A Component-Based Approach for Supervisory Control

Gábor Kovács
Dept. Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest, Hungary
Email: gkovacs@iit.bme.hu

Laurent Piétrac
Laboratoire Ampère
INSA Lyon
Villurbanne, France
Email: laurent.pietrac@insa-lyon.fr

Kiss Bálint
Dept. Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest, Hungary
Email: bkiss@iit.bme.hu

***Abstract*— This paper reports a novel approach for the supervisory control of discrete event systems. Based on components, the approach provides principles of object-oriented software design to be used in the framework of Supervisory Control Theory. The concepts of abstract and concrete components allow the modeling of complex systems at a high level of abstraction, making specification and control design easier. Besides aiming modularity and reusability, the proposed framework allows to introduce concepts of composition, polymorphism and inheritance into the design of supervisory controllers.**

I. Introduction

In the last decades, evolving complexity of industrial and commercial systems have raised the need for formal methods to ensure safe operation without extensive use of validation and verification. To answer these needs, Supervisory Control Theory (SCT) has been proposed, which guarantees that the closed loop behavior will meet the prescribed specifications.

On the other hand, SCT uses ordinary state machines for modeling, which makes the procedure of control synthesis cumbersome. Although state machines are well formalized, the size of their state space rises drastically with the complexity of the system modeled, posing a difficulty also for the control engineer responsible for modeling and specification, and also for numerical computation of the supervisor. To overcome this problem, different hierarchic and modular approaches for supervisory control has been proposed (see, for example, [1] and references within). However these approaches answer many of the arising problems, there exists no unified framework providing modularity and reusability, the two features mostly desired by control engineers.

In the domain of software development, the object-oriented paradigm has quickly become a widely used solution for these problems. The object-oriented paradigm proposed features like polymorphism or inheritance, which might be used also in control design, and are adopted by various approaches, e.g. by the standard IEC 61499 [2]. However, these principles have to be adapted such that they meet the formalism of SCT. Although propositions have been made to include object-oriented paradigms in the framework of supervisory control (see [3] or [4]), no unified approach exists to take advantage of object-oriented principles in the field of control design. In this paper, the author tries to bridge the gap between the two fields by proposing a component based methodology for supervisory control design.

The remaining part of this paper is organized as follows. Section II summarizes briefly the basic concepts and notations of SCT. Section III introduces components, building blocks of the approach, while section IV presents some concepts supported by the framework. Section V concludes the paper.

II. Preliminaries

Here only some fundamental principles and notations of SCT [5] and automata theory are presented in order to keep the paper as self-contained as possible. For further details the reader may refer to [6] and [7].

The discrete-event system G is described by the 5-tuple $G = (Q^G, \Sigma^G, \rho^G, q_0^G, Q_m^G)$ with Q^G as its state set, Σ^G as its event set, $\rho^G : Q^G \times \Sigma^{G^*} \to Q^G$ as its extended partial transition function, q_0^G as its initial state and Q_m^G as the set of its marking states. The event set Σ^G can be divided into the distinct sets of controllable and uncontrollable events so that $\Sigma^G = \Sigma_C^G \cup \Sigma_U^G$ where $\Sigma_C^G \cap \Sigma_U^G = \emptyset$. The notation $\rho(q, \sigma)!$ means that there exists a transition associated with the event $\sigma \in \Sigma^G$ leaving the state $q \in Q^G$.

The language generated by G is denoted by $L(G)$. The prefix closure of a language L is denoted by $\overline{L}$. An important operation on languages is the natural projection to a given alphabet Σ, defined by the followings:

$$\begin{aligned} P_\Sigma(\varepsilon) &= \varepsilon \\ P_\Sigma(\sigma) &= \begin{cases} \sigma & \text{if } \sigma \in \Sigma \\ \varepsilon & \text{otherwise} \end{cases} \\ P_\Sigma(\sigma.t) &= P_\Sigma(\sigma).P_\Sigma(t) \end{aligned}$$

The inverse projection is defined by $P_\Sigma^{-1}(s) = \{t \in \Sigma \mid P_\Sigma(t) = s\}$ and can be also extended to languages. The extension of these projections to languages is straightforward.

The synchronous composition of two DESs G_1 and G_2 describes the operation of a system in which G_1 and G_2 operates synchronously. The synchronous product operation of two DESs is defined by $L(G_1 \parallel G_2) = P^{-1}_{\Sigma^{G_1} \cup \Sigma^{G_2}}(L(G_1)) \cap P^{-1}_{\Sigma^{G_1} \cup \Sigma^{G_2}}(L(G_2))$.

The goal of supervisor synthesis is to define a supervisor which can restrict the operation of the system to meet the

constraints of the specifications, modeled by a DES E. The supervisor S is a function $S : L(G) \rightarrow \Gamma$ defined by $\Gamma = \{\gamma = PWR(\Sigma) \mid \gamma \supseteq \Sigma_U\}$ where γ represents the set of events authorized by S and $PWR(\Sigma)$ is the set of all subsets (the power set) of Σ. If the specifications are controllable, the automaton $H = S/G$ describing the supervised system is the product of G and E. In other cases the supremal controllable sublanguage (or maximal permissive sublanguage) of E can be found, which allows the greatest possible set of controllable events, see [8] and [9]. This sublanguage allows only those operations described by $L(G)$ which respects the constraints given by E.

III. A COMPONENT-BASED APPROACH

In order to illustrate the principles presented in this paper, a simple running example of a manufacturing cell depicted by Fig. 1 will be used in the followings. The system is composed of two 2-DOF pick and place arms, each one consisting of two translational joints and a gripper.

Development of a controller based on the methodology of SCT needs a model of the plant, describing the possible evolution of its IOs, and therefore resulting in a large state space. The classical way to overcome this problem is to decompose the system to subsystems, and then create models for these subsystems independently, which can then be composed by the operation of synchronous composition. This naive approach of modularity might simplify the modeling process, however, the specifications have to be given for the global model, using events corresponding to IOs of subsystems. Even if specifications are composed of sub-specifications, the problem of large state spaces arises during the synthesis of the controller. For this simple system, such a monolithical model contains nearly 4 000 states. The component-based methodology described in the sequel helps avoiding the problem of state explosion and provides a solution for reusing models and using different levels of abstraction, allowing easy modification of the controller if one of the parts is replaced by another with different technological details but with same functionnality (e.g. a hydraulic cylinder is replaced by an electrical linear drive).

The key concept of the object oriented paradigm is the object itself, so at first its definition should be studied. According to the common definition, the object is a part of the system which can be *uniquely identified* and is described by its *behavior* with regard to its environment, its *internal structure* and its *state*.

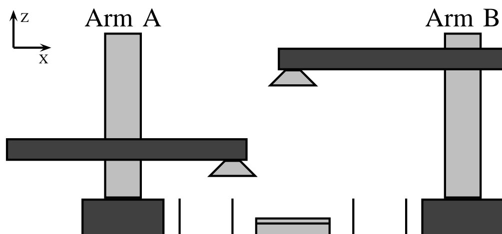

Fig. 1. Layout of the example

Considering the emphasized features, one can relate the notion of object to the concept of component, of which the system is built from. In case of a robotic arm, one would choose the gripper and the two joints as components. These components can be uniquely identified (i.e. gripper, axis X and axis Z), and their behavior, information on their internal structure in the form of possible evolution of their IOs and their states are stored in the corresponding FSM-models. Here another concept, namely the one of classes is recalled, which is a description of the organization and actions shared by one or more similar objects. Since the two joints are identical, one might consider the class Axis, and the objects Axis X and Axis Z as its instances. In the followings, the term component will be used in the meaning of class.

In order to achieve modularity and reusability, the first step is to separate the behavior (i.e. abstract functionality) and implementation of classes. In the object-oriented paradigm, it can be achieved by defining an abstract class, the interface, which contains only the declaration of the methods, and therefore describes a behavior in an abstract manner. Based on abstract interfaces, classes can be defined, implementing the methods. However, in the field of control engineering, one has to deal not with abstract constructs, but well defined, physically existing components, so the basis of modeling should be the physical manifestation of the component.

The aim is to, based on a traditional discrete-event IO model, obtain an appropriate functional model, describing the behavior of the component at a higher level of abstraction. The complete procedure of passing from a model to an other is given in [10], here only the most important definitions will be given and the modeling procedure, summarized by Fig. 2, will be briefly presented.

A. Models of a component

Definition 1: The technological model represents all possible evolution of inputs and outputs of the component which are allowed by its physical manifestation. Formally, the technological model is given by the 5-tuple $G^{Tech} = (Q^{Tech}, \Sigma^{Tech}, \rho^{Tech}, q_0^{Tech}, Q_m^{Tech})$.

1) Nominal model: In practice there are requirements which shall be respected no matter what functionality the given component should realize. They represent the avoidance of operations which are physically possible, and therefore allowed by the technological model, but which cause instable or dangerous behavior of the component. For example, a linear axis should not be started towards the negative direction if it is at the negative extremity. These specifications are referred to as Specifications of Safety, Security and Liveliness (S3L) which is given by the language E^{S3L}.

Since the aforementioned specifications need to be respected, a suitable nominal model can be obtained as the generator of the supervised plant respecting E^{S3L}, i.e. the minimal generator of the supremal controllable language of E^{S3L} with respect to G^{Tech}.

Definition 2: The nominal model defines the most permissive operation of the process respecting the specifications of

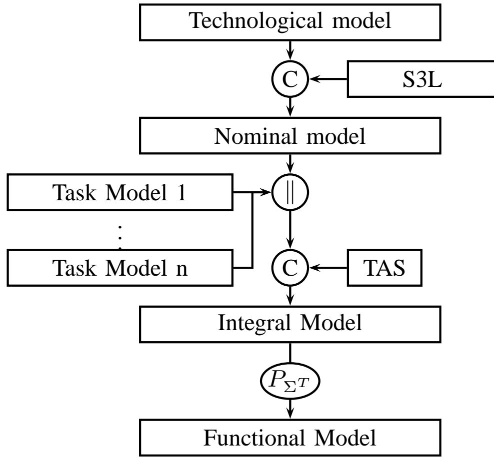

Fig. 2. Models of a component

safety, security and liveliness and is given by the 5-tuple $H^N = (Q^N, \Sigma^N, \rho^N, q_0^N, Q_m^N)$.

2) Tasks: The resulting nominal model depicts only operation of the component allowed by E^{S3L}. However, the description is given in details, by technological events, so the nominal model is an ideal candidate to build an abstract functional model on. The way from technological to functional representations is paved by the tasks, gathering appropriate event sequences of the nominal model.

Definition 3: The task core model is a suitably selected part of the technological model and is described by the 5-tuple $G_j^{Tc} = (Q_j^{Tc}, \Sigma_j^{Tc}, \rho_j^{Tc}, q_{0,j}^{Tc}, Q_{m,j}^{Tc})$ for the jth task. Its state and event sets are subsets of the nominal model, i.e. $Q_j^{Tc} \subseteq Q^N$ and $\Sigma_j^{Tc} \subseteq \Sigma^N$, and its transition function is defined such that the task core model generates a prefix event sequence as the nominal model starting from its state equivalent to $q_{0,j}^{Tc}$.

Before giving the definition of the task, two new events shall be introduced. The first one, corresponding to the start of a task, is the controllable *start event* and is denoted by σ_j^{start}. The other, reporting the completion of a task, is the uncontrollable *confirmation event* and is denoted by σ_j^{conf}. These events are collected to the alphabet of task events $\Sigma^T = \bigcup_j \{\sigma_j^{start}, \sigma_j^{conf}\}$.

Definition 4: A task is given by the 5-tuple $G_j^T = (Q_j^T, \Sigma_j^T, \rho_j^T, q_{0,j}^T, Q_{m,j}^T)$. The state set of the task core model is extended by a new initial state, i.e. $Q_j^T = q_{0,j}^T \cup Q_j^{Tc}$, and the event set of the task is the union of the event set of the corresponding task core model and the task events: $\Sigma_j^T = \Sigma_j^{TC} \cup \{\sigma_j^{start}, \sigma_j^{conf}\}$. The initial state of the task model is defined to be the newly added state, i.e. $q_{0,j}^T = q_0'$. The transition function is defined as follows:

$$\begin{aligned}
\rho_j^T(q_{0,j}^T, \sigma_j^{start}) &= q_{0,j}^{Tc} \\
\rho_j^T(q_{0,j}^T, \sigma) &= q_{0,j}^T, \ \forall \sigma \in \Sigma_j^{Tc} \\
\rho_j^T(q, \sigma_j^{conf}) &= q_{0,j}^T, \ q \in Q_{m,j}^{Tc}
\end{aligned}$$

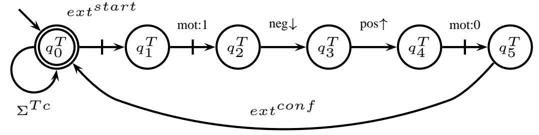

Fig. 3. Model of the extension task

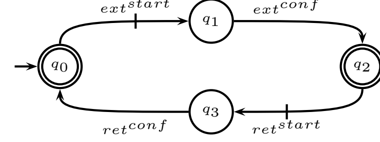

Fig. 4. Functional model of the axis component

For other states:

$$\rho_j^T(q, \sigma) = \rho_j^{Tc}(q, \sigma) \ , \ \forall \rho_j^{Tc}(q, \sigma)!$$

The only marking state of the task model is the initial state, i.e. $Q_{m,j}^T = \{q_{0,j}^T\}$.
Note that task models represent controlled behavior of the component.

The model of the extension task for a bistable translational joint realized by a linear motor is depicted by Fig. 3. Task core events in $\Sigma^{Tc} = \{$*mot:0,mot:1,pos* $\uparrow$, *neg* $\downarrow\}$ correspond to the stop of the motor, start of the motor to the positive direction, rising edge of the position sensor at positive extremal position and falling edge of the sensor at the negative extremity, respectively. Task start and confirmation events are ext^{start} and ext^{conf}, respectively.

3) Integral model: Now a common model can be defined, which comprises also the functional and the technological representations. An intermediate step is to create a model by the parallel composition of the task models and the Technological model and check its controllability with respect to the so-called Task Alternance Specification (TAS), which captures the property that technological events can only happen inside tasks (i.e. preceded by a task start event and succeeded by a task confirmation event), which is true if the model is well covered. It also allows the activity of only one task of a component at the same time.

If $G^{Tech} \parallel_j G_j^T$ is controllable with respect to $E^{TAS} = L(G^{TAS})$, then the model is said to be well covered by the tasks and the integral model can be defined as follows. Otherwise, task models have to be redefined or new tasks have to be included.

Definition 5: The integral model is the composition $G^I = G^{Tech} \parallel_j G_j^T \parallel G^{TAS}$ and is given by the 5-tuple $G^I = \{Q^I, \Sigma^I, \rho^I, q_0^I, Q_m^I\}$.

4) Functional model: The functional behavior of the model can be described by tasks, depicting the operation corresponding to a series of low-level events by two task events. Therefore, the operation (respecting the specifications of safety, security and liveliness)can be considered as starting tasks and waiting for their completion. So the functional

model can be obtained as the projection of the integral model to the task events.

Definition 6: The functional model is the generator of the language $L^F = P_{\Sigma^T}(L^I)$ and is given by the 5-tuple $G^F = \{Q^F, \Sigma^F, \rho^F, q_0^F, Q_m^F\}$ where $\Sigma^F = \Sigma^T$.

The functional model of a bistable translational joint is depicted by Fig. 4. Note that events of the model are start and confirmation events of the extension and retraction (*ext* and *ret*) tasks, and no events directly connected to the evolution of IOs are present. This abstraction allows to use the same functional model even if the physical component is replaced by another one with different technological details. However, in that case, the corresponding task models have to be also replaced to handle the low-level technological behavior described by the IO events.

Theorem 1: The language generated by the parallel composition of the functional model and the task models equals the language generated by the integral model:
$P^{-1}_{\Sigma^{Tech} \cup \Sigma^T}(L^F) \cap (\bigcap_j P^{-1}_{\Sigma^{Tech} \cup \Sigma^T}(L(G_j^T))) = L^I$
Proof of the theorem can be found in [10].

It is straightforward that the language of the integral model projected to the set of technological events is a subset of the nominal model, i.e. according to the theorem, the parallel composition of the functional model and the task models respects the specifications of safety, security and liveliness. This property suggests that the supervisory control architecture of a component might be decomposed to the set of task supervisors, which manipulate the IOs of the component in order to meet the task models, and a functional controller, which is responsible for the coordination of task controllers. These supervisors, based on the technological and functional models, can be synthesized using classical methods of SCT. For details, the reader is referred to [11].

B. Abstract and concrete components

Since the basis of the approach is the use of components, at first components have to be formally defined. Considering the models of a component, one can easily relate the functional model to the concept of interface as it describes the behavior by the mean of task events, but does not defines the task models themselves, i.e. it does not implements its methods and therefore can not be instantiated. It is straightforward that an abstract component can be related to a functional model, such that $C^* = G^F$. The abstract component is defined formally as follows.

Definition 7: An abstract component is a 5-tuple $C^* = (Q, \Sigma, \rho, q_0, Q_m)$, where the event set is composed of task start and confirmation events, i.e. $\Sigma = \Sigma^{start} \cup \Sigma^{conf}$, $\Sigma^{start} \cap \Sigma^{conf} = \emptyset$.

While abstract components are related to interfaces, i.e. abstract classes, components are analogous to concrete classes, which can be directly instantiated. In the object-oriented paradigm, the object's behavior is realized by the methods, which are invoked by sending appropriate messages to an object. In the framework presented in this paper, parts of a components behavior are represented by the tasks, which can be related to methods.

According to the analogy of classes and components, a component model should include all the task models it is capable to realize. On the other hand, a component might contain information about its physical representation, given by the technological model. Also, to describe the admissible behavior, the nominal model, or the specifications of safety, security and liveliness should be included. Since the nominal model can be easily derived from the specifications, only the latter has to be stored. Although, since the operation of computing the supremal controllable sublanguage might need significant computational effort, it is practical to include also its nominal model to the definition of a component.

Definition 8: A concrete component is given by a 4-tuple $C = (G^{Tech}, E^{S3L}, H^N, T)$, where G^{Tech} is the technological model of the component, E^{S3L} describe the specifications for safety, security and liveliness and H^N is the nominal model of the component. The set T contains the task models associated to the component, i.e. $T = \{G_j^T\}$.
Beside the concepts of concrete and abstract components, also their relationship has to be specified. In the software engineering field, it is said that a class implements an interface if it has all the methods defined in the abstract class. However, as defined above, the abstract component contains not just the set of task events (i.e. the declaration of methods), but also the functional model, i.e. in which order these events might follow each other. Therefore, the definition of implementation is more restrictive than in the field of software engineering.

Definition 9: An abstract component C^* is implemented by a concrete component $C = (G^{Tech}, E^{S3L}, H^N, T)$ if $P_{\Sigma^{Tech}}(L(C^*) \parallel_i L(G_i^T) : T_i \in T)$ is controllable with respect to $L(H^N)$, where Σ^{Tech} is the event set of G^{Tech}.

C. Composition

Object composition is a method to combine simple objects into more complex ones. Unlike subtyping or inheritance, which define an *is-a* relationship, composition expresses that a composite object (or component) is built up from simpler ones, i.e. it *has* objects (components) as its parts. In order to be unambiguous, the term atomic component will be used in the sequel for components not composed of other ones.

Unlike atomic components, composed components are not obtained as a result of modeling a physical component, but are assembled from previously defined component models, which will be referred to as its subcomponents in the sequel. Since the aim is to allow reusability and modularity as far as possible, composed components should be based on the composition of abstract subcomponents.

The behavior of a composed component is usually more restricted than the behavior of its subcomponents operating independently. If the working space of two robotic arms are disjoints, there is no need to specify constraints on their nominal behavior. However, when these two arms are situated as depicted by Fig. 1, only one of the robots can operate with its axis X extended in order to avoid collision. Therefore, beside the subcomponents, a specification describing their joint admissible behavior should be included in the composed

component. In order to allow further abstraction, tasks might be defined for the admissible behavior. The formal definition of the composed component is given as follows.

Definition 10: The composed component is a 4-tuple $\overline{C} = (\underline{C}, E^{coord}, \underline{H}, T)$, where $\underline{C} = \{C_1^*, \ldots, C_n^*\}$ is a set of abstract subcomponents, E^{coord} is the specification describing how abstract models in $\underline{C}$ are allowed to interact, while $\underline{H}$ is the minimal generator of the largest controllable sublanguage of E^{coord} with respect to $\|_i \; C_i^* : C_i^* \in \underline{C}$. T is the set of task models associated to the composed component.

In order to be coherent with the definitions of the previous sections based on atomic components, the set of underlying component $\underline{C}$ can be replaced by the automaton $G^{Tech} = \|_i \; C_i^*, \in \; \underline{C}$, so the composed component can be described by the 4-tuple $\overline{C} = (G^{Tech}, E^{coord}, \underline{H}, T)$, where $\underline{C} = \{C_1^*, \ldots, C_n^*\}$.
There is no difference between the definition of implementation in case of atomic and composed components, i.e. a composed component C is said to implement an abstract component C^* if $P_\Sigma^N(C^* \; \|_i \; L(T_i) : T_i \in T)$ is controllable with respect to $L(H^N)$, where Σ^N is the event set of H^N. However, if one investigates the model H^N, it is clear that its event set is the union (or a subset of the union) of events in $\underline{C}$, which is a set of abstract components. Therefore, technological events in the composed components model H^N correspond not to physical IOs, but task events of abstract subcomponents. As a consequence, tasks of the composed component will realize the desired behavior by invoking tasks of the subcomponents in the adequate order. For example, event set of composed component 2-DOF arm contains task events of abstract joints and an abstract gripper.

The operation of composition is naturally recursive, so subcomponents might also be composed ones. Composition allows control engineers to model complex systems using a top-down methodology, just like in the field of software engineering. Regarding the example of the manufacturing cell, it is straightforward that the system can be decomposed to two components, namely the two pick-and-place arms. According to the actual configuration, these abstract components are now realized by a concrete component consisting of two translational joints and a gripper. Subcomponents of the arm component, i.e. joints and the gripper are also abstract ones. These components can not be further decomposed (or they are not needed to be further decomposed), so these components are considered as atomic ones. Following this procedure, the component tree of Fig. 5. can be constructed.

The procedure of implementation follows a bottom-up methodology. At first the atomic concrete components implementing the abstract ones are selected and the controllers for their tasks are synthesized. Then, following the edges of the component tree, controllers for the tasks of composed components are obtained. This methodology allows various implementations for the supervisory control system.

IV. Features

In [12] the author has studied the terms used in OOP-related references and collected the ones mentioned the most.

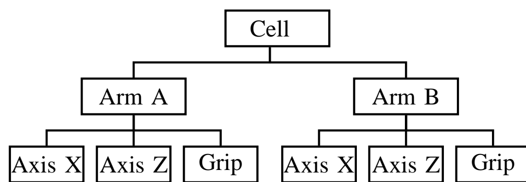

Fig. 5. Component tree of the cell

Besides the fundamental concepts of objects, classes and methods, the features mostly associated to the object-oriented paradigm are *encapsulation, inheritance* and *polymorphism*.

A. Encapsulation and local responsibility

According to the definition, encapsulation is a technique for designing classes and objects that restricts access to the data and behavior by defining a limited set of messages that an object of that class can receive. The aim of encapsulation is to keep the information and the way it is processed strictly together, and separate them from other objects of the system.

This is exactly what the framework using tasks provides. Events of concrete components are divided into two disjoint sets, namely the set of technological events, corresponding to the events of IOs (or tasks of subcomponents) and the set of task events. Composition of objects and control synthesis is carried out on abstract components, which contain only the signatures (i.e. the start and configuration events) of the tasks. Therefore, other components, even those which are composed of the given component, might influence the behavior of the given component by task start events, which can be related to messages invoking a given method. For example, the start event of the 'Retract axis Z' task of the composed component Arm A can be considered as a message sent towards the axis component, requesting its retraction. After finishing this operation, the axis sends a reply message to Arm A in the form of the task confirmation event. Other events, or the actual state of the IOs are unknown for the arm component and also for its supervisor.

Another important concept is local responsibility. Each component has its own scope of responsibility within it can act, i.e. it can only enable or disable controllable events included in its task models. Moreover, components might only gain information directly from their environment only by the events included in their task models. It means that IOs of a given component can not be set or read by other components. However this feature might seem restrictive at first sight, it ensures safety and security as responsibility for ensuring the correct behavior is delegated to the component, e.g. the details of the operation of the joints has not to be considered when designing a controller for the arm.

B. Inheritance

In object-oriented software design, inheritance is a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class. In the proposed approach, a significant distinction has been made

between abstract and concrete components, so the concept of inheritance should be defined within and between them.

Inheritance between an abstract and a concrete component is given by the property of implementation, when the given concrete component implements all the tasks of which the signatures were given in the abstract component. An abstract component can be thought of as an abstract class with virtual methods, and its methods are overridden by the methods (i.e. tasks) of the concrete component. Note that multiple inheritance is supported by the framework, as a concrete component can implement the behavior of various abstract ones, e.g. a linear motor can either realize a bistable actuator or one which can be stabilized in any position. However, generally these behaviors can not be implemented simultaneously, so even a concrete component inherits behaviors from multiple abstract ones, it has to be selected which of them it will implement in the actual system.

Between abstract components, inheritance affects only the abstract behavior which they define, so an abstract component can be considered the child of an other if it describes at least the same behavior.

Definition 11: An abstract component C_1^* is a descendant of the abstract component C_2^* (resp. C_2^* is an ancestor of C_1^* if $L(C_2^*) \subseteq L(C_1^*)$.

In case of concrete components, inheritance concerns not only abstract behavior, but also its implementation, so a concrete component is considered to be the child of an other one if it implements the same tasks in the same way.

Definition 12: A component C_1 is a descendant of the component C_2 (resp. C_2 is an ancestor of C_1), if $T_2 \subseteq T_1$ and $L(H_1^N) : H_1^N \in C_1$ is controllable with respect to $L(H_2^N) : H_2^N \in C_2$.

Note that the inheritance between two concrete components is a strict relation. It is possible that a concrete component can implement the same abstract behavior as an other concrete one, however, it is not the descendant of the other since it implements the behavior in a different way. For example, a linear drive can implement the behavior of a monostable pneumatic cylinder, however, since the operation of the linear drive and therefore its IOs are different, it is not a descendant of the pneumatic cylinder.

C. Polymorphism and substitutability

Polymorphism is another key feature of the object oriented paradigm. It is the ability of different classes to respond the same message and implement the method appropriately. Analogously, an abstract component is polymorphic if there exists more than one concrete component implementing it. Note that, according to the definition of implementation, the concrete component needs to have not only the corresponding task models, but also has to be able to execute the tasks in the order specified by the abstract component.

If two concrete components implement the same abstract component, they can be substituted with each other, since they show the same interface (by the mean of their task events) to other components. When using the traditional approach of supervisory control, even a slight technological modification (e.g. replacing a hydraulic cylinder by a linear motor) needs the remodeling of the whole system, and therefore the resynthesis of the supervisor. The property of substitutability provides that if one component of the system is changed, only the model of the given part has to be constructed, and therefore only the part of the controller corresponding to the changed component has to be newly synthesized (e.g. only the task supervisors implementing the behavior of the axis component has to be changed, other parts of the control system remain the same).

V. Conclusion

The presented framework allows the use of several principles of object-oriented software design in the framework of Supervisory Control Theory. The use of components provide a modular development method supporting reusability while allowing the control engineer to take advantage of concepts like inheritance or polymorphism.

Future work includes the definition of a component library enforcing the reuse of components and the definition of supervisory control architecture based on the presented framework.

Acknowledgment

Research presented in this paper was partially funded by the Hungarian National Scientific Research Foundation grant OTKA K71762. Also, it is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project, supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References

[1] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Boston: Kluwer Academic Publishers, 2000.

[2] V. Vyatkin, *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. Instrumentation, Systems and Automation Society, 2007.

[3] M. A. Shayman and R. Kumar, "Process objects/masked composition: an object-oriented approach for modeling and control of discrete-event systems," vol. 44, no. 10, pp. 1864–1869, 1999.

[4] M. Fabian and B. Lennartson, "Petri nets and control synthesis: An object-oriented approach," in *In Proceedings of the I.M.S*, 1994, pp. 13–15.

[5] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81 –98, jan 1989.

[6] W. Wonham, *Notes on Control of Discrete Event Systems*. Toronto: University of Toronto, 2002.

[7] J. Hopcroft and J. Ulmann, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[8] R. Kumar, V. Garg, and S. Marcus, "On controllability and normality of discrete event systems," *Systems & Control Letters*, vol. 17, pp. 157–168, 1991.

[9] R. Brandt, V. Garg, R. Kumar, F. Lin, S. Marcus, and W. Wonham, "Formulas for calculating supremal controllable and normal sublanguages," *System & Control Letters*, vol. 15, pp. 157–168, 1990.

[10] G. Kovács and L. Piétrac, "Multi-face modeling for rapid prototyping of discrete event control systems," in *Proc. European Control Conference 2009*, 2009.

[11] G. Kovács, L. Piétrac, and E. Niel, "Supervisory control based on multi-face modelling of discrete event systems," in *Proc. 10th International Workshop on Discrete Event Systems*, 2010.

[12] D. J. Armstrong, "The quarks of object-oriented development," *Commun. ACM*, vol. 49, pp. 123–128, February 2006. [Online]. Available: http://doi.acm.org/10.1145/1113034.1113040