A}$ together with a morphism $c : X \mapsto F(X)$.
The object $X$ is often called state space and $c$ is called the transition or coalgebra structure. $X$ represents the kind of data that characterize the system, $\mathcal{F}(X)$ represents the interface by means of which we can interact and we can see the system and $c(\cdot)$ represents the dynamics of the system and it links the evolution of the state to the I/O interface. Intuitively, coalgebras describe general state-based systems provided with dynamics given by $c$. For a state $x \in X$, the result $c(x)$ tells what the successors of $x$ are, if any. The codomain of $c(\cdot)$ is often called the interface of the coalgebra.

In object-oriented software, the classes of objects are the basic elements, whose behavioral description is the fundamental issue during modeling and design tasks. It is possible to distinguish between class specifications and class implementations. Class specifications are linguistic entities consisting of three parts describing:

1. the methods
2. the logical assertions that the methods have to satisfy
3. the conditions that should hold for newly created objects

Class specifications define the features of a class in general without any reference to the specific data type which constitutes the state space of the class and without any specific interpretation of the methods of the class. A class specification defines what behavior a class should be able to reproduce and NOT how it has to be designed to do it. A very large number of class specification are characterized by methods having one of the following two forms:

$$ \text{at} : X \mapsto Y \quad \text{or} \quad \text{proc} : X \times U \mapsto X $$  \hspace{1cm} (6)

where $U$ and $Y$ are constant sets, not depending on the “unknown” type $X$. The method $\text{at}$ represents an attribute method which gives for a state $x \in X$ an observable attribute $\text{at}(x) \in Y$; one can only observe the state via such attributes. The method $\text{proc}$ represents a procedure method which has an effect on the state space $X$: it yields from a state $x \in X$ and a parameter value $u \in U$ a new state $\text{proc}(x, u)$ and the effect of this procedure can be visible via the attributes. $X$ is best seen as a black box to which we have limited access via the specified methods. In fact we do not really care about what is inside $X$ as long as $X$ comes with the methods that satisfy the assertions.

Class implementations are the interpretation of class specifications and, basically, correspond to class definitions in o-o programming languages. Thus, class implementations define the particular type of data that constitute the state space and the methods that implement those given by the class specification, satisfying the assertions. Formally, a class implementation is given by a triple

$$ (X, c : X \mapsto \mathcal{T}(X), x_0) $$  \hspace{1cm} (7)

where $X$ is an interpretation of the state space $X$, $c : X \mapsto \mathcal{T}(X)$ is a coalgebra that gives an interpretation of the methods, that is it consists of a set of functions that implement the methods defined in the specification; $\mathcal{T}(X)$ represents the I/O interface of the system. Finally $x_0 \in X$ is an element which should satisfy the creation conditions reported in the class specification.

We can access to the states only through the methods; thus, once a class implementation is given, it is important to identify the states which are indistinguishable
by the methods. This is formalized in [9] by giving the following definition of bisimulation, an equivalence relation on state spaces which implies the equivalence of the external behavior of two classes:

**Definition 4** [9] Consider a functor $T$ such that $X \mapsto T(X)$ and a coalgebra $\phi = (\phi_1, \phi_2) : X \mapsto T(X)$ of this functor giving interpretations of an attribute and a procedure method acting on a set $X$.

(i) A bisimulation relation on $\phi$ is a relation $R \subseteq X \times X$ on $X$ such that for each pair $x, y \in X$ we have that $(x, y) \in R$ if and only if

\[ \phi_1(x) = \phi_1(y) \text{ and } \forall u \in U, R(\phi_2(x, u), \phi_2(y, u)) \]

(ii) Two elements $x, y \in X$ are called bisimilar (with respect to the coalgebra structure $\phi$) if there is a bisimulation relation $R \subseteq X \times X$ such that $(x, y) \in R$.

Therefore, two states are bisimilar if it is not possible to distinguish them using the methods of the class.

Given a class specification $S$ it is possible to define a category $\text{Class}(S)$ of implementations of the specification. Summarizing the details of [9], the objects of $\text{Class}(S)$ are all the implementations of the specification $S$ and the morphisms link implementations whose external behavior is indistinguishable using the methods (i.e. they preserve bisimulation).

Physical components of a manufacturing system can be modeled as dynamic systems. Since within the coalgebraic framework for software classes the focus is on the generated behavior rather than on the functional implementation, we believe that a key ingredient for the extension of this modeling language to dynamical systems is the concept of interface behavior as in the behavioral approach. The basic idea in our coalgebraic description of dynamical systems is that a dynamical system can be seen as a class that implements a certain interface behavior. Analogously to what happens for o-o programming languages, it is possible to distinguish between dynamical system specification and dynamical system implementation. The first defines the interface behavior that a dynamical system must be able to reproduce and the ways in which it is possible to interact with the system (the methods) while the second represents a particular implementation of a given specification.

A dynamical system specification consists of three parts: methods, assertions and creation conditions. As for software classes, methods represent the way through which it is possible to interact with the system; they represent both a way through which we can get information about the system state and a way by which we can influence the system’s evolution. A fundamental difference between software classes and dynamical systems is time evolution. The external behavior generated at the interface of software classes is event-based and it is determined by the (asynchronous) invocation of the methods. One of the key characteristics of dynamical systems is that they evolve synchronously with respect to an ordered time set (e.g. $\mathbb{R}$, $\mathbb{Z}$, etc.), changing the information presented at the external interface at each time step (which can be infinitesimal). This means that for dynamical systems the methods that determine their generable behavior have to be invoked at each instant.

A very large number of dynamical systems can be represented by considering only two kinds of methods: the attribute method and the procedure method. Let $U$
and \( Y \) be a defined input and output space respectively and let \( X \) be the “unknown” state space \(^a\) and \( T \) be an unknown ordered time set. Notice that the specification leaves both the state space AND the time set free to be interpreted. In this way we leave a greater flexibility in the interpretation of the specification and, therefore, in building implementations which, for example, can be either time continuous or time discrete and whose state can evolve either on a manifold or on a lattice. The attribute method, renamed \texttt{out}, and the procedure method, renamed \texttt{evolve}, are given by:

\[
\texttt{out} : X \mapsto Y \quad \texttt{evolve} : X \times T \times U \mapsto X
\]

Thus, the \texttt{out} method maps the state into the output space and the method \texttt{evolve} encodes the evolution of the system and it depends on the time set \( T \). The role of the input is to parameterize the evolution of the system. Moreover, no specific structure is given to the methods. The method specification says only that we can see the state space only in the output space \( Y \) and that the system has to be able to evolve with respect to a time set and that we can influence the evolution by elements in the set \( U \). For software classes assertions represent logical conditions that the methods have to satisfy and they are sufficient to describe the external behavior that the class has to produce. For dynamical systems, logical assertions are no more sufficient since they do not easily allow to take the time evolution into account. In dynamical systems specifications, the assertions consist of a particular I/O behavior \( B \) that the system must be able to reproduce. This means that the methods of the dynamical system have to be related in such a way that it is possible to reproduce the behavior \( B \); it is possible to select a particular I/O trajectory in the behavior by selecting a proper input trajectory. Finally, the creation conditions represent the initial output condition that have to be satisfied by the implementations of the specification at an instant that is chosen to be the initial one. Thus, if the initial output of the system has to be equal to \( y_0 \) we must have that the initial state must lie in the preimage of the \texttt{out} method with respect to \( y_0 \). For dynamical systems, these creation conditions could be omitted. In this case we do not set any restrictions on the initial condition of the interface behavior. Summarizing, a dynamical system specification can be described by the following:

**Dynamical System spec:**

\begin{verbatim}
D methods
    out: X \mapsto Y
    evolve: X \times T \times U \mapsto X
assertions
    (u(t), y(t)) \in B
creation
    y(t_0) = y_0
\end{verbatim}

Dynamical systems implementations are the interpretation of dynamical systems specifications and they define the particular structure of the system. They are defined by the triple \(((X,T), c : X \times T \mapsto T(X), m_0)\), where \( T(X) \) represents the

\(^a\)\( X \) has to be such that it is possible to define state trajectories over the time set for the dynamical system. Nevertheless, no further assumptions need to be made neither on the dimension of \( X \) or on further structures it can possess (e.g. Lie group structure) or on other additional features.
I/O interface of the system. The choice of $X$ and $T$ implies the choice of a structure for the state space and for the ordered time set of the dynamical system that has to be such to guarantee the existence of trajectories. The coalgebra $c(\cdot)$ consists of a set of maps related in such a way to allow to implement the behavior requested in the assertions compartment of the specification. Basically the coalgebra defines the output map, which is associated to the method $\text{out}, h: X \mapsto Y$ and the evolution map, which is associated to the method $\text{evolve}, \Phi: X \times T \times U \mapsto X$. The map $\Phi$ is what is usually called state transition map in system theory [12]. The choice of the output map and of the evolution map has to be such that for every possible pair $(u(\cdot), y(\cdot))$ we have that

$$(u(t), y(t)) \in B$$

(8)

In other words we require that the system implements the specified behavior. Finally, $x_0 \in X$ is the initial state of the implementation and it has to be such that the initial output configuration satisfy the creation conditions reported in the specification.

Thus, we have seen that it is possible to represent both software classes and dynamical systems using the same mathematical framework. It is also possible to extend the concepts of bisimulation and of bisimilar states given in Def. 4 for dynamical systems implementations. In this case the definition becomes exactly that given in [26], meaning that the coalgebraic modeling framework for dynamical systems is consistent with the results already obtained in dynamical systems theory. Furthermore, given a dynamical system specification $D$, it is possible to define a category $\text{Class}(D)$ of dynamical systems that implement it. In this category there will be all the dynamical systems which implement the behavior given in the specification; morphisms will relate systems whose I/O behavior is indistinguishable using the methods, namely systems which are the same up-to-bisimulation [26].

4 Behavioral inheritance

Within the coalgebraic modeling framework, it is possible to define the concept of inheritance, both between class specifications and between class implementations, for object-oriented software [9]. A class specification $S$ inherits from a class specification $T$ if all the methods, assertions and creation conditions of $T$ form part of $S$, but $S$ may be extended with:

1. additional methods;
2. additional assertions (or redefined assertions of $T$);
3. additional creation conditions (or redefined creation conditions of $T$);
4. an extended output type $Y' \supseteq Y$ of an attribute method in $T$ and an extended input type $U' \supseteq U$ of a procedure method in $T$.

These four points ensure that implementations of the child specification $S$ are also implementations of the parent specification $T$, namely that the external behavior generated by $T$ is contained in the external behavior generated by $S$. Formally, we say that $S$ inherits from $T$ if and only if there is a forgetful functor $F$ between
the categories $\text{Class}(S)$ and $\text{Class}(T)$, whose role is to erase the extra structure we added in the specification $S$. Thus, a specification $S$ inherits from a specification $T$ if erasing some methods, deleting (or weakening) some assertions and creation conditions from any of its implementations we get an implementation of the specification $T$.

It is also possible to define inheritance also for class implementations: a class implementation $B \in \text{Class}(S)$ inherits from a class implementation $A \in \text{Class}(T)$ if there is a morphism of classes $f : \mathcal{F}(B) \rightarrow A$ in the category $\text{Class}(T)$. In this case $B$ is called a subclass of $A$. Formally:

**Definition 5** Consider a class specification $S$ inheriting from a class specification $T$, together with the resulting forgetful functor $\mathcal{F}$

\[ \text{Class}(S) \xrightarrow{\mathcal{F}} \text{Class}(T) \]

We say that a class $B \in \text{Class}(S)$ inherits from a class $A \in \text{Class}(T)$ if there is a morphism of classes $f : \mathcal{F}(B) \rightarrow A$ in the category $\text{Class}(T)$. We shall call $B$ a subclass of $A$ and $f$ a coercion map from $B$ to $A$.

Loosely speaking, a class $B$ is a subclass of $A$ if, after having forgotten the extra stuff (methods, assertion and creation conditions) added in the inheritance process, we obtain a class $\mathcal{F}(B)$ which is characterized by the same external behavior of class $A$, namely that is equal to $A$ up to bisimulation [9].

Using the coalgebraic framework described in the previous section, it is possible to extend the notion of inheritance to dynamical systems. We have reported a set of 4 possible extensions through which it is possible to build a child specification for software classes. We will now restate these possible extensions giving them a more system theoretic interpretation and defining the concept of inheritance for dynamical systems specification.

A dynamical system specification $D$ inherits from a dynamical system specification $E$ if all the methods, assertions and creation conditions of $E$ form part of $D$, but:

1. $D$ may have additional methods. It is possible to add outputs methods and to add evolution methods. In the first case we allow the child specification to have a deeper view of the state space through the methods. In the second case we allow the state to evolve in several ways; only one evolution method must be invoked at a certain instant meaning that the state can evolve following only one integral curve of an evolutionary vector field at a time. Adding several evolutions methods means to allow several possible choices for the state evolution and, consequently, for the interface behavior.

2. $D$ may have additional assertions and, moreover, the assertions of $E$ may be redefined. For dynamical systems, if $\mathcal{B}$ is the behavior required by the specification $E$, it means that the child system could implement a behavior $\mathcal{B}_1 \supseteq \mathcal{B}$. In this way all the trajectories of the parent behavior can be implemented by the child and, therefore, the LSP is satisfied. The fact of requiring a super-behavior of $\mathcal{B}$ to be implemented extends the capabilities of the dynamical system and it can be seen as an addition of extra assertions to the parent specification.
3. $D$ may have additional creation conditions and, moreover, the creation conditions of $E$ may be redefined. This extension allows to the derived system to have an extended set of admissible initial values for its outputs.

4. The output space $Y$ and the input space $U$ may be extended to super-spaces $Y' \supseteq Y$ and $U' \supseteq U$ respectively. It is therefore possible to extend the output space to support more kind of attributes and the input space to provide a further capability to parameterize the evolution of the state.

These four points imply that implementations of the child specification are also implementations of the parent specification and that children systems can be used in place of the parent system without any loss of performance but with an extra flexibility and extra functionalities added thanks to the inheritance process. This property can be formalized in terms of category theory in the following terms:

**Definition 6** Let $D$ and $E$ be two class specifications. We say that $D$ inherits from $E$ if and only if there is a forgetful functor $F$ between the categories $\text{Class}(S)$ and $\text{Class}(T)$

It is possible to straightforwardly apply the construction provided for classes implementations to dynamic systems implementations. Thus we have that an implementation $B$ of the child specification $D$ is a subclass of an implementation $A$ of the parent specification $E$ if, after having forgotten all the extensions introduced in the inheritance process, the external behavior of the class $F(B)$, where $F$ is the forgetful functor, is the same as that of the system $A$. Namely, if $A$ and $F(B)$ are the same up to bisimulation.

Summarizing, we have shown how to define inheritance for dynamical systems. Using a coalgebraic approach, we have achieved the following goals:

1. The inheritance process is described in the same way both for software classes and for dynamical systems, providing a further step in the definition of a unified framework for modeling distributed manufacturing systems.

2. Using the behavioral approach we have preserved a proper degree of generality and the proposed framework can be applied to general dynamical systems (e.g. linear, nonlinear, hybrid, discrete, etc.)

5 Example

The aim of this section is to provide two examples that illustrate the usage of the concepts introduced in the paper in the “design by extension” paradigm. In the first one, we will show that behavioral specifications for logic control systems, using UML state diagrams, can be formalized following the coalgebraic approach to oo software of [9], while in the second, we will describe a physical system using the same conceptual framework.

5.1 Behavioral extension in logic control

A kind of manufacturing system quite common in the packaging industry is the so-called “horizontal wrapper”, whose generic working principle can be schematized
as shown in Fig. 1. The processing principle of the machine is the following: products are inserted in an horizontal “tube” of packaging material (i.e. plastic film), 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.

![Packaging machine with horizontal flow](image)

**Figure 1** Packaging machine with horizontal flow

In this example, we will consider only a simple part of the logic control program of the system, particularly the one related to the temperature regulation of the longitudinal sealing unit. We assume that, first, a base version of this regulator is required and that its behavior (an ON/OFF logic) is specified by the UML Statechart of Fig. 2. The diagram shows that the system can react to the Init, Run and Stop events, it receives as an input the measured temperature of the sealing head and its (boolean) outputs are those related to the heater control signal and to the confirmation of good temperature regulation.

![Basic Statechart for a longitudinal sealer](image)

**Figure 2** Basic Statechart for a longitudinal sealer

Assuming that the admissible traces of the state machine graphically described by Fig. 2 are contained in the set $B$, we can formalize the behavior of the software module that will implement the temperature regulator by giving the following class specification:

```plaintext
class spec: SealingUnit
  methods
    out: X -> Y
    init: X x U -> X
```
in which there is a method for each event that the system can receive, specifying how the system changes its internal state in response to those events (according to the current value of its inputs), and a method to map the state of the system to the value of its outputs. The state space $X$ will be mapped in a specific data-structure during the implementation of the class, so that its formal details are not necessary. The assertion of the specification formalize that the I/O behavior of the system must be exactly included in the set $B$ of admissible traces of the class’ state machine. The creation conditions specify nothing else but the initial value of the system’s outputs.

An enhanced version of the SealingUnit class may be designed with the aim to speed up, if required, the initial ramp of the temperature, adopting for example an additional heater. Therefore, we assume that the behavioral specification for this derived class EnhancedSealingUnit is designed as shown in Fig. 3.

![Refined Statechart for a longitudinal sealer](image)

**Figure 3** Refined Statechart for a longitudinal sealer

We can write formal definition of the derived sealing unit class as follows:

class spec: EnhancedSealingUnit
methods
  out: $X \mapsto Y'$
  init: $X \times U \mapsto X$
  run: $X \times U \mapsto X$
  stop: $X \times U \mapsto X$
  fastrise: $X \times U \mapsto X$
assertions
  $(u, y') \in B'$
creation
In this case, the derived class specification contains additional methods and
the behavior $B'$, projected on the I/O space of the base class, contains $B$, since
the traces of the state machine of Fig. 3 include those of the state machine Fig. 2
(i.e. no occurrence of the $\text{FastRise}$ event), plus additional ones. Actually, the
refined state machine has been designed following the principles suggested in [3], in
particular by elaborating substrates in inherited states.

The formalization of the inheritance relationship between the two class of longi-
tudinal sealer is given, in our coalgebraic framework, by a forgetful functor that links
Class($\text{SealingUnit}$) and Class($\text{EnhancedSealingUnit}$), whose role is to project
the output space $Y'$ into $Y$, to drop the additional event response method and to
drop all the traces in $B'$ in which the event $\text{Fastrise}$ is present.

5.2 Behavioral extension in dynamical systems

Suppose that, as a part of a complex manufacturing machinery, we need a system
for moving a tool (e.g. a welder) over a circle of radius $l$ where a given working
process has to be implemented.

The following specification defines the class of dynamical systems that can imple-
ment the requested task:

**Dynamical System spec: Manip**

- **methods**
  - $\text{out}: X \rightarrow Y$
  - $\text{evolve}: X \times U \times T \rightarrow X$

- **assertions**
  - $(u(t), y(t)) \in B_m$

- **creation**
  - $x_p^2 + y_p^2 = l^2$

where $Y = \mathbb{R}^2$ is the output space and its elements, denoted with the pair $(x_p, y_p)$,
represent the position of the tool over the working plane and $U = \mathbb{R}$ is an input
that allows to change the position of the tool. The behavior $B_m$ consists of the
circular trajectory or, more formally, we can write

$$B_m = \{(u(\cdot), y(\cdot)) \in \mathbb{R}^2 \mid x_p^2 + y_p^2 = l^2\}$$

(9)

where $l$ is the radius of the circle.

An implementation of this specification is given by a 1-DOF robot, illustrated in
Fig. 4, whose link length is $l$. For the sake of simplicity, we will describe a robot
by means of a kinematic model where the input is the angular velocity imposed at
the joints and the output is the cartesian position of the end-effector [15]. Nothing
would conceptually change if we considered a dynamic (e.g. Euler-Lagrange) model.

Referring to Fig. 4, the kinematic model of a 1-DOF manipulator is given by:

$$\dot{\theta}(t) = u(t) \quad (10a)$$

$$\begin{bmatrix} x_p(t) \\ y_p(t) \end{bmatrix} = \int_0^t \begin{bmatrix} -l \sin \theta(\tau) \\ l \cos \theta(\tau) \end{bmatrix} u(\tau) d\tau \quad (10b)$$
where \( u \in U \) is the angular speed of the rotational joint. This system is an implementation of \( P \) where \( X = \mathbb{R} \) and \( T = \mathbb{R} \). The evolve method is given by the flow of Eq. (10a) and the out method is given by Eq. (10b). By simple computations, it can be shown that, independently of \( u \) and of any initial configuration, we have that \( x_p^2 + y_p^2 = l^2 \). Thus we have that the robot implements the behavior \( B_m \) and that the creation condition imposed by the specification is satisfied.

Suppose that we want to extend the capabilities of our manufacturing system by allowing to position the tool not only over a circle of radius \( l \) but in any point of a circle of radius \( 2l \). We can implement this extension using the inheritance process described in this paper. In particular, we can define the following specification \( \text{Manip2} \), which inherits from the specification \( \text{Manip} \).

**Dynamical System spec:** \( \text{Manip2} \)

**methods**

- **out:** \( X \mapsto Y \)
- **evolve:** \( X \times U_c \times T \mapsto X \)

**assertions**

\[
(u_c(t), y(t)) \in B_{m2}
\]

**creation**

\[
x_p^2 + y_p^2 \leq 4l^2
\]

where \( Y = \mathbb{R}^2 \) is the same output space considered in the previous specification and its elements are denoted in the same way. \( U_c = \mathbb{R}^2 \supset U \) is the space of the inputs that allow to change the position of the tool; notice that additional inputs are provided in the derived specification. The behavior \( B_{m2} \) consists of all the trajectories that lay inside a circle of radius \( 2l \) or, more formally, we can write:

\[
B_c = \{(u_c(\cdot), y(\cdot)) \text{ s.t. } x_p^2 + y_p^2 \leq 4l^2\}
\]  \hspace{1cm} (11)

Since \( B_{m2} \supset B_m \), the behavior reproducible by implementations of \( \text{Manip2} \) strictly contains that reproducible by implementations of \( \text{Manip} \). No additional methods are provided but the creation conditions imposed by the specification \( \text{Manip2} \) relax those required by the specification \( \text{Manip} \). An implementation of this specification is given by the 2-DOF robot that is illustrated in Fig. 5, where the two links have the same length \( l \). The kinematic model of this 2-DOF manipulator is given by:

\[
\begin{pmatrix}
\dot{\theta}_1 \\
\dot{\theta}_2
\end{pmatrix} = u(t)
\]  \hspace{1cm} (12a)
$\theta_1$, $\theta_2$, $l$

$(x_p, y_p)\quad l\quad \theta_2$

Figure 5 Two DOF Robot

$$\begin{pmatrix} x_p(t) \\ y_p(t) \end{pmatrix} = \int_0^t \begin{pmatrix} -l\sin(\theta_1) - l\sin(\theta_1 + \theta_2) \\ l\cos(\theta_1) + l\cos(\theta_1 + \theta_2) \end{pmatrix} u(\tau) d\tau$$

(12b)

where the dependence on time of $\theta_1$ and $\theta_2$ has been omitted to lighten the notation and where $u$ is the angular speed of the rotational joints. This system is an implementation of Manip$^2$ where $X = \mathbb{R}^2$ and $T = \mathbb{R}$. The evolve method is given by the flow of Eq.(12a) and the cut method is given by Eq.(12b). By simple computations, it can be shown that, independently of $u$, we have that $x_p^2 + y_p^2 \leq 4l^2$, where $l$ is the length of the links. Thus we have that the robot implements the behavior $B_{m2}$ and that the creation condition imposed by the specification is always satisfied.

Since Manip$^2$ inherits from Manip, all the behaviors reproducible by implementations of Manip should also be reproducible by implementations of Manip$^2$. In fact, it is sufficient to set the 2-DOF at an initial configuration such that $x_p^2 + y_p^2 = l^2$, namely such that the creation condition requested by $P$ is satisfied and to set $\dot{\theta}_2 = 0$; in this way, for any value of $\dot{\theta}_1$ the position of the end-effector will move on a circle of radius $l$.

Therefore, the role of the forgetful functor that links Class(Manip) and Class(Manip$^2$) is to drop all the initial configurations of the 2-DOF that are not allowed by the specification Manip and to drop all the trajectories in $B_m$ in which $\dot{\theta}_2 \neq 0$. This forgetful functor formalizes the inheritance of behavior between a basic 1-DOF robot and an extended one, namely a 2-DOF robot.

6 Conclusion

We have presented an o-o modeling framework for dynamical systems based on category theory and on the behavioral approach. This framework allows to formally define the concept of inheritance between classes of dynamical systems, with a strong emphasis on behavioral conformity. We consider the latter property very important in order to introduce the principle of design by extension in the domain of mechatronic system. The examples presented in the paper describe two simple cases of design by extension, in which a logic control system for a manufacturing system is refined by adding new states and new admissible events to its state machine and mechanical system like a 1-DOF robot is modified in order to have an additional DOF. In both cases, the derived classes exhibits a dynamic
behavior which inherits and extends that of the basic ones. Our future work aims to analyze more significant cases of study, in which the proposed modeling framework will highlight its usefulness for the design of “enhanced” mechatronic components.

References


Behavioral inheritance in object-oriented models for mechatronic systems


