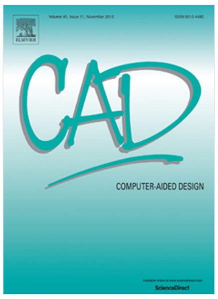
# An Automated Approach for Execution Sequence-Driven Software and Physical Co-Design of Mechatronic Systems Based on Hybrid Functional Ontology

Yue Cao, Yusheng Liu, Xiaoping Ye, Jianjun Zhao



PII: S0010-4485(20)30135-4
DOI: https://doi.org/10.1016/j.cad.2020.102942
Reference: JCAD 102942

To appear in: *Computer-Aided Design*

Received date : 9 January 2019
Revised date : 17 December 2019
Accepted date : 15 August 2020

# An automated approach for execution sequence-driven software and physical co-design of mechatronic systems based on hybrid functional ontology

Yue CAO$^1$, Yusheng LIU$^1$*, Xiaoping YE$^2$, Jianjun ZHAO$^3$*

(1. State Key Lab. of CAD&CG, Zhejiang University, Hangzhou, P.R.China, 310027)

(2. College of Engineering and Design, Lishui University, Lishui, P.R.China, 323000)

(3. School of Mechanical Science & Engineering HUST, Wuhan, P.R.China, 430074)

**Abstract:** Most mechatronic systems have becoming software-intensive. Even in their early design, the software and physical domains intersect with each other deeply. It is significant to capture and process the cross-domain influences automatically to avoid design defects in late stages. Semantic web technologies have been recognized as effective enabling technologies to support cross-domain knowledge representation and inference. However, how to correlate the knowledge of software and physical designs, which have divergent characteristics in an ontological knowledge is the difficulty. In this study, with the help of semantic web technologies, an automated software-physical co-design approach is proposed based on a hybrid function ontology, which unifies the physical-centric flow-based functional representation and software-centric data/control flow diagram. Software and physical designs are linked to this unified functional ontology such that the execution sequences controlled by software can constrain the physical design and the updates on the execution sequences introduced by physical design can be reflected on software behaviors. An ontology-based framework is implemented to support this approach. Two case studies from different application areas are illustrated to show its effectiveness.

**Keywords:** Software-intensive mechatronic system, software-physical co-design, functional modeling, design automation, ontologies

## 1. Introduction

Most mechatronic systems (MTSs) have been becoming software-intensive [1]. MTSs generally comprise basic systems (BSs), which is composed by multi-physics subsystems, low-level controllers, sensors, and actuators to implement regulated continuous physical processes, coordinated by high-level information processing [2]. In essence, the high-level information processing is implemented by "*software*". The design of MTSs requires integrated efforts from multiple domains [3]. The synergetic design of multi-physics and low-level controllers that regulate the physical processes have been well investigated by many researchers from conceptual to detailed design [4–6]. However, how to combine the design of software with them is still an open problem[1,7]. Many sequential design methods that develop software after the physical design has been frozen have been widely criticized [8,9]. In fact, software and physical designs are closely correlated with each other in early design. For example, the workflow coordinated by software constrains the selection of working structures of physical subsystems, and the chosen physical concepts may introduce new functions that change the workflow of software. If these cross-domain influences are not considered in early design, design defects and unexpected changes may occur in late stages and hence have to be solved with a high cost. Therefore, it is significant to provide computational support to capture and process such cross-domain influences in early system design.

In this study, the software-intensive mechatronic systems with high demand on execution sequences, e.g. systems of manufacturing and robotics domains are focused. They are required to achieve functions in a step-by-step manner following certain sequences. Such execution sequences work as important constraints on the physical concept selection and software generation, and meanwhile, may be changed by the physical and software design processes. Therefore, a specific aspect of software-physical influences which concentrates on the execution sequences of actions of the systems can be observed and is investigated in this study. An automated approach to assist the software and physical co-design driven by execution sequences is proposed. Semantic web technologies have been adopted as the main enabling technologies since they provide unified computer-readable representation and powerful reasoning capability across domains. They have been intensively used to share and infer knowledge among different disciplines, design stages, and platforms [10–13]. However, the heterogeneous and unlinked knowledge of software and physical designs hinders their co-design. To this end, a hybrid functional ontology that unifies their functional representations is proposed to correlate the software and physical designs in an ontological knowledge base. Software and physical designs are explicitly linked to the unified functions and thus execution sequences of physical processes can be captured and affect the software and physical design automatically. Specifically, physical structures can be constrained by execution sequences of functions coordinated by software, and behaviors of software can be generated and adapted to physical structures. The contributions of the proposed approach are two-folds: (1) a hybrid functional ontology unifying software and physical functions is proposed and the structure and behavior knowledge of physical and software design is linked through it; (2) reasoning on the design knowledge is enabled by semantic web technologies such that the execution sequences extracted from the hybrid functional ontology can be used to constrain and generate the physical and software design simultaneously.

This paper is organized as follows. Literature related to multidisciplinary co-design is reviewed in Section 2. Section 3 gives an overview of the proposed approach. The hybrid functional ontology is introduced in Section 4. The method of evaluating physical modules by execution sequences is discussed in Section 5. Automated software behavior generation and updating are explained in Section 6. The implementation and case study are introduced in Section 7. Section 8 discusses the advantages and limitations of this approach. Section 9 concludes this paper and summarizes future works.

## 2. Related work

### 2.1 Multidisciplinary co-conceptual design

It has been a consensus and necessity that multidisciplinary co-design of MTSs should be based on a common knowledge representation. The architecture of an MTS can be described from multiple viewpoints addressing different concerns held by the system's stakeholders[14]. In this study, three viewpoints, i.e., functional, structural and behavioral which are commonly involved in the conceptual design process are the focuses. Therefore, the co-design approaches are classified and reviewed according to the unified representations they are based on from three viewpoints.

Functional modeling is an important research area in engineering design. Among the various functional modeling methods, the most significant is from Pahl and Beitz [15] and Hirtz *et al.* [16], who proposed a flow-based functional representation and the functional basis as a unified set of verbs and nouns for functions and flows [17]. This unified functional representation has been broadly used in many design areas. For example, Wan *et al.* [18] proposed a functional-level co-design method for automotive cyber-physical systems; Chen *et al.* [19] implemented a knowledge-based framework to support creative conceptual designs of multi-disciplinary systems; Helms *et al.* [20] introduced a computational synthesis approach to generate product architecture solutions. Another commonly used functional representation is the Function-Behavior-State (FBS) functional ontology [21], which represents functions as hierarchies involving subjective purposes, objective behaviors, and states. Based on this functional representation, Cabrera *et al.* [5] proposed an architecture model to support cooperative design of mechatronic products. However, these functional representations mainly focus on continuous changes of attributes (states) of objects that are physical-centric whereas the discrete execution sequences of these changes controlled by software are overlooked. On the contrary, behavior trees [22] are a formal graphical modeling language to capture the expected behaviors of systems represented in natural language requirements by their states and sequences among these states. However, it focuses on the discrete behaviors controlled by software. Therefore, these functional modeling formalisms are difficult to reveal correlations between software and physical design.

Some researchers support the co-design of multi-disciplines based on a common structural model. Zheng *et al.* [23] defined a multi-disciplinary interface model for designing mechatronic systems. The interface incompatibility captured in the models can drive architecture design evolution [24]. Thramboulidis [25] proposed a 3+1 SysML view model to correlate the multi-domain design in a development process based on the V-model. Kernschmidt *et al.* [26] proposed a model-based framework based on the *SysML4Mechatronics* profile for the design and change influences analysis across disciplines. Fan *et al.* [27] proposed an approach to solve design conflicts between multi-domains by optimizing the values of design variables in a unified system design model. According to these methods, the co-design of multi-domains are mainly driven by incompatible interfaces/ports or conflicting values of design parameters. Therefore, they are difficult to capture design decisions from the behavioral perspective.

Unified behavioral models can support the co-design of multi-domains by simulation. Cao *et al.* [28] proposed a unified behavior modeling formalism to support co-simulation of physical and control systems of MTSs. Tian and Voskuijl [29] proposed a multi-physics simulation model to provide information for configuring simulations such that the multi-domain optimization can be supported. Behavior models are created late in the system design process after architectures are generated. Therefore, these methods can be used for optimizing and validating instead of generating designs.

Based on the above analysis, it can be seen that function-based co-design, in which the cross-domain design decisions are captured and shared in a common functional model has at least two-fold advantages. First, co-design can be supported from both structural and behavioral perspectives since functional models specify both interfaces and intended behaviors of components. Second, functional models are created before structural and behavioral models such that design changes can be noticed and solved earlier. However, the prerequisite is that software functions should be integrated with existing physical-centric functional modeling formalisms.

Besides the architectural-level co-design, some studies focus on the co-design of physical structure and their continuous low-level controllers. Malmquist *et al.* [30] proposed a holistic design methodology that generates the optimal design concepts against both static and dynamic criteria. Some works evaluated the synthesized design concepts by defining quantitative criteria which consider the objectives of various domains simultaneously. Typical criteria are the mechatronic design quotient [31] and multi-criteria profile [32]. However, these works did not take high-level control software into consideration.

## 2.2 Multidisciplinary co-detailed design

In this stage, the main concern of design engineers is to determine the optimal values of design variables and parameters of each subsystem. Li *et al.* [33] presented the Design For Control (DFC) approach which emphasizes obtaining a simple dynamic model of the mechanical structure to facilitate the controller design. Mohebbi *et al.* [34,35] applied and further improved the DFC methodology to a mechatronic quadrotor system to conduct a system-level optimization including both structures and controllers. Behbahani and de Silva [36] presented a systematic methodology for detailed mechatronic design based on the mechatronic design quotient. Mohebbi *et al.*[37] proposed a multidisciplinary objective function, by optimizing which, the detailed design can be accomplished. Van Brussels *et al.* [38] proposed a co-engineering framework, which can derive the motion controller from the finite-element description of the mechanical structure and optimize the structure and controller simultaneously to achieve robust performance for all configurations. These methods deal with attributes instead of architectures and hence cannot be applied to the architectural design stage.

## 2.3 Behavioral synthesis of control software

Some works attempted to generate the behavior of control software from the system design. Sakao *et al.* [39] proposed an approach which can generate both the qualitative and quantitative control sequence from the FBS model. However, the generated control program is centralized instead of modularized. Tranoris *et al.* [40] proposed a framework to generate the IEC 61499-based control model from the system design in SysML. Vogel-Heuser *et al.* [41] proposed a model-based design approach to support the whole development lifecycle from requirement engineering to code generation based on IEC 61131-3. However, these approaches essentially rely on model transformation so that the system design is required to provide all the information needed for control behavior generation.

## 2.4 Mechatronic system architecture

Components of mechatronic systems normally are organized in a hierarchical way to construct the system architectures. Thramboulidis [8,25] proposed the mechatronic system V-model including *system*, *composite mechatronic component* (MTC), *primitive MTC*, and *discipline-specific components*. Bassi *et al.* [42] vertically divided the architecture into *system*, *subsystem*, and *component* levels. Barbieri *et al.* [43] developed the system in two levels, i.e., *module* and *component*, but they did not explain the definitions and differences of these two concepts [44]. The hierarchy proposed by Zheng *et al.* [45] includes *system*, *subsystem*, *mechatronic module*, and *discipline-specific components*. These hierarchies do not differentiate the software and physical parts and hence cannot show the correlations between their designs.

Some architectures differentiated the software and physical parts. The reference architecture defined in VDI 2206 [2] differentiated the components into BSs and information processing. A BS is further decomposed into basic physical systems, controllers, sensors and actuators. However, the high-level information processing is not further modularized.

## 2.5 CPS design

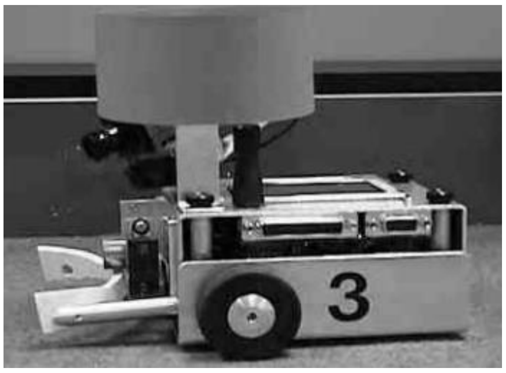
Some works from the CPS design area are also inspiring to this study. Specific to the industrial automation systems, Thramboulidis [46] classified the components of the system as *composite Cyber-Physical Component* (CPC), *primitive CPC*, *cyber component* and *physical component*. This work was enlightening since it explicitly differentiated the cyber and physical parts, but it doesn't clarify how the modules are determined. Vanherpen *et al.* [47] related models and ontologies to support the consistency check of different views for designing CPSs. However, this approach mainly addressed the static properties such as safety, load, cost instead of dealing with dynamics of the systems. Penas *et al.* [48] compared the mechatronic systems and CPSs and illustrated how the current techniques used in mechatronic design can be extended to CPSs design. However, no systematic design approach is proposed. Fitz *et al.* [49] proposed a metamodel for CPSs. It included not only the computing and physical parts but also the networking related description, which is essential for CPS design. However, it only involves structural perspective without considering functions and behaviors. In summary, currently, there is still a lack of comprehensive architecture model or systematic design approach for CPSs design.

## 3. Method overview

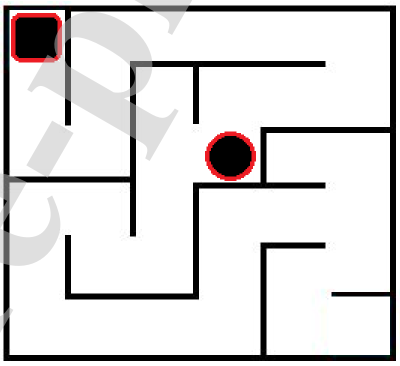
### 3.1 Motivation analysis

The main challenge to be addressed in this study is to provide a computer-aided design approach to capture and deal with the influences between software and physical designs of MTSs. Here, a mobile robot is used as a motivational case study to illustrate how the two domains influence each other in a typical system design process.

The mobile robot is designed to fetch a colored can in a maze. The concept of the robot is shown in Fig. 1(a), and the task is sketched in Fig. 1(b). The square represents the robot at its initial position and the circle stands for the can to be picked. To achieve this function, the robot will first detect obstacles (walls or the can) around it, i.e., on its left, right, and in front, and determine the types of obstacles according to their colors. If the can is detected, the robot will turn to the correct direction to pick it up. When no can is detected around the robot, it will turn to an indicated direction and move forward to the next square. The direction to proceed with is determined by some trajectory planning algorithm. For example, the "*wall hugger*" algorithm [50] can be used.



(a) Concept of the mobile robot [50]



(b) Sketch of the task

Fig. 1 Mobile robot

In a typical system design process, system designers first identify system functions, i.e., tasks to be realized by the system from the above-mentioned requirement specification. For example, *DetectColor* and *DetectPosition* are functions to detect and classify obstacles around the robot; *TurnLeft*, *TurnRight*, and *MoveForward* are required for the robot movement. Note that some functions are required to be executed in a certain order. For example, when the robot is instructed to move one square left, it should turn left first, and then move forward.

After functional analysis, designers then synthesize the functions into a system architecture constituted by multidisciplinary components. Generally, functions to change or detect the physical status of a system or its environment (so-called *physical functions* in this study) should be realized by multi-physics mechanisms with their controllers; functions for execution coordination and high-level control strategies (so-called *software functions*) are implemented by software.

To synthesize physical functions, physical modules can be retrieved from a module repository. Note that when multiple functions are planned to be implemented by a single physical module, the execution sequences of these functions should be considered as constraints when selecting the modules. For example, both functions *TurnLeft* and *MoveForward* of the mobile robot can be implemented by a *wheeled cart* with either *differential drive* or *Ackermann steering*. In the former solution, the wheeled cart has two independent driven wheels placed on the two sides of the cart and one or two passive wheels. In the latter solution, the cart has two combined driven rear wheels and two combined steered front wheels. However, the latter solution should be excluded since it has to turn and move at the same time instead of independently as required by the two functions. Considering the execution sequences are coordinated by software, it reflects how software influences physical design.

Software programs implementing high-level control strategies should be designed by specialists in machine learning and control [51]. They are not the focus of this study. Instead, software modules coordinating execution sequences, so-called *software coordinators*, are the objectives. This type of modules should be designed according to the workflow of functions. For example, the movement of the robot is coordinated by *TransferCoordinator*, which sends requests to the controller of the wheeled cart to trigger turning and moving actions in sequence as specified in the functional model. Such software modules are closely related to physical design and hence have to be adapted to physical functions. For example, designers may choose a physical module *jaw* to implement the function *Pick*. Before picking the can by the jaw, the jaw should be extended to be close enough to the can. A new physical function *ExtendJaw* is introduced, which will cause updates of software behaviors.

From the just described system design process, two important aspects of influences between software and physical designs can be recognized: (1) selection of physical modules is constrained by the workflow of functions coordinated by software; (2) behavior of software coordinators should be generated and updated according to functions of physical structures. These influences are originated from the functional-level so that a unified functional model can serve as a common source of the two domains of designs to capture and propagate cross-domain design decisions.

## 3.2 Method overview

To automate influencing between software and physical designs, a co-design approach is proposed by leveraging semantic web technologies as shown in Fig. 2. The core of this approach is the ontological knowledge base. Such knowledge base normally is constituted by two components, including the TBox storing a set of universally quantified assertions (inclusion assertions) stating general properties of concepts and roles, and the ABox comprising assertions on individual objects (instance assertions) [52]. The TBox of the knowledge base in this approach constitutes three ontologies. The Hybrid Functional Ontology (HFO) unifies the software and physical functions by merging the flow-based functional representation [15] and the data/control flow diagrams [53] commonly used to describe physical and software functions respectively. The Physical Structure Ontology (PSO) and Software Coordinator Ontology (SCO) describe the physical and software designs respectively, in which physical and software structures are linked to the unified functions through their behaviors. Behaviors are the processes executed by physical or software structures to realized certain functions.

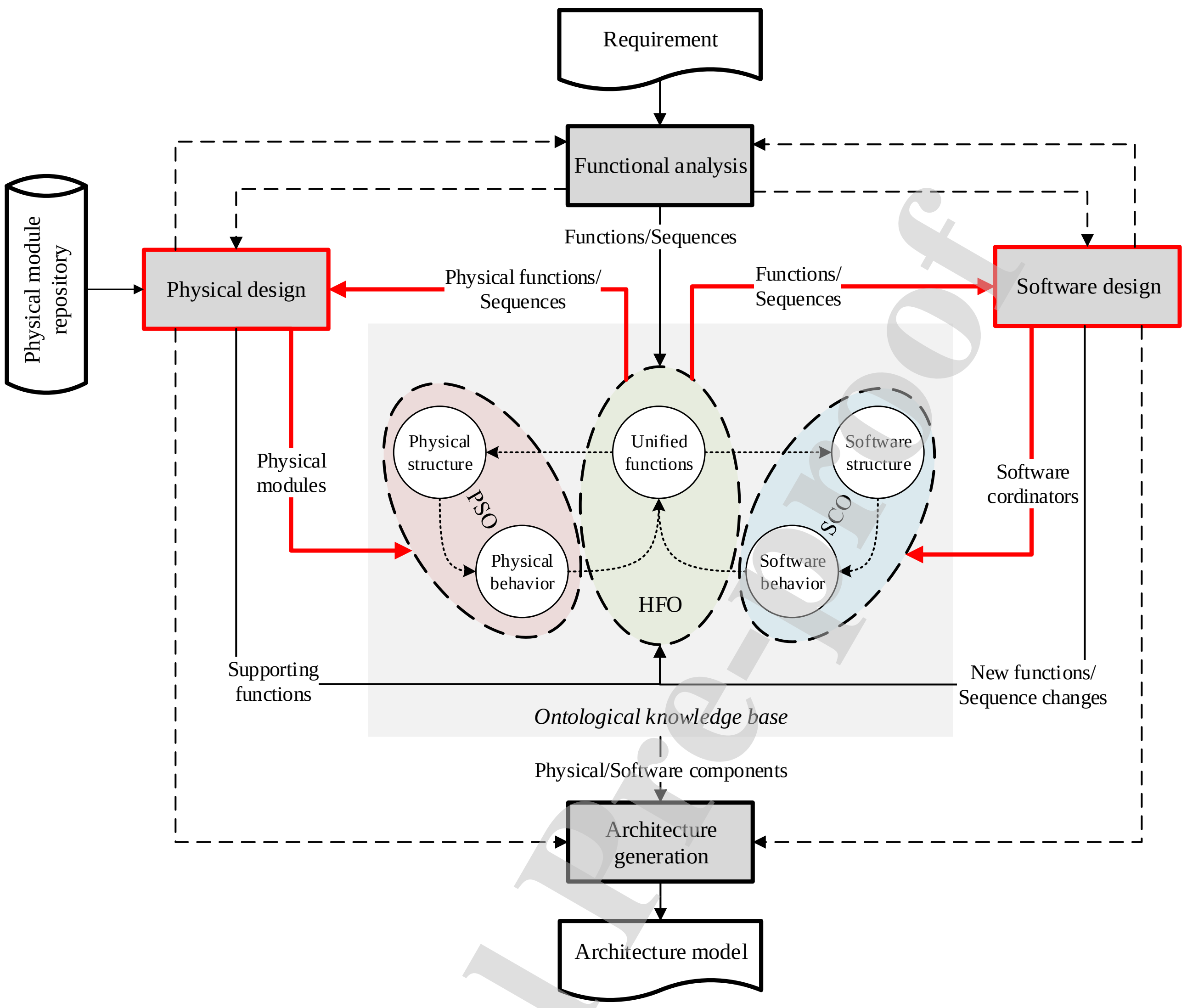


Fig. 2 Semantic web technologies-enabled co-design approach

If a designer starts a design from scratch, the design process mainly includes the following steps:

(1) *Functional analysis*: the designer first extracts the functions from requirements and analyze them until they can be directed used to retrieve physical components. These leaf-level functions and their execution sequences originally represented as graphical models in SysML are transformed into the ontological knowledge base to populate the ABox of the HFO. After that, the physical and software design steps can be started in parallel.

(2) *Physical design*: physical functions that are intended to be implemented by physical structures and their controllers are synthesized into physical modules by retrieving from a physical module repository. These physical modules are imported into the PSO. Execution sequences between functions work as constraints to filter the retrieved modules. Please notice that execution sequences are only one aspect of the criteria for this evaluation from the functional perspective. Other criteria from non-functional perspectives like cost, performance, and reliability should be considered in real practice but not discussed in this study. Besides that, some supporting functions may be introduced in the physical design and updated into the HFO.

(3) *Software design*: software coordinators and their behaviors are generated from the execution sequences between functions and recorded in the SCO. Designers may change the sequences or introduced new functions in this step. For example, designers may change the execution sequence between two actions from in sequence to in parallel to decrease the execution time. These changes should be updated to HFO as well.

It can be seen that Step (2-3) may introduce changes to the functional model. Specifically, new supporting functions may be introduced in the physical design, and execution sequences of functions may be updated in the software design. These changes are solved by triggering step (1-3) iteratively until no functional changes are made. Please notice that changes to the structure or behavior directly such as adding new components manually are not supported yet. It requires designers to manually reflect the changes to the functional knowledge to keep consistency of the knowledge base.

(4) *Architecture generation*: after all the physical modules and software components are generated, they should be grouped together to construct the system architecture.

This study focuses on providing computational support to Step (2-3) which are highlighted in Fig. 2. Ontologies and semantic web technologies are the main enabling technologies. The ontology-based approach has been recognized as an effective enabling technology to support knowledge sharing between different domains [10,54,55]. The basis of the ontology-based knowledge interoperability is that several views of the same artefact are bound to exist, which result in multi-viewpoint models naturally overlapping with each other [56]. Therefore, the key issue of this method is to find correlations between system design and control application design models. Functions as explained in Section 4.1 is identified as the key to correlate the software and physical design knowledge. As the mainstream ontology languages, OWL2 [57] is adopted as the ontology description language of the knowledge base because of its expressiveness and firm mathematical foundation. The formal and unified ontological representation across domains and the capability of automated semantic inference can benefit the implementation of the proposed approach.

### 3.3 Scope discussion

To involve software into the mechatronic system design, its role in the system should be clarified first. The control of mechatronic systems is achieved in several layers [48,58]. The low-level control regulates the continuous physical processes, e.g., closed-loop or open-loop control; the middle-level control coordinates the execution of different continuous processes, e.g., logic and sequence control; the high-level control addresses global management related functions such as planning, monitoring, diagnosing, etc. Since the cooperation of the low-level control and physical systems has been studied extensively and the high-level control has a relatively weak influence on the conceptual design of physical systems, the middle-level control, which is mainly implemented by software programs is the objective to be synergistically designed with the physical part in this study.

Second, among different types of MTSs, this approach is particularly suitable for designing the software-intensive ones with high demand on execution sequences, e.g. systems of manufacturing and robotics domains. However, MTSs that are physical-dominated with less involvement of software, e.g., hair dryer, are not proper objective systems.

Finally, the workflow controlled by software may affect the physical design from multiple aspects. Besides constraints applied by execution sequences, other kinds of influences such as execution time may affect the selection of physical modules. For example, the physical modules have to be cooled down before next round of execution to meet the requirements on the execution time. However, in this study, only the execution sequences are considered.

## 4. Hybrid functional ontology

The common practice of existing function-based co-design methods is that functions of multidisciplinary components are simply connected through flows that represent objects transferred among them. The drawback is that it can only capture relationships between intended interfaces instead of behaviors of them. In this study, a hybrid functional model (HFM) is proposed, which embeds the continuous changes specified by physical functions in the discrete execution sequences organized by software. The rationale of this representation is explained and its related formal definitions are proposed first. Then, the HFO is established to store and process the functional models in an ontological knowledge base.

### 4.1 Rationale and related definitions

Software and physical subsystems of MTSs have different characteristics and thus different design strategies were applied to them respectively. On one hand, system design methods for embedded software such as Harmony SE [59] and OOSEM [60] are normally scenario-driven. Functions of a system are modeled as data/control flow diagrams [53], which show the execution sequences of functions as well as their data interactions. These functions are then allocated to components of the software system such that their interfaces, operations, and statecharts can be generated. On the other hand, the conceptual design of multi-physics systems is driven by physical status changes. Functions are modeled as expected status changes to physical objects. Then physical structures together with their low-level controllers are explored and determined to realize that changes through continuous physical processes [15].

Although these two types of design look quite divergent, an essential semantic correlation can be revealed on the functional-level, i.e., each step in a scenario may "trigger" functions of physical systems to be implemented. For example, functions of an autonomous vehicle involve complex execution sequences of *move* and *stop* steps; these two steps are ultimately realized by physical structures such as wheels and brakes. This idea is quite similar to that of hybrid automata [61], whose basic idea is that discrete behaviors of software are represented by states and transitions between them, whereas continuous behaviors of multi-physics modeled by differential-algebraic equations are embedded in each state. This approach for abstracting a system is so-called "cyberizing the physical" [62]. Besides that, the flow-based structure of both types of functions provides the syntax-level common basis to unify them.

According to the above rationale, the HFM is proposed by embedding physical functions in the flow-based functional representation in a data/control flow diagram. The data/control flow diagram describes the workflow of the system. The steps in the workflow trigger different types of functions to achieve concrete physical status changes.

***Definition 1***: A *hybrid functional model* (HFM) is a formal definition of functions of a system that requires a joint effort of software and physical components to be implemented. It is defined as a graph $HFM = (A, P, E, \delta, \varphi, \tau)$ where:

- $A = \{a_1, a_2, \ldots, a_n\}$ is a set of actions (steps) in the workflow of the system.
- $P = \{p_1, p_2, \ldots, p_n\}$ is a set of parameters. They can be input, output, or internal parameters of the system.
- $E = \{e_1, e_2, \ldots, e_n\}$ is a set of flows connecting actions or parameters.
- $\delta: (A \cup P) \times E \to (A \cup P)$ is a function that denotes flows between actions and parameters. Actions are connected through their *Pins*, which are instances of interfaces of the functions they trigger.
- $\varphi: A \to (PF \cup SF \cup CN)$ is a function that assigns different types of concrete functions to actions that trigger them. *PF*, *SF*, and *CN* stand for *physical functions*, s*oftware functions*, and *control nodes* respectively, which will be defined later.
- $\tau: (P \cup E) \to FT$ is a function that assigns the types to parameters and flows. *FT* stands for flow types, which will be defined later.

***Definition 2***: A *physical function* specifies a physical process that changes the status of flows. It is triggered by actions in the HFM and implemented by multi-physics components controlled by their controllers. It is defined as a tuple $PF = (U, V, \Sigma, \omega, \tau)$ where:

- *U* and *V* are sets of input and output flow objects of functions. They are interfaces of functions, through which functions can be connected with each other by flows.
- $\Sigma$ is a set of effects performed by functions. An effect is an observable objective change on the input flow objects, e.g., the *temperature* of a flow *gas* is changed from *low* to *high*; the type of a flow of *gas* is changed to *liquid*. It is further defined in Section 4.3.
- $\omega: U \times \Sigma \to V$ specifies the effects consuming inputs to generate certain outputs.
- $\tau: (P \cup E) \to FT$ is a function that specifies the flow types of *U* and *V*.

***Definition 3***: A *software function* specifies a computational process that implements high-level planning algorithms. It is triggered by actions in the HFM and requires software and hardware (electronic components) to implement. It is defined as a tuple $SF = (U, V, \Sigma, \omega, \tau)$.

Definitions of elements in the tuple are similar as physical functions except $\Sigma$. Because of the complexity and variety of software programs [63], it is difficult to define a common formal abstraction for the expected operations performed by software functions as for physical functions in Section 4.3. For simplicity, they are represented by verbs reflecting their purposes.

***Definition 4***: *Control nodes* specify complex execution sequences of actions in the HFM. Typical control nodes are *fork*/*join* for concurrent execution and *decision*/*merge* for optional execution.

***Definition 5***: *Flow types* specify the types of flow objects or parameters. Flow types are differentiated to *object* and *control*.

The de facto flow classification used for physical functional modeling is summarized in the Functional Basis [16], in which three primary classes, i.e., *material*, *energy*, and *signal* are defined. Meanwhile, in software design, *data* and *control* flows are used for passing data exchanged between actions and control tokens for activating actions. The two classifications are merged to *object* and *control* flows as what has been done in SysML [64]. *Objects* can be classified as *material*, *energy* or *signal* when they are expected to be processed by physical components, and *data* for software components.

## 4.2 Hybrid functional ontology (HFO)

To represent and store functional models in a computer-readable way, the HFO is developed and defined in OWL2 DL on the basis of the Basic Formal Ontology (BFO), which is a small upper-level ontology to support the creation of lower-level domain ontologies. The concepts in above formal definitions are represented as OWL classes and their data/object properties. The core terminology of the HFO is shown in Fig. 3. These classes and relations constitute the TBox of the knowledge base, whereas the functional models created by designers are represented as individuals of them, which populate the ABox.

The core of HFO are two types of functions, both inherited from *Function* in BFO. *Hybrid functions* are complex functions that require a joint effort of software and physical components to implement. The semantics of each hybrid function is described by a *workflow*. *Workflow* is inherited from *Process* in BFO and specifies how the hybrid function is expected to be implemented. It is divided into several sub-processes represented by *Node*. Nodes can be *Control nodes* for execution sequence control purpose or *Actions* that trigger basic functions. Nodes are connected by *Flows* to specify execution sequences or objects transferred between each other. *Basic functions* are the atomic functions that can be implemented by either physical or software components. Its semantics is explained in the following section. In essence, hybrid functions can be treated as composite functions composed of multiple basic functions. However, because a basic function may be triggered multiple times in a hybrid function, to describe the concrete processes in detail, concepts such as *Workflow* and *Action* are proposed. The inputs and outputs of functions are represented by *Parameter*. They are instances of *FlowObjects* including *Material*, *Energy*, and etc. which are inherited from *Material Entity* or *Generally Dependent Continuant* in BFO.

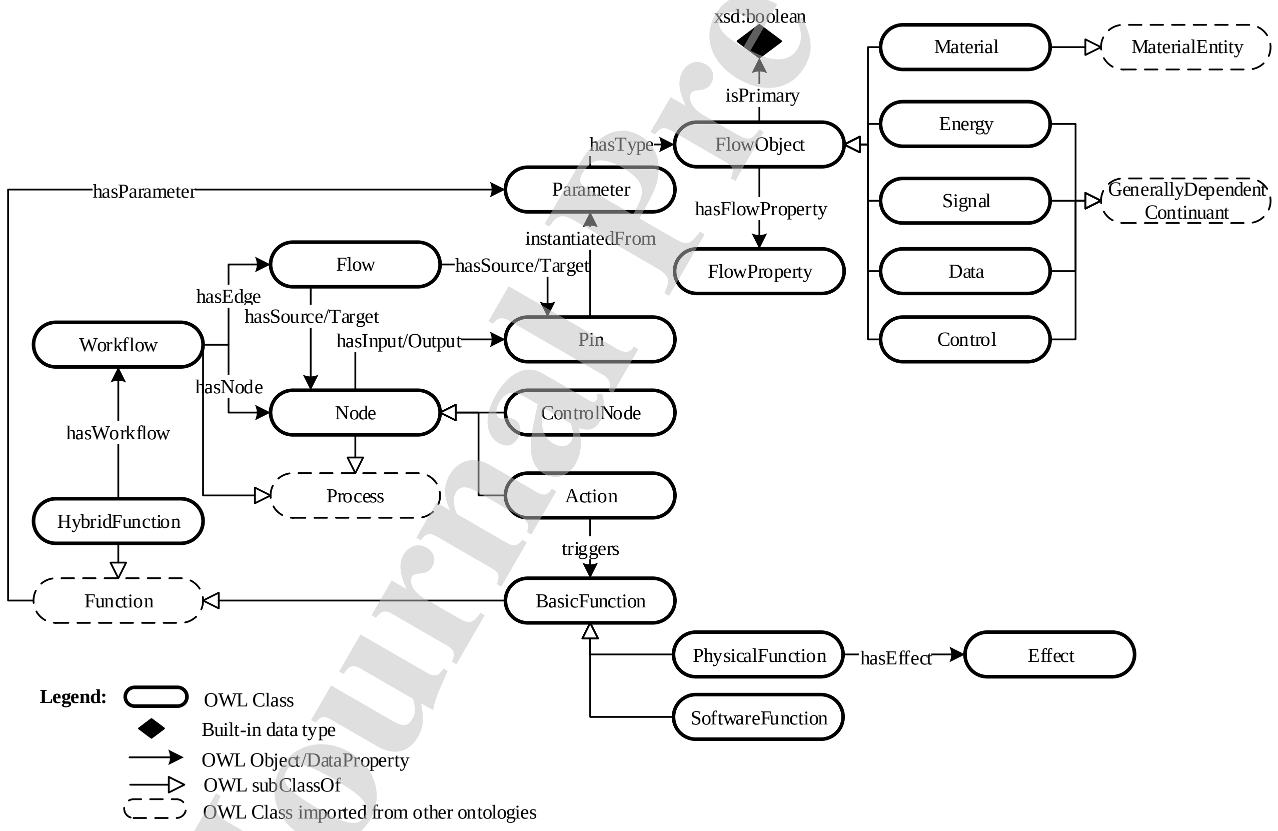


Fig. 3 Core terminology of HFO

These concepts should be defined formally to automatically classify individuals and keep consistency of the knowledge base. The rich primitives provided by OWL2 [57] are leveraged to solve this problem. *Class expressions* describe sets of individuals by formally specifying conditions on the individuals' properties. They are necessary and sufficient conditions to define a class. OWL classes can be set as equivalent to class expressions to build *class axioms* which can be consumed by semantic web reasoners to automatically determine whether the facts described by individuals of classes are consistent or which classes an individual belongs to. Table1 shows the axioms defined for different classes of functions. The *HybridFunction* is characterized by its workflow which has at least one action that triggers physical functions. The *BasicFunction* is the union of *PhysicalFunction* and *SoftwareFunction*. The *PhysicalFunction* and *SoftwareFunction* are differentiated by the modules they are allocated to. The disjoint relationships are set between *HybridFunction* and *BasicFunction* and between *PhysicalFunction* and *SoftwareFunction*.

Table1. Sample axioms for function classification

| **Class** | **Class expression** |
|---|---|
| HybridFunction | Function and (hasWorkflow some (Workflow and (hasNode some (Action and (triggers some PhysicalFunction))))) |
| BasicFunction | PhysicalFunction or SoftwareFunction |
| PhysicalFunction | Function and (triggeredBy some (Action and (allocatedTo some (ModuleInstance and (instantiatedFrom some PhysicalModule))))) |
| SoftwareFunction | Function and (triggeredBy some (Action and (allocatedTo some (ModuleInstance and (instantiatedFrom some SoftwareModule))))) |

## 4.3 Functional effect (FE)

In the formal definition of the physical function, *effects* can be further defined. Since they are mainly used for retrieving working principles, their concrete description depends on the used specific conceptual design methods. In the conceptual design methods that do not support automated retrieval, the simplest method to represent effects of functions is using verbs, which are standardized in the functional basis [16]. However, since this representation cannot be comprehended by computers, the semantics of verbs are further detailed by some other methods, such as constraints on flows [19], functional effects (FEs) [65] to better represent effects in a machine-understandable way. In the former method, working principles are synthesized in a simulation-based method that the solutions in the knowledge base are compared with the constraints of the required functions. In the latter method, each of the working principles in the knowledge base is index by the functional effect that can be provided by it, so that it can be retrieved by matching the required and provided FEs.

In this study, the FE-based method is chosen because of its capability of automated functional decomposition [65,66] and architecture synthesis [67]. The corresponding ontology of FE is shown in Fig. 4. In general, four kinds of changes of flows are defined according to the functional basis. Each kind of changes is further detailed to reflect their semantics. For example, the FE of a function "to heat gas from 0 to 20 degrees centigrade" is a *value change*, *gas* is the flow object to be manipulated; *temperature* is the property to be changed; the value of this property is changed from 0 to 20; the change trend is *increase*. Please refer to [65,66] for more details about FEs.

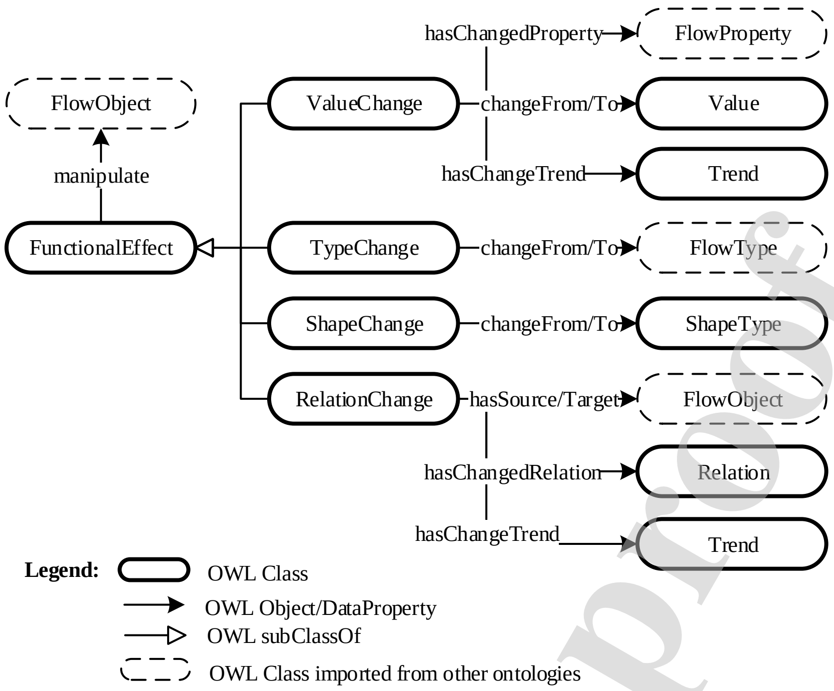


Fig. 4 Terminology for describing FE

Similar to functions, the different classes of FEs are also formally defined for the automated classification of individuals. *ValueChange* is the subclass of *FunctionalEffect* that changes *Value*. *TypeChange* is the subclass of *FunctionalEffect* that changes *FlowType*. *ShapeChange* is the subclass of *FunctionalEffect* that changes *ShapeType*. *RelationChange* is the subclass of *FunctionalEffect* that changes the *Relation* between *FlowObject*. These classes are specified as disjoint with each other.

Table2. Sample axioms for functional effect classification

| **Class** | **Class expression** |
|---|---|
| ValueChange | FunctionalEffect and (changeFrom some Value) and (changeTo some Value) |
| TypeChange | FunctionalEffect and (changeFrom some FlowType) and (changeTo some FlowType) |
| ShapeChange | FunctionalEffect and (changeFrom some ShapeType) and (changeTo some ShapeType) |
| RelationChange | FunctionalEffect and (hasSource some FlowObject) and (hasTarget some FlowObject) and (hasChangedRelation some Relation) |

To illustrate the functional modeling process, the mobile robot is taken as an example. Its HFM is graphically modeled as shown in Fig. 5. SysML is adopted as a possible choice to facilitate users to create HFMs graphically since the definition of HFM are quite close to the abstract syntax of activity diagram (ACT) of SysML. In fact, the HFM can be treated as domain-specific customization of ACTs. Specifically, *Activity* can represent different types of functions; *Parameters* of an *activity* represent flow objects of a function; *Actions* in SysML that are instantiated from *activities* indicate actions in HFM; *Control nodes*, *object flows* (full lines in which red lines show the energy flows, pink lines show the data flows, and blue lines show the material flows), *control flows* (dashed lines), and *pin* in SysML are counterparts of the similar concepts in HFM. In this figure, it shows that the mobile robot will start to detect the position and color of the obstacles around it, and then send data to determine the next steps. According to the determined direction (WallDir) and position of the can (CanDir), 7 paths can be selected. For example, the first path including two actions TurnLeft and MoveForward means that the mobile robot should go to the left square beside it.

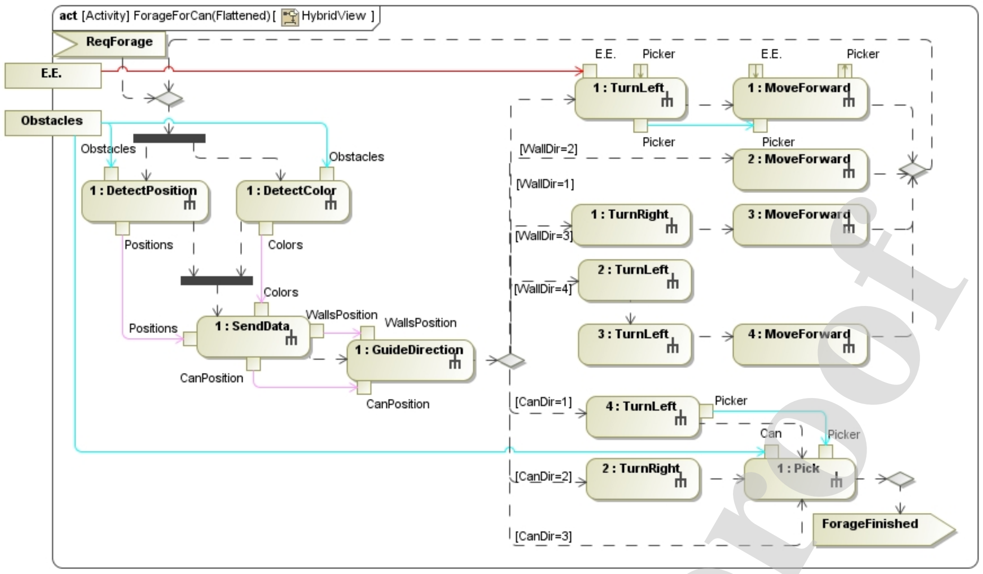


Fig. 5 HFM of *Mobile robot*

However, since SysML is a domain-independent modeling language, no specific model elements are provided to support modeling the domain-specific concept FE. For example, using the general construct Block, designers cannot distinguish different types of FEs or specify their properties. To solve this problem, the extensibility mechanisms of UML/SysML, i.e., stereotypes and tags are leveraged. They can be used to create new model elements that are suitable for particular domains. Different kinds of FEs are defined as stereotypes whose tags represent the properties of FEs. Please refer to [65] for details. For example, the FE of *MoveForward* is modeled as a stereotyped *block* in SysML as shown in Fig. 6. It is a value change as indicated by its stereotype. The flow to be manipulated is *Picker*; its property *Position* is to be changed; the change trend is *Increase*. They are represented by corresponding tags.

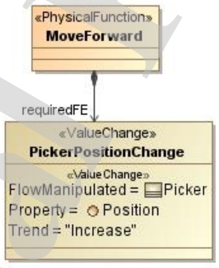
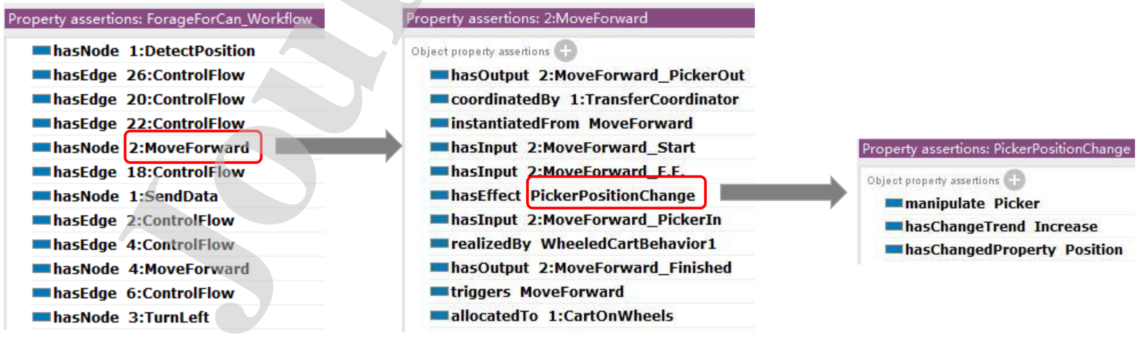


Fig. 6 FE of *MoveForward*

From the graphical model in SysML, facts can be automatically extracted and populated into the knowledge base. An excerpt of the facts about the HFM of the mobile robot is shown in Fig. 7.


Property assertions: ForageForCan_Workflow

hasNode 1:DetectPosition
hasEdge 26:ControlFlow
hasEdge 20:ControlFlow
hasEdge 22:ControlFlow
hasNode 2:MoveForward
hasEdge 18:ControlFlow
hasNode 1:SendData
hasEdge 2:ControlFlow
hasEdge 4:ControlFlow
hasNode 4:MoveForward
hasEdge 6:ControlFlow
hasNode 3:TurnLeft

Property assertions: 2:MoveForward

Object property assertions

hasOutput 2:MoveForward_PickerOut
coordinatedBy 1:TransferCoordinator
instantiatedFrom MoveForward
hasInput 2:MoveForward_Start
hasInput 2:MoveForward_E.E.
hasEffect PickerPositionChange
hasInput 2:MoveForward_PickerIn
realizedBy WheeledCartBehavior1
hasOutput 2:MoveForward_Finished
triggers MoveForward
allocatedTo 1:CartOnWheels

Property assertions: PickerPositionChange

Object property assertions

manipulate Picker
hasChangeTrend Increase
hasChangedProperty Position

Fig. 7 Excerpt of facts of HFO

## 5. Automated evaluation of physical modules by execution sequences

The main task of physical design is to choose multi-physical mechanisms and their continuous controllers to implement physical functions. When a physical module only implements a single function, it can be determined by simply comparing its provided behavior and the required behavior of the function. However, when a physical module implements multiple functions, the module should satisfy the execution sequences of these functions. Traditional physical design approaches have not considered such constraints and hence may generate incorrect solutions. To solve this problem, the description of physical structures are enhanced first. The enhanced physical structures are related with functions in a common knowledge base and hence can be automatically evaluated against the functional knowledge.

### 5.1 Physical structure ontology (PSO)

It is straightforward to link the physical structures with the functions they implement directly. However, such mappings are too inflexible and cannot reflect the essential causal relationship for reasoning. Therefore, instead, the mappings between them are constructed through their behaviors. Specifically, functions specify their expected behaviors, and structures realize functions through their provided behaviors. The mapping between functions and physical structures can be constructed by matching the behaviors. *Behavior* is defined as an OWL class which means the realization of the functions in the structure. It can be differentiated into two sub-classes. *PhysicalBehavior* is the behavior of physical structures, which normally is continuous and described by differential-algebraic equations. *SoftwareBehavior* is the behavior of software components, which normally is discrete and described by state machines. The formal definitions of these classes are illustrated in Table3.

Table3. Sample axioms for behavior classification

| Class | Class expression |
|---|---|
| PhysicalBehavior | Behavior and (hasEquation some Equation) |
| SoftwareBehavior | Behavior and (hasState some State) and (hasTransition some Transition) |
| Behavior | PhysicalBehavior or SoftwareBehavior |

However, besides the conventional definition of functions and behaviors, the execution sequences between functions are considered in this study. Therefore, to determine whether a physical module can satisfy the execution sequences of functions, the *execution relation* between its behaviors should also be described. For example, a *wheeled cart with Ackermann steering* have to move and turn dependently; a *wheeled cart with differential drive* can move and turn independently but at different times; *cart and rotating shaft driven by gears* can move and rotate independently and concurrently.

Another observation is that the mappings between function and physical structure actually exist between their instances instead of themselves. A function can be triggered by multiple actions under different circumstances, which may be realized by different physical structures. For example, the *stop* function of a vehicle may be triggered while it is running or when it is stopped. The two actions are realized by *handbrake* and *footbrake* respectively. Similarly, a physical module can be instantiated to multiple components of a system to implement different functions. For example, a *rotating shaft* can constitute the transferring subsystem to rotate a pipe, or the bending subsystem to bend a pipe.

Based on the above two observations, the PSO is defined as shown in Fig. 8.

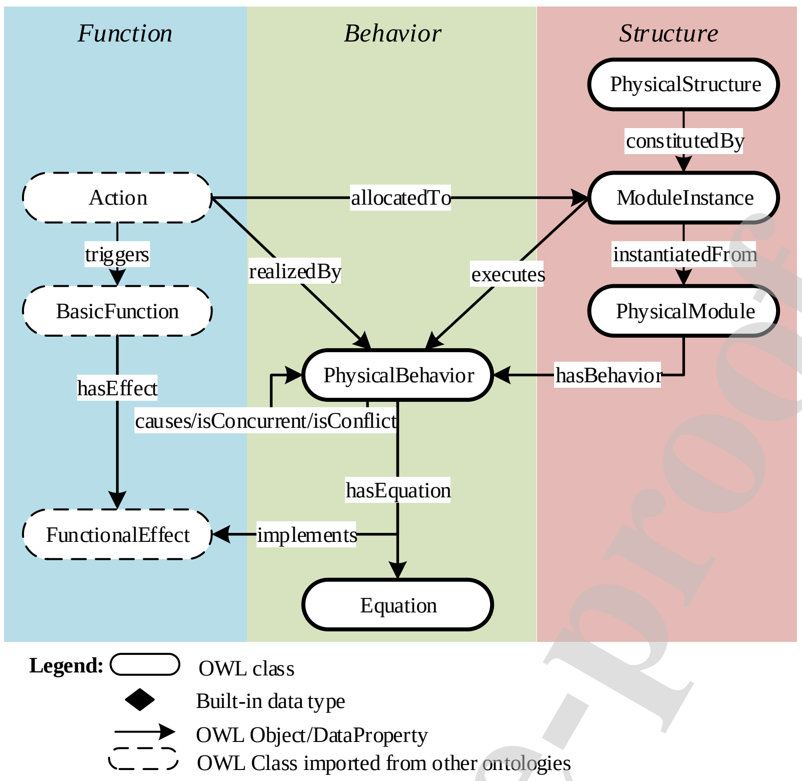


Fig. 8 Core terminology of PSO

The execution relations between behaviors of physical modules are explicitly represented as object properties of *physical behavior*. Several object properties are defined to describe the execution relations between two behaviors. *causes* indicates that one behavior causes another because of the physical structure of the module. *isIndependent* means that the two behaviors can be separately controlled. It is further specialized to *isConcurrent* and *isConflict* to indicate whether the two separately controlled behaviors can be triggered at the same time.

**5.2 *Module instance* represents instances of physical modules that constitute the physical structure. The mappings between function and structure are established between *actions* and *module instances*. Specifically, actions are *realized by* physical behaviors and hence are *allocated to* module instances *executing* those behaviors.Automated evaluation of physical modules**

With the linked functional and structural knowledge, it is possible to choose the suitable physical modules by functional constraints automatically. Here, the challenge is how to define the rules for choosing and how to represent the rules as axioms that can be processed by semantic web reasoners such as Pellet [68] used in this study.

### 5.2.1 Rule template

The rules for evaluating modules according to execution sequences of functions can be various. For example, if two functions run sequentially, one function should not be caused when the module behaves to implement another; if two functions run in parallel, the module implementing them should be able to accomplish the effects of both functions at the same time. However, they follow the same pattern as follows. When a module instance $m_k$ realizes any two actions $a_i$ and $a_j$ via its behaviors $pb_i$ and $pb_j$, $m_k$ is determined to be unsatisfactory if the execution relations between $pb_i$ and $pb_j$ are not compliant with the execution sequences between $a_i$ and $a_j$. A rule template can be abstracted and defined as an implication $AR \wedge l_1 \wedge l_2 \wedge r_1 \wedge r_2 \wedge e_1 \wedge e_2 \wedge BR \rightarrow s$, where,

- $AR = \{ar_1, ar_2, \ldots, ar_n\}$ is a set of propositions declaring the relationship $r_a$ between two actions $a_i$ and $a_j$. It can be a direct relationship, e.g., their execution sequence, or complex relationship between their properties;
- $l_1$ and $l_2$ are propositions declaring that the two actions $a_i$ and $a_j$ are allocated to the module instance $m_k$;
- $r_1$ and $r_2$ are propositions declaring the two actions $a_i$ and $a_j$ are realized by the two physical behaviors $pb_i$ and $pb_j$;
- $e_1$ and $e_2$ are propositions declaring the two physical behaviors $pb_i$ and $pb_j$ are executed by the module instance $m_k$;
- $BR = \{br_1, br_2, \ldots, br_n\}$ is a set of proposition declaring the relationship $r_b$ between the two physical behaviors $pb_i$ and $pb_j$. It can be a direct relationship, e.g., their execution relation, or complex relationship between their properties;

- $s$ is a proposition indicating whether the module instance $m_k$ can satisfy the required actions.

The rule template can be instantiated by varying *AR* and *BR*. For example, the relationships can be set as *inSequence* and *causes* respectively to represent the Rule 1 illustrated in the next section.

#### 5.2.2 Rule representation in SWRL

Evaluation rules customized from the rule template should be represented as axioms acceptable by semantic web reasoners. With these axioms, semantic web reasoners can infer whether the physical structure can satisfy functions based on the existing knowledge. SWRL can specify rules in an implication form and hence is a proper rule language to describe filter rules. All the propositions involved in the rules are represented as atoms of the form of $P(x, y)$, where *P* is a data/object property in the knowledge base. For example, the sample rule 1can be defined as SWRL Rule 1.

**Rule 1: Inferring *isUnsatisfactory* of physical modules by *inSequence***

inSequence(?$a_i$, ?$a_j$) ^ allocatedTo (?$a_i$, ?$m_k$) ^ allocatedTo(?$a_j$, ?$m_k$) ^ executes (?$m_k$, ?$b_i$) ^ executes (?$m_k$, ?$b_j$) ^ realizedBy(?$a_i$, ?$b_i$) ^ realizedBy (?$a_j$, ?$b_j$) ^ causes(?$b_i$, ?$b_j$)
-> isUnSatisfactory(?$m_k$, *true*)

Note that the atom inSequence(?$a_i$, ?$a_j$) in the antecedent of Rule 1 refers to an object property *inSequence* whose related facts are not populated. Therefore, this knowledge should be inferred by reasoners based on the existing knowledge. This is the purpose of SWRL Rule 2.

**Rule 2: Inferring *inSequence* between actions**

ControlFlow(?*f*) ^ hasSource(?*f*, ?$pin_s$) ^ hasTarget(?*f*, ?$pin_t$) ^ Action(?$a_i$) ^ Action(?$a_j$) ^ hasOutput(?$a_i$, ? $pin_s$) ^ hasInput(?$a_j$, ? $pin_t$)
-> inSequence(?$a_i$, ?$a_j$)

To illustrate the physical module evaluation process, two functions *MoveForward* and *TurnLeft* are taken as examples. Several physical modules can be retrieved to realize these functions. For example, *WheeledCartWithAckerMannSteering* can be retrieved following the FE-based retrieval process described in [67] as illustrated in Fig. 9. Specifically, each physical module stored in an ontological repository has a property *providedFE* to document the FEs that can be provided by it. These FEs work as the index of physical modules and are normally defined in a more general form. The instances shown in Fig. 9 follow the terminology of ValueChange illustrated in Fig. 4. For example, the *WheeledCartWithAckerMannSteering* can provide the FE *ValueChange4*, which can increase or decrease the position of the translational 2-dimentional objects. On the other hand, the function to be implemented also defines its required FEs, which are normally in a more specific form. For example, the function *MoveForward* requires the FE *PickerPositionChange*, which increases the position of the picker. During retrieval, each of the FEs required by the functions will be matched with the *providedFE* of the physical modules. If all the properties of the required FEs of a function are equal to or subclasses of the properties of the provided FEs of a physical module, then the physical module can be retrieved.

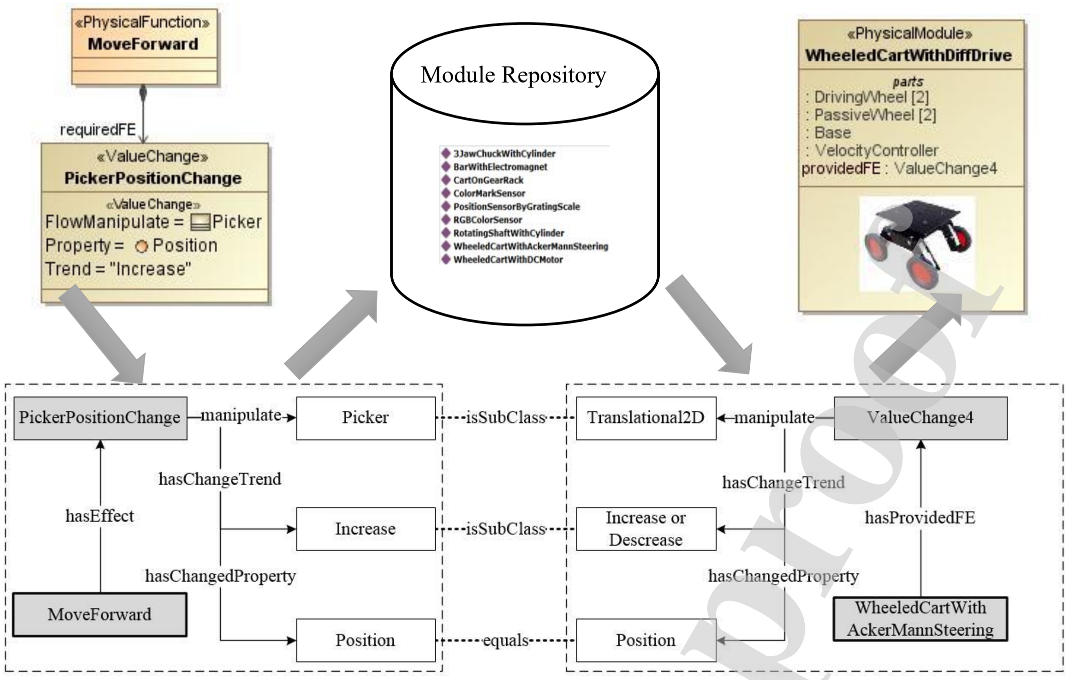
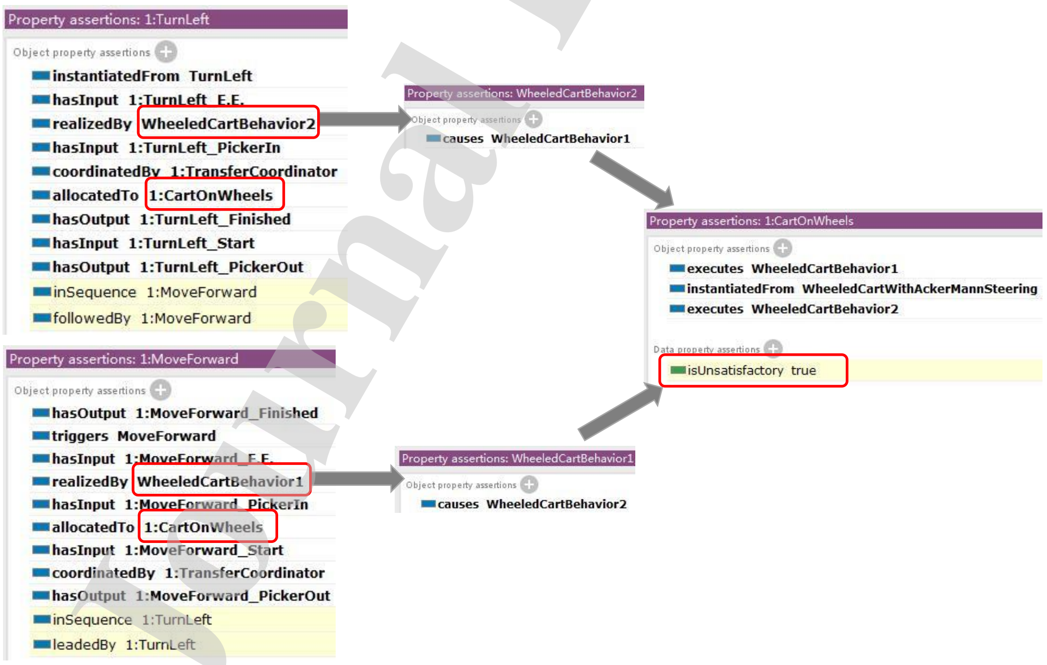


Fig. 9 Module retrieval process

The retrieved physical modules are instantiated to module instances to constitute the system and implement actions in the functional model. For example, designers create *1:CartOnWheels* from *WheeledCartWithAckerMannSteering* to implement actions such as *1:TurnLeft* and *1:MoveForward*. This information is stored in the knowledge base as shown in Fig. 10. The two actions are realized by two behaviors *WheeledCartBehavior1* and *WheeledCartBehavior2*. Since the two actions are executed in sequence whereas the two behaviors occur at the same time, semantic web reasoners can determine that the module instance cannot satisfy the required functions.



Property assertions: 1:TurnLeft

Object property assertions

- instantiatedFrom TurnLeft
- hasInput 1:TurnLeft_F.E.
- realizedBy WheeledCartBehavior2
- hasInput 1:TurnLeft_PickerIn
- coordinatedBy 1:TransferCoordinator
- allocatedTo 1:CartOnWheels
- hasOutput 1:TurnLeft_Finished
- hasInput 1:TurnLeft_Start
- hasOutput 1:TurnLeft_PickerOut
- inSequence 1:MoveForward
- followedBy 1:MoveForward

Property assertions: WheeledCartBehavior2

Object property assertions

- causes WheeledCartBehavior1

Property assertions: 1:CartOnWheels

Object property assertions

- executes WheeledCartBehavior1
- instantiatedFrom WheeledCartWithAckerMannSteering
- executes WheeledCartBehavior2

Data property assertions

- isUnsatisfactory true

Property assertions: 1:MoveForward

Object property assertions

- hasOutput 1:MoveForward_Finished
- triggers MoveForward
- hasInput 1:MoveForward_F.E.
- realizedBy WheeledCartBehavior1
- hasInput 1:MoveForward_PickerIn
- allocatedTo 1:CartOnWheels
- hasInput 1:MoveForward_Start
- coordinatedBy 1:TransferCoordinator
- hasOutput 1:MoveForward_PickerOut
- inSequence 1:TurnLeft
- leadedBy 1:TurnLeft

Property assertions: WheeledCartBehavior1

Object property assertions

- causes WheeledCartBehavior2

Fig. 10 Excerpt of facts of PSO

## 6. Automated software behavior generation and updating

Software coordinators organize different physical modules to achieve the main function of the target system according to its workflow. Therefore, their design is closely related to physical functions. To keep consistency between software design and physical functions, the software coordinators should be correlated with the functional model in the knowledge base. Based on correlations between them, the behaviors of software coordinators can be automatically generated and updated.

## 6.1 Software coordinator ontology (SCO)

Similar to PSO, software can be correlated with functions through their behaviors. Here, the execution semantics of behaviors of software coordinators is analyzed first.

As the functions of systems become more and more complex, the intelligence of systems is required to be decentralized and embedded into several software components [69]. Therefore, the software subsystem of an MTS should be constituted by several software coordinators, each of which coordinates a piece of the workflow of the whole system. To achieve the actions included in its allocated piece of workflow, a software coordinator needs to send requests to the low-level controllers of the physical modules that realize these actions. Once an action is finished, the low-level controller will send feedbacks to the high-level coordinator so that the coordinator will switch to the next action in the workflow. For example, in the workflow of the mobile robot, a piece of workflow including *1:TurnLeft* and *1:MoveForward* are coordinated by *TransferCoordinator*. To achieve the two actions, a request will be sent to the controller of *CartOnWheels* to activate its behavior to implement *TurnLeft*. When the operation is finished, a request will be sent back to the coordinator to switch to the next state, in which the next action *1:MoveForward* is activated.

Based on above semantics, the behavior model of a software coordinator can be abstracted as a Finite State Machine, which is formally defined as $SBM = (R_{in}, R_{out}, S, s_0, s_n, \delta, \omega)$, where,

- $R_{in}$ is a set of requests with conditions accepted by the software coordinator from the modules it coordinates or other software coordinators.
- $R_{out}$ is a set of requests sent out from the software coordinator to the modules it coordinates or other software coordinators.
- $S$ is a set of states of the software coordinator, during which it waits for the required actions to finish.
- $s_0$ and $s_n$ are the initial and final states respectively.
- $\delta: S \times R_{in} \rightarrow S$ is a function that specifies the transitions between states triggered by input requests.
- $\omega: S \times R_{in} \rightarrow R_{out}$ is a function that specifies the output requests can be generated in each transition.

These concepts are represented in the SCO as shown in Fig. 11 to describe software coordinators in the knowledge base. Software coordinators are correlated with functions from two aspects: (1) actions are coordinated by software coordinators as indicated by the object property *coordinatedBy*; (2) actions are executed during the states of software coordinators as indicated by *realizedDuring*.

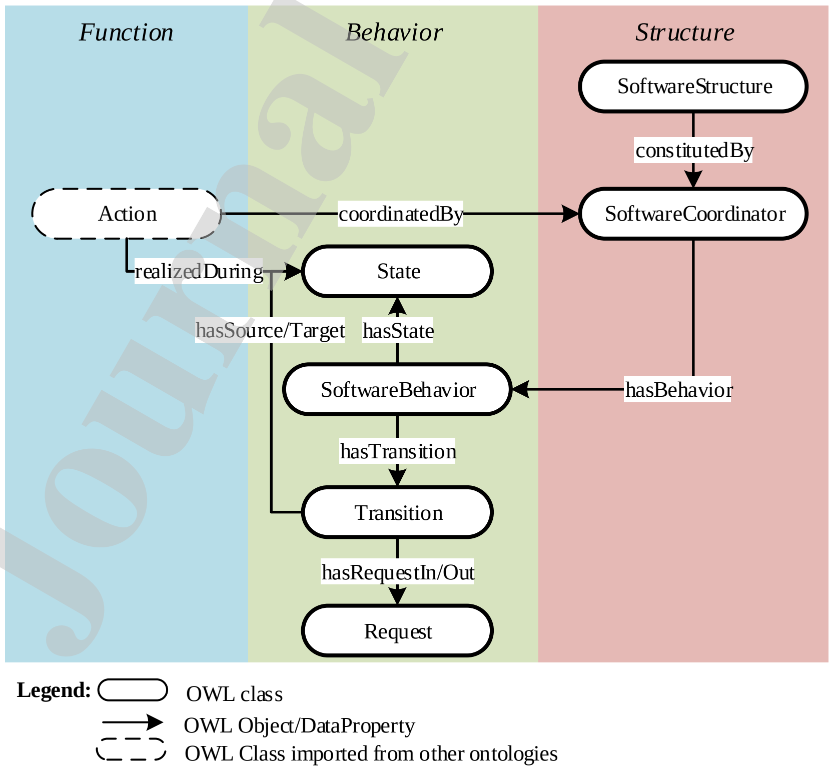


Fig. 11 Core terminology of SCO

## 6.2 Automated software behavior generation algorithm

It can be seen from the ontology that there exist correspondences between actions allocated to a software coordinator and states of the behavior model of this software coordinator. It offers a possibility to develop an algorithm to automatically generate the behavior model from the functional knowledge.

Here, the prerequisite is that, for the software coordinator *sc* whose behavior is to be generated, actions allocated to it are specified by designers. To distribute actions to software coordinators, several aspects such as functional and structural characteristics should be considered. For example, in the mobile robot system, actions triggering *DetectPosition*, *DetectColor* and *SendData* are grouped together from the functional perspective since they are used for the same purpose, i.e., environment detection; actions triggering *TurnLeft*, *TurnRight*, and *MoveForward* are grouped together from the structural perspective since they are realized by the same physical mechanism.

The pseudo-code of the behavior generation algorithm is given in Algorithm 1 and described as follows:

[1] The actions allocated to *sc* are mapped to the states of its behavior model (Algorithm 1 line 9-11)

[2] The control flows among these states are mapped to transitions among these states (Algorithm 1 line 12-16).

[3] Special attention needs to be paid to control flows crossing the boundary of *sc*. For those incoming from actions allocated to other software coordinators, i.e., $E_{input}$, they should be mapped to transitions started from the initial state.

- If there is only one such control flow, it means this flow is the unique entry point of the piece of workflow coordinated by *sc*. Therefore, the request triggering the corresponding transition is named as the same name as *sc* (Algorithm 1 line 17-21). For example, the request to trigger the workflow coordinated by *TransferCoordinator* is named as *reqTransfer*.
- If there are multiple such control flows, they should be mapped to transitions connected to the initial state via a *choice* node. Requests triggering these transitions should be manually specified by designers according to the semantics of the paths (Algorithm 1 line 22-29).

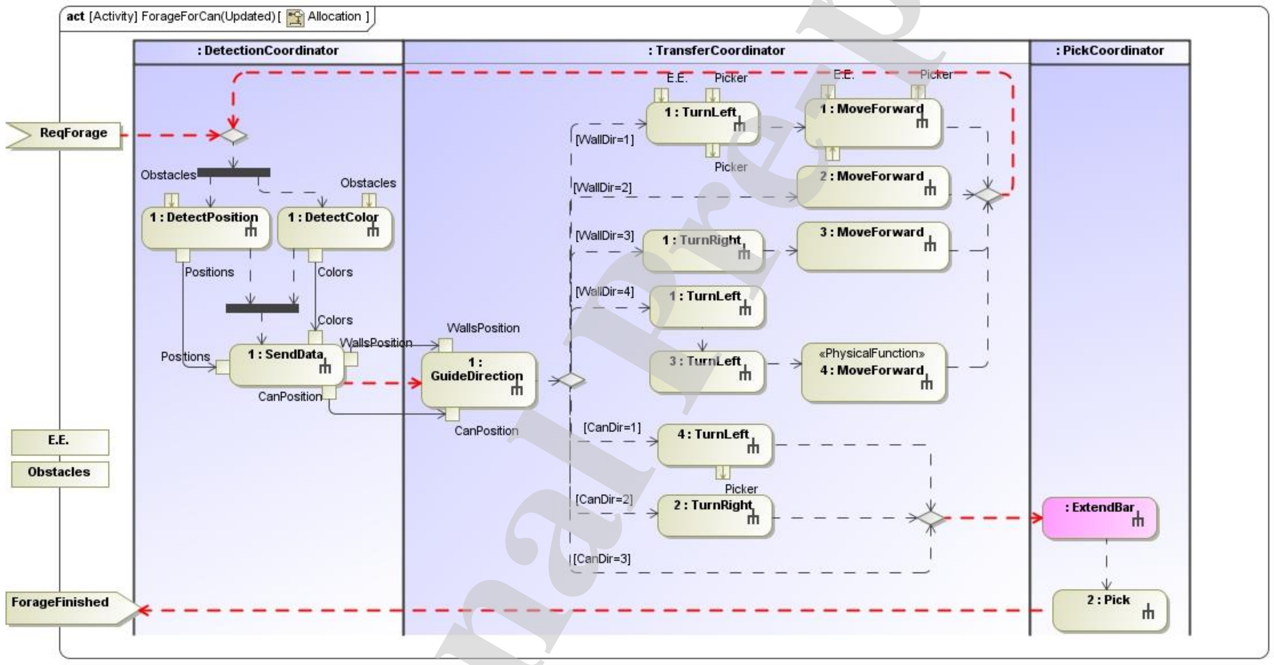
[4] Control flows outgoing from *sc*, i.e., $E_{output}$ are transformed similarly.

**Algorithm 1**: Automated software behavior model generation algorithm

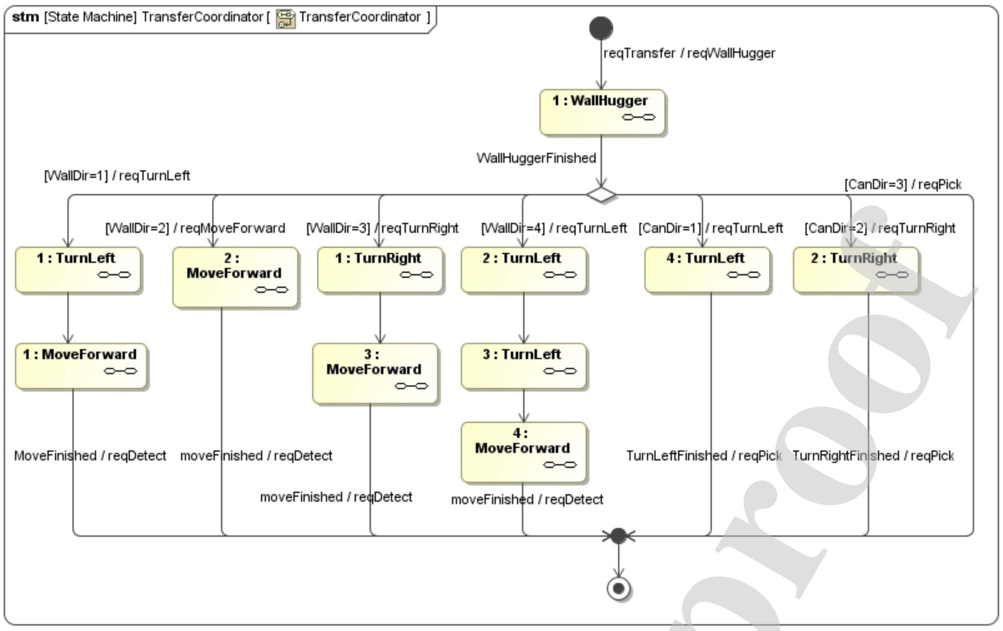
```
Input: The hybrid functional model: $HFM = (A, P, E, \delta, \varphi, \tau)$
Input: A software coordinator sc whose behavior model is to be generated; Actions coordinated by sc $A_{sc} = \{a_i | a_i \in A \land a_i.coordinatedBy = sc\}$
Output: The generated behavior model of sc: $SBM = (R_{in}, R_{out}, S, s_0, s_n, \delta, \omega)$
1  $A_{other} = A - A_{sc}$ // define and populate the set of actions that are not coordinated by sc
2  $E_{inner} = \{e_i | \tau(e_i) = control \land s(e_i) \in A_{sc} \land t(e_i) \in A_{sc}\}$, where $s(e)$ and $t(e)$ are functions that get the source and
3  target actions connected by e respectively // define and populate the internal flows connecting actions coordinated by sc
4  $E_{input} = \{e_i | \tau(e_i) = control \land s(e_i) \in A_{other} \land t(e_i) \in A_{sc}\}$ // define and populate the incoming flows of sc that start from
5  other software coordinators
6  $E_{output} = \{e_i | \tau(e_i) = control \land s(e_i) \in A_{sc} \land t(e_i) \in A_{other}\}$ // define and populate the outgoing flows of sc that point to
7  other software coordinators
8  $R_{in} = R_{out} = \emptyset$ // define the requests received and sent by sc
9  $S = \emptyset$ // define the set of states of sc
10 $Map: A_{sc} \to S$ // define a mapping from the actions coordinated by sc to its states
11 // populate Map and S
12 foreach $a_i \in A_{sc}$ do
13     $S = S \cup \{s_i\}$ where $s_i = translate(a_i)$
14     $Map(a_i) = s_i$
15 // map the internal control flows of sc to its transitions
16 foreach $e_i \in E_{inner}$ do
17     $R_{in} = R_{in} \cup \{r_{in}\}$ where $r_{in} = request(s_1.name, e_i.condition)$, $s_1 = Map(s(e_i))$
18     $R_{out} = R_{out} \cup \{r_{out}\}$ where $r_{out} = return(s_2.name)$, $s_2 = Map(t(e_i))$
19     $\delta(s_1, r_{in}) = s_2$
20     $\omega(s_1, r_{in}) = r_{out}$
21 // map the incoming flows of sc to the transitions started from its initial state
22 // if there is only one incoming flow
23 if $|E_{input}| = 1$ then
24     $R_{in} = R_{in} \cup \{r_{in}\}$ where $r_{in} = request(sc.name, e_i.condition)$, $e_i \in E_{input}$
25     $R_{out} = R_{out} \cup \{r_{out}\}$ where $r_{out} = return(s_2.name)$, $s_2 = Map(t(e_i))$
```

```
26          $\delta(s_0, r_{in}) = s_2$
27          $\omega(s_0, r_{in}) = r_{out}$
28    // if there are multiple incoming flows
29    else
30          $S = S \cup \{c_0\}$ where $c_0 = instantiatedFrom(Choice)$
31          $\delta(s_0, r_0) = c_0$
            foreach $e_i \in E_{input}$ do
                $R_{in} = R_{in} \cup \{r_i\}$ where $r_i$=specify($e_i$), $s_2 = Map(t(e_i))$
                $R_{out} = R_{out} \cup \{r_{out}\}$ where $r_{out} = return(s_2.name)$
                $\delta(c_0, r_i) = s_2$
                $\omega(c_0, r_i) = r_{out}$
      // map the outgoing flows of sc to the transitions pointing to its final state
      if $|E_{output}| = 1$ then
            …
```

Fig. 12 shows the input and output of this algorithm in SysML. Fig. 12(a) is an ACT of SysML which graphically represents the hybrid functional model. Actions allocated to a software coordinator are located in its *activity partition*. Cross-boundary control flows are highlighted. The output of the algorithm is the behavior model of a software coordinator which is modeled by the state machine (STM) of SysML. Fig. 12(b) shows the generated behavior of *TransferCoordinator*.



(a) Allocations between actions and software coordinators



(b) Behavior model of *TransferCoordinator*

Fig. 12  Inputs and results of behavior generation

### 6.3 Software behavior updating caused by supporting functions

It is necessary for the behaviors of software coordinators to be updated during iterative design processes. One of the causes of the behavior updating is the physical design. Some *supporting functions* may be inserted during physical design. Here, *supporting function* refers to that enables the prerequisites of other functions to be satisfied [65]. Accordingly, software coordinators may need to be updated to coordinate the execution sequences between supporting functions and functions they support.

Because of the correspondences between functions and behaviors of software coordinators, behavior models can be updated in the same way as functional models. Therefore, the key problem to update software behaviors is how to insert supporting functions into the functional models.

It is obvious that the ways to insert the supporting functions is determined by the execution sequences between supporting functions and their supported functions. Different types of execution sequences and corresponding methods to insert supporting functions are listed in Table1. Here, *a'* stands for the action that triggers the supporting function, $a_2$ represents the action that triggers the supported function (the original function whose prerequisite is the supporting function), and $a_1$ is the previous action of $a_2$ in the original model. The purpose is to insert *a'* between $a_1$ and $a_2$.

| Execution sequence | Example | Method to insert actions |
|---|---|---|
| **None** | Supported function: *pick (by sucking)*<br>Physical module: *bar with electromagnet*<br>Supporting function: *extend bar* | $\begin{tikzcd} a_1 \arrow[r] & a'_2 \end{tikzcd}$ |
| **Sequential** | Supported function: *pick (by clamping)*<br>Physical module: *jaw*<br>Supporting function: *extend jaw* | $\begin{tikzcd} a_1 \arrow[r] & a' \arrow[r] & a_2 \end{tikzcd}$ |
| **Parallel** | Supported function: *move pipe*<br>Physical module: *cart driven by gears*<br>Supporting function: *hold pipe* | $\begin{tikzcd} & & a' \\ a_1 \arrow[r] & {\mid} \arrow[ur] \arrow[dr] & \\ & & a_2 \end{tikzcd}$ |
| **Nondeterministic** | Supported function: *move forward*<br>Physical module: *wheeled cart on diff drive*<br>Supporting function: *generate power* | Manually insert |

Table1.  Classification of execution sequences between supporting and supported functions

*None* means there are no execution sequences to be coordinated by software. Instead, *a'* will be naturally executed after $a_2$ is finished driven by physical laws. Therefore, there is no need to insert *a'* into the HFM since the execution is not coordinated by software. Instead, *a'* and $a_2$ can be combined into a composite physical function *a'*$_2$.

*Sequential* means *a'* and $a_2$ need to be sequentially activated by software. Therefore, *a'* should be inserted right before $a_2$ in the functional model.

*Parallel* means *a'* and $a_2$ need to be activated by software in parallel. A fork node should be inserted and lead to *a'* and $a_2$ for their concurrency.

*Nondeterministic* means the execution sequence between *a'* and $a_2$ should be determined by designers manually. For example, the supporting function *GeneratePower* is required for achieving *MoveForward*. However, the timing to activate this function should be determined by designers. For this case, *a'* should be manually inserted by designers.

## 7. Implementation and case study

An ontology-based design framework is developed to support the proposed automated co-design approach. A CNC bending machine is used to illustrate how the proposed approach is practically used in a typical system design process.

### 7.1 Ontology-based system design framework

The ontology-base design framework includes three layers. The *frontend* provides a user interface for designers to create and view models graphically. The *transformation layer* works as an intermediary between the graphical models and ontologies, which transforms models to ontologies and vice versa. The *backend* performs reasoning on the ontological knowledge base to conduct certain tasks automatically in the design process.

A mainstream SysML modeling platform, i.e., MagicDraw is adopted currently as the frontend of this framework with developed plugins. The transformation layer is implemented base on ATL [70], which is an active model transformation framework. The mapping rules between the metamodel of SysML with extensions and the TBox of the knowledge base are defined to drive the transformation process. The backend contains modules to assist different tasks in the design process. Currently the implemented functions include functional decomposition [65,66], component retrieval and combination based on FEs [67], and component selection and software behavior maintenance proposed in this study. The backend is implemented in Java using Jena APIs. Protégé [71] is used as the ontology development platform, and Pellet [68] is adopted as the semantic web reasoner.

The core of the design framework is the ontological knowledge base in OWL2. Therefore, maintaining consistency of the knowledge is very important. To this end, efforts from two aspects are done in this framework.

First, axioms are defined in OWL2 and SWRL such that the semantic web reasoners can reason on the knowledge to maintain its consistency. The axioms are defined from three aspects: (1) formal definition of OWL classes based on class axioms; (2) description of the property characteristics in OWL2 such as the domain/range, inverse of, and etc.; (3) SWRL rules for specific needs such as evaluating the physical structure against execution sequences.

However, because of the open-world assumption (OWA) adopted by semantic web technologies (statements may be true as long as they are not known to be false), some of the checks cannot be done by semantic web reasoners. For example, it cannot determine whether there are any actions that are not allocated to certain components because if an action doesn't declare any allocatedTo property, the reasoner will assume it has such properties. Therefore, the code-level support is implemented as the second aspect of checks to address such kind of traceability check among the function, behavior and structure domains.

Regarding the software and physical domains, their consistency is maintained through the functional model. Any changes in these two designs must be made on the functions by designers manually. Then the unified functions are synthesized into the two domains through the F-B-S links.

### 7.2 Ontology-aided system design of CNC bending machine

The main function of the CNC bending machine is to bend a straight pipe according to the shape data provided by operators. A concept of and sample pipe bent by such machines are shown in Fig. 13. The shape data is an array of 3-tuples whose length is the number of expected curves on the pipe. Each tuple in the array specifies the expected location *y*, angle *b*, and bending angle *c* of each curve on the pipe. The radius of each curve is fixed. Three main physical processes are involved in the production process: (1) to move the pipe to the expected location *y*, (2) to rotate the pipe to the expected angle *b*, and (3) to bend the pipe to the expected angle *c*.

(a) Concept of the bending machine

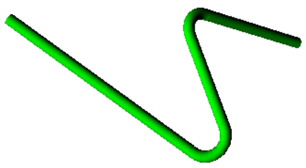
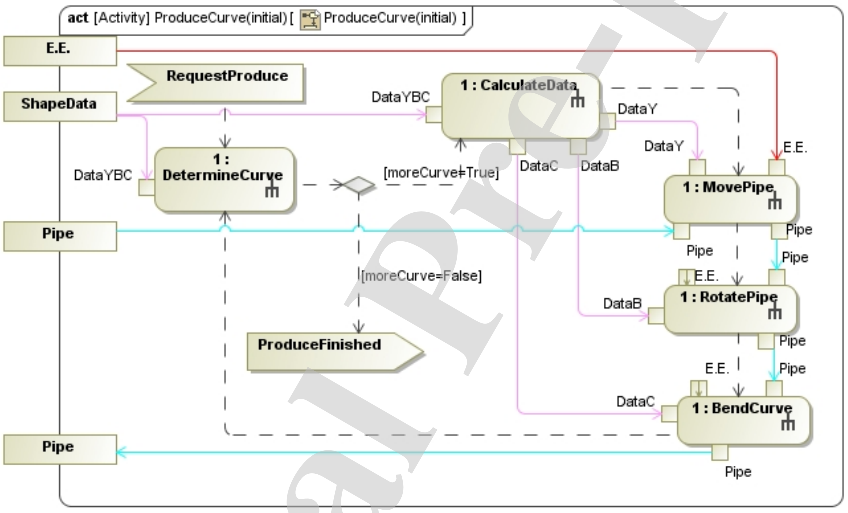

(b) Sample pipe bent by the machine

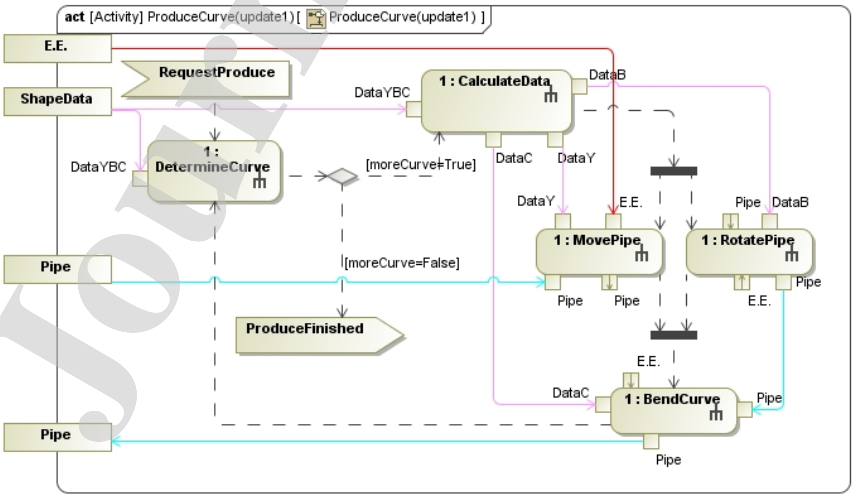
Fig. 13 CNC bending machine

***(1) Functional modeling***

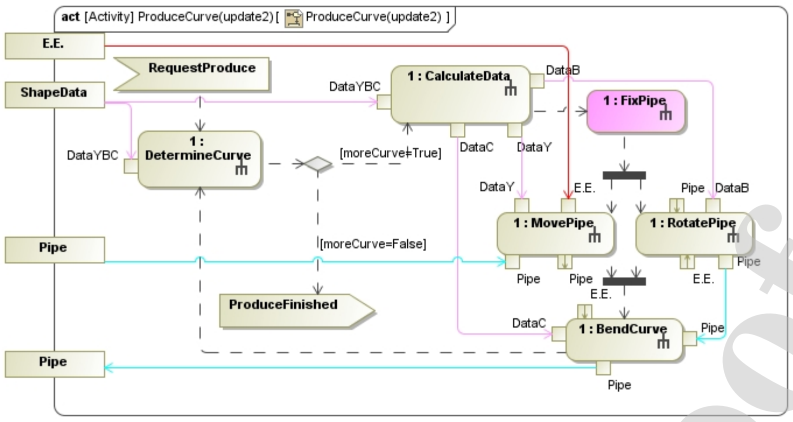
The first step of the system design process is functional modeling. After designers analyzed the requirement specifications presented above, the workflow of the main function *ProduceCurve* can be identified and modeled as shown in Fig. 14(a). This functional model is imported into the HFO to support the following ontology-based design process. After that, either the software design or physical design can be started. Different from the design of *Mobile Robot*, the software design is conducted first in this case study. Fig. 14(b) and (c) represent the updates of the functional model during the software and physical design respectively. Details are explained in the following sections.



(a) Initial



(b) Updated for concurrency

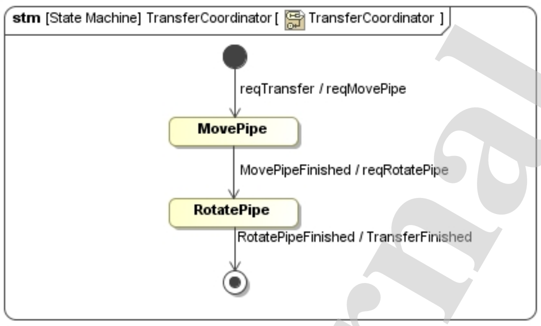


(c) Updated for supporting functions

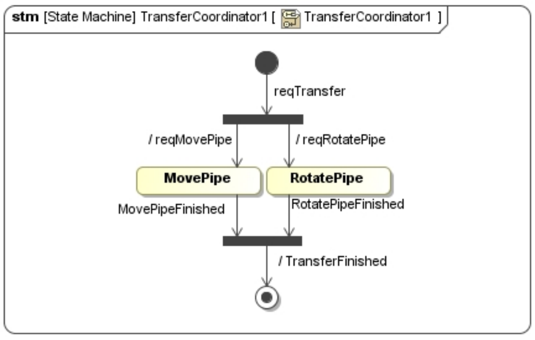
Fig. 14 Functional model of the CNC bending machine

***(2) Software design***

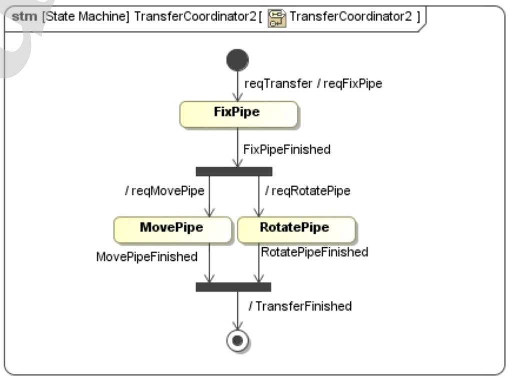
Based on the functional model, the designer identified three software coordinators, i.e., the *CentralCoordinator* coordinating the *DetermineCurve* and *CalculateData* functions, *TransferCoordinator* coordinating *MovePipe* and *RotatePipe*, and *BendCoordinator* coordinating *BendCurve*. This information is recorded as facts of the SCO. From the facts, behavior models of these software coordinators can be automatically generated by algorithm 1 proposed in Section 6.2. For example, the state machine of *TransferCoordinator* is shown in Fig. 15(a). Fig. 15(b) and (c) show the updates to the behavior model during the subsequent iterations of the software and physical design respectively. For example, during software design, the designer may determine that the moving and rotating should be executed in parallel to meet certain real-time requirement. The designer then updated the functional model and behavior models to reflect this change as shown in Fig. 14(b) and Fig. 15(b) respectively. The facts of the HFO and SCO are updated accordingly.


stm [State Machine] TransferCoordinator [ TransferCoordinator ]
reqTransfer / reqMovePipe
MovePipe
MovePipeFinished / reqRotatePipe
RotatePipe
RotatePipeFinished / TransferFinished

(a) Initial



(b) Updated for concurrency



(c) Updated for supporting functions

Fig. 15 Behavior model of *TransferCoordinator*

***(3) Physical design***

In the physical design, several candidate components are retrieved according to the FEs of physical functions. They are instantiated to the constituents of the system. The facts of PSO are populated accordingly.

Evaluation program is run against the facts automatically as introduced in Section 5.2. For example, the two functions *1:MovePipe* and *1:RotatePipe* are executed in parallel so that the components implementing them should be able to provide the two FEs at the same time. Mechanisms that do translation and rotation dependently or conflict with each other will be excluded. Based on the evaluation, *cart with rotating shaft driven by gears* is chosen to realize the two functions. This physical module requires supporting function *FixPipe* before the transfer process. The supporting function is inserted into the functional model as shown in Fig. 14(c).

***(4) Software design (iteration 2)***

Because the functional model is updated during physical design, the software design needs to be updated accordingly. For example, because of the two supporting functions, the behavior model of the software coordinator *TransferCoordinator* is updated automatically as shown in Fig. 15(c).

***(5) Architecture generation***

After all the software and physical components are determine, they should be grouped to generate the architecture of the system. The direct and fine-grained relationships can be established between physical and software modules according to their functional relationships. A software coordinator and the physical modules implementing the actions coordinated by it can compose a new type of modules, i.e., *intelligent mechatronic modules*. As shown in Fig. 16, they represent a new level of partition, which refactors the traditional MTS architecture [2]. The black-box information processing is decomposed into several software modules, each of which manipulates a certain group of BSs as resources for achieving their assigned functions.

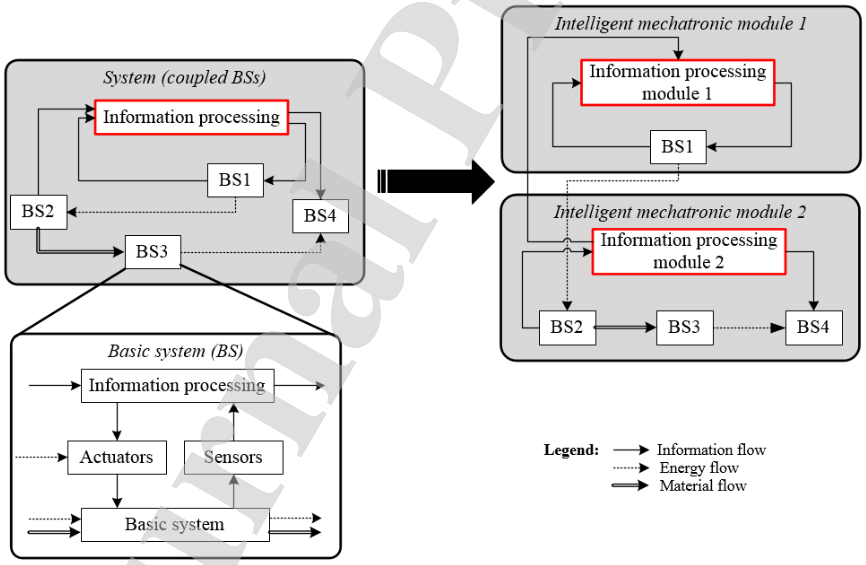


Fig. 16 Re-modularization of MTS system-level architecture

Based on this idea, the generated architecture of the CNC bending machine is shown in Fig. 17.

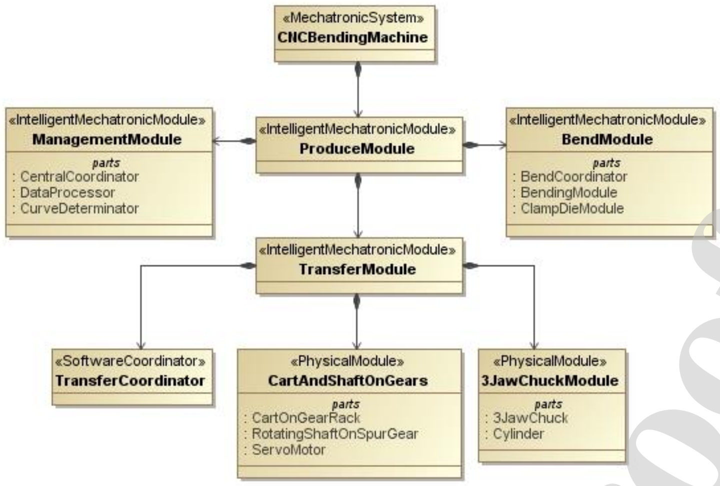

«MechatronicSystem» CNCBendingMachine
«IntelligentMechatronicModule» ManagementModule
parts
: CentralCoordinator
: DataProcessor
: CurveDeterminator
«IntelligentMechatronicModule» ProduceModule
«IntelligentMechatronicModule» BendModule
parts
: BendCoordinator
: BendingModule
: ClampDieModule
«IntelligentMechatronicModule» TransferModule
«SoftwareCoordinator» TransferCoordinator
«PhysicalModule» CartAndShaftOnGears
parts
: CartOnGearRack
: RotatingShaftOnSpurGear
: ServoMotor
«PhysicalModule» 3JawChuckModule
parts
: 3JawChuck
: Cylinder

Fig. 17 Architecture of the CNC bending machine

## 8. Discussion

It can be seen from the case study that interactions between software and physical design happen quite frequently when the design advances iteratively. The semantic web technologies-enabled approach proposed in this study can assist the co-design process effectively. The quality of the design can be improved from three aspects:

(1) For physical design, a new type of constraints, i.e., execution sequences of functions are considered in this study such that the unsatisfactory physical concepts can be filtered out in the early design.

(2) For software design, the behavior of a type of software components, i.e., software coordinators can be automatically generated. Moreover, it can be automatically updated according to the new functions introduced by physical design.

(3) The generated architecture is improved by a new modularization in which the software and physical components are linked through their functions. Therefore, the software and physical subsystems are no longer separated but integrated in fine-granularity.

The foundation of this approach is the correlated cross-domain knowledge in the ontological knowledge base. Functional knowledge represented by HFO is the common source to correlate the software and physical designs, through which design constraints and changes can be synchronized between them to keep consistency and integrity of the whole system design. Physical and software structures are connected to the unified functions through their behaviors such that physical structures can be evaluated against constraints related to execution sequences, and software behaviors can be automatically updated by supporting functions.

However, there are still some limitations of the proposed approach.

First, other types of functional properties can be included to constrain the physical design, e.g., the execution time of each function. It can be achieved by enhancing the ontologies proposed in this study. For example, the properties to describe the execution time of functions can be added, and correspondingly, the cool down time of the behavior of physical modules should also be recorded. Then, the two properties can be compared during the evaluation. However, the challenge is how to collect the values for above-mentioned properties in the early design. The qualitative values proposed by the qualitative simulation theory [72] may be leveraged to address it.

Second, the scope of the interactions should be extended. As mentioned in Section 3.3, there are three levels of control in the mechatronic systems. Currently, only the interactions between the middle-level control and the physical design are considered. Interactions among other levels of controls and between different levels of control and the physical structures should also be included.

Third, the consistency of the ontological knowledge base should be maintained in a more systematic and comprehensive way. Currently, only some cases are addressed by the formal definitions of concepts in OWL2 and code-level implementation. Possible inconsistencies should be collected and addressed especially when incremental updates are made on the knowledge.

## 9. Conclusions and future work

In this study, an automated approach to assist software-physical co-design is proposed, which is enabled by semantic web technologies. The knowledge of software and physical designs are correlated through the unified functional ontology such that design decisions can be synchronized across domains to avoid design defects in early design. The main novelty of this work is that it observed the impact between the software-physical design in the early conceptual design of mechatronic systems and proposed an ontology-based approach to support such interactions. Contributions of this approach are summarized as follows:

(1) The HFO is proposed, which can represent the functions of software and physical subsystems of MTSs in a unified knowledge schema.

(2) The physical design knowledge is enhanced by the dependencies of behaviors of physical structures and linked to functions in the PSO. Based on it, physical structures can be evaluated against the execution sequences of the functions so that unsatisfactory conceptual solutions can be identified in early design.

(3) Software design knowledge is linked to the functional knowledge in SCO, and algorithms are proposed so that the software behaviors can be automatically generated from and adapted to the functional knowledge to keep consistency of the software and physical design.

An ontology-based design framework is developed to automate the tasks involved in the approach. Case studies from different domains have demonstrated its generality and efficiency.

Efforts can be put on the following directions in the future. More influences between software and physical design need to be collected and investigated to enlighten the co-design process.

This methodology has the potential to be extended to the CPS design. However, CPSs are geographically distributed and functionally and structurally open systems, in which the components are able to make situational decisions and reorganize their structure and behavior dynamically. Therefore, the cyber parts are distributed to subsystems working autonomously instead of following determinate top-down functionality. The physical parts are geographically distributed and hence have fewer constraints when they are combined. In fact, CPSs can be treated as systems of systems, whereas this methodology currently is expected to be applied to individual standalone systems. To support CPS design, the systems should be modeled properly first. The high-level control and the networking aspects should be included in the system models. Besides that, the emergent behaviors should be considered when synthesizing functions into different constitutes of the system.

**Acknowledgement:**

The authors appreciate the support from the National Key Technology Support Program (2016YFD0400301), XXX(SHWXX20171ZL01), the National Science Foundation of China (61672247, 61772247 and 61873236), and the Open Fund of Science and Technology on Thermal Energy and Power Laboratory (No. TPL 2019A03).

## Reference

[1] Bricogne M, Le Duigou J, Eynard B. Design processes of mechatronic systems. Mechatron. Futur., 2016, p. 75–89. doi:10.1007/978-3-319-32156-1.

[2] VDI. Design methodology for mechatronic systems. VDI 2206. 2004.

[3] Wikander J, Törngren M, Hanson M. The science and education of mechatronic engineering. Robot Autom Mag 2001;8:20–6. doi:10.1109/100.932753.

[4] Alvarez Cabrera AA, Foeken MJ, Tekin OA, Woestenenk K, Erden MS, De Schutter B, et al. Towards automation of control software: A review of challenges in mechatronic design. Mechatronics 2010;20:876–86. doi:10.1016/j.mechatronics.2010.05.003.

[5] Alvarez Cabrera AA, Woestenenk K, Tomiyama T. An architecture model to support cooperative design for mechatronic products: A control design case. Mechatronics 2011;21:534–47. doi:10.1016/j.mechatronics.2011.01.009.

[6] Saathof R, Thier M, Hainisch R, Schitter G. Integrated system and control design of a one DoF nano-metrology platform. Mechatronics 2017;47:88–96. doi:10.1016/j.mechatronics.2017.08.013.

[7] Zheng C, Bricogne M, Le Duigou J, Eynard B. Survey on mechatronic engineering: A focus on design methods and product models. Adv Eng Informatics 2014;28:241–57. doi:10.1016/j.aei.2014.05.003.

[8] Thramboulidis K. Model-integrated mechatronics—toward a new paradigm in the development of manufacturing systems. IEEE Trans Ind Informatics 2005;1:54–61. doi:10.1109/TII.2005.844427.

[9] Schäfer W, Wehrheim H, Sch W. The challenges of building advanced mechatronic systems. Futur. Softw. Eng. FOSE 07,

2007, p. 72–84. doi:10.1109/FOSE.2007.28.

[10] Yang QZ, Zhang Y. Semantic interoperability in building design: Methods and tools. Comput Aided Des 2006;38:1099–112. doi:10.1016/j.cad.2006.06.003.

[11] Chungoora N, Young RI, Gunendran G, Palmer C, Usman Z, Anjum NA, et al. A model-driven ontology approach for manufacturing system interoperability and knowledge sharing. Comput Ind 2013;64:392–401. doi:10.1016/j.compind.2013.01.003.

[12] Feldmann S, Herzig SJI, Kernschmidt K, Wolfenstetter T, Kammerl D, Qamar A, et al. Towards effective management of inconsistencies in model-based engineering of automated production systems. IFAC-PapersOnLine 2015;48:916–23. doi:10.1016/j.ifacol.2015.06.200.

[13] Lu W, Qin Y, Liu X, Huang M, Zhou L, Jiang X. Enriching the semantics of variational geometric constraint data with ontology. Comput Des 2015;63:72–85. doi:10.1016/j.cad.2014.12.008.

[14] ISO/IEC/IEEE. ISO/IEC/IEEE 42010: Systems and software engineering — Architecture description. 2011. doi:10.1109/IEEESTD.2012.6170923.

[15] Pahl G, Beitz W. Engineering design—a systematic approach. 3rd ed. Springer-Verlag London Limited; 2007.

[16] Hirtz J, Stone RB, McAdams DA, Szykman S, Wood KL. A functional basis for engineering design: Reconciling and evolving previous efforts. Res Eng Des 2002;13:65–82. doi:10.1007/s00163-001-0008-3.

[17] Kruse B, Gilz T, Shea K, Eigner M. Systematic comparison of functional models in SysML for design library evaluation. Procedia CIRP 2014;21:34–9. doi:10.1016/j.procir.2014.03.175.

[18] Wan J, Canedo A, Al Faruque MA. Cyber–physical codesign at the functional level for multidomain automotive systems. IEEE Syst J 2017;11:2949–59. doi:10.1109/JSYST.2015.2472495.

[19] Chen Y, Liu ZL, Xie YB. A knowledge-based framework for creative conceptual design of multi-disciplinary systems. Comput Aided Des 2012;44:146–53. doi:10.1016/j.cad.2011.02.016.

[20] Helms B, Shea K. Computational synthesis of product architectures based on object-oriented graph grammars. J Mech Des 2012;134:021008. doi:10.1115/1.4005592.

[21] Umeda Y, Ishii M, Yoshioka M, Shimomura Y, Tomiyama T. Supporting conceptual design based on the function-behavior-state modeler. Artif Intell Eng Des Anal Manuf 1996;10:275–88. doi:10.1017/S0890060400001621.

[22] Dromey RG. From requirements to design: Formalizing the key steps. Proc. - 1st Int. Conf. Softw. Eng. Form. Methods, SEFM 2003, 2003, p. 2–11. doi:10.1109/SEFM.2003.1236202.

[23] Zheng C, Le Duigou J, Bricogne M, Eynard B. Multidisciplinary interface model for design of mechatronic systems. Comput Ind 2016;76:24–37. doi:10.1016/j.compind.2015.12.002.

[24] Zheng C, Le Duigou J, Bricogne M, Dupont E, Eynard B. Interface model enabling decomposition method for architecture definition of mechatronic systems. Mechatronics 2016;40:194–207. doi:10.1016/j.mechatronics.2016.10.008.

[25] Thramboulidis K. The 3+1 SysML view-model in model integrated mechatronics. J Softw Eng Appl 2010;03:109–18. doi:10.4236/jsea.2010.32014.

[26] Kernschmidt K, Feldmann S, Vogel-heuser B. A model-based framework for increasing the interdisciplinary design of mechatronic production systems. J Eng Des 2018;29:617–43. doi:10.1080/09544828.2018.1520205.

[27] Fan H, Liu Y, Hu B, Ye X. Multidomain model integration for online collaborative system design and detailed design of complex mechatronic systems. IEEE Trans Autom Sci Eng 2016;13:709–28. doi:10.1109/TASE.2015.2390039.

[28] Cao Y, Liu Y, Fan H, Fan B. SysML-based uniform behavior modeling and automated mapping of design and simulation model for complex mechatronics. Comput Aided Des 2013;45:764–76. doi:10.1016/j.cad.2012.05.001.

[29] Tian F, Voskuijl M. Automated generation of multiphysics simulation models to support multidisciplinary design optimization. Adv Eng Informatics 2015;29:1110–25. doi:10.1016/j.aei.2015.07.004.

[30] Malmquist D, Frede D, Wikander J. Holistic design methodology for mechatronic systems. Proc Inst Mech Eng Part I J Syst Control Eng 2014;228:741–57. doi:10.1177/0959651814527936.

[31] Behbahani S, de Silva CW. Mechatronic design quotient as the basis of a new multicriteria mechatronic design

methodology. IEEE/ASME Trans Mechatronics 2007;12:227–32. doi:10.1109/TMECH.2007.892822.

[32] Mohebbi A, Achiche S, Baron L. Multi-criteria fuzzy decision support for conceptual evaluation in design of mechatronic systems: a quadrotor design case study. Res Eng Des 2018;29:329–49. doi:10.1007/s00163-018-0287-6.

[33] Li Q, Zhang WJ, Chen L. Design for control-a concurrent engineering approach for mechatronic systems design. IEEE/ASME Trans Mechatronics 2001;6:161–9. doi:10.1109/3516.928731.

[34] Mohebbi A, Achiche S, Baron L. Design of a vision guided mechatronic quadrotor system using design for control methodology. Trans Can Soc Mech Eng 2016;40:201–19.

[35] Mohebbi A, Gallacher C, Harrison J, Willes J, Achiche S. Integrated structure-control design optimization of an unmanned quadrotor helicopter (UGH) for object grasping and manipulation. Proc. Int. Conf. Eng. Des. ICED, vol. 4, 2017, p. 623–32.

[36] Behbahani S, de Silva CW. System-based and concurrent design of a smart mechatronic system using the concept of mechatronic design quotient (MDQ). IEEE/ASME Trans Mechatronics 2008;13:14–21. doi:10.1109/TMECH.2007.915058.

[37] Mohebbi A, Achiche S, Baron L. Integrated and concurrent detailed design of a mechatronic quadrotor system using a fuzzy-based particle swarm optimization. Eng Appl Artif Intell 2019;82:192–206. doi:10.1016/j.engappai.2019.03.025.

[38] Van Brussel H, Sas P, Németh I, De Fonseca P, Van Den Braembussche P. Towards a mechatronic compiler. IEEE/ASME Trans Mechatronics 2001;6:90–105. doi:10.1109/3516.914395.

[39] Sakao T, Umeda Y, Tomiyama T, Shimomura Y. Model-based automatic generation of sequence-control programs from design information. IEEE Expert Syst Their Appl 1997;12:54–61. doi:10.1109/64.590076.

[40] Tranoris C, Thramboulidis K. A tool supported engineering process for developing control applications. Comput Ind 2006;57:462–72. doi:10.1016/j.compind.2006.02.006.

[41] Vogel-Heuser B, Schütz D, Frank T, Legat C. Model-driven engineering of manufacturing automation software projects - a SysML-based approach. Mechatronics 2014;24:883–97. doi:10.1016/j.mechatronics.2014.05.003.

[42] Bassi L, Secchi C, Bonfé M, Fantuzzi C. A SysML-based methodology for manufacturing machinery modeling and design. IEEE/ASME Trans Mechatronics 2011;16:1049–62. doi:10.1109/TMECH.2010.2073480.

[43] Barbieri G, Fantuzzi C, Borsari R. A model-based design methodology for the development of mechatronic systems. Mechatronics 2014;24:833–43. doi:10.1016/j.mechatronics.2013.12.004.

[44] Thramboulidis K. Comments on “A model-based design methodology for the development of mechatronic systems.” Mechatronics 2015;28:1–3. doi:10.1016/j.mechatronics.2015.05.001.

[45] Zheng C, Hehenberger P, Le Duigou J, Bricogne M, Eynard B. Multidisciplinary design methodology for mechatronic systems based on interface model. Res Eng Des 2017;28:333–56. doi:10.1007/s00163-016-0243-2.

[46] Thramboulidis K. A cyber-physical system-based approach for industrial automation systems. Comput Ind 2015;72:92–102. doi:10.1016/j.compind.2015.04.006.

[47] Vanherpen K, Denil J, David I, De Meulenaere P, Mosterman PJ, Torngren M, et al. Ontological reasoning for consistency in the design of cyber-physical systems. 2016 1st Int. Work. Cyber-Physical Prod. Syst. CPPS 2016, 2016. doi:10.1109/CPPS.2016.7483922.

[48] Penas O, Plateaux R, Patalano S, Hammadi M. Multi-scale approach from mechatronic to Cyber-Physical Systems for the design of manufacturing systems. Comput Ind 2017;86:52–69. doi:10.1016/j.compind.2016.12.001.

[49] Fitz T, Theiler M, Smarsly K. A metamodel for cyber-physical systems. Adv Eng Informatics 2019;41:100930. doi:10.1016/j.aei.2019.100930.

[50] Bräunl T. Embedded robotics - Mobile robot design and applications with embedded systems. Springer-Verlag Berlin Heidelberg; 2008. doi:10.1007/3-540-34319-9.

[51] Akkaya I, Derler P, Emoto S, Lee EA. Systems engineering for industrial cyber – physical systems using aspects. Proc IEEE 2016;104:997–1012. doi:10.1109/JPROC.2015.2512265.

[52] De Giacomo G, Lenzerini M. TBox and ABox Reasoning in Expressive Description Logics. KR, 1996, p. 316–27.

[53] Kavi KM, Buckles BP, Bhat UN. A formal definition of data flow graph models. IEEE Trans Comput 1986;C-35:940–8. doi:10.1109/TC.1986.1676696.

[54] Young RIM, Gunendran AG, Cutting-Decelle AF, Gruninger M. Manufacturing knowledge sharing in PLM: A progression towards the use of heavy weight ontologies. Int J Prod Res 2007;45:1505–19. doi:10.1080/00207540600942268.

[55] Ye Y, Yang D, Jiang Z, Tong L. Ontology-based semantic models for supply chain management. Int J Adv Manuf Technol 2008;37:1250–60. doi:10.1007/s00170-007-1052-6.

[56] Chungoora N, Young RIM. The configuration of design and manufacture knowledge models from a heavyweight ontological foundation. Int J Prod Res 2011;49:4701–25. doi:10.1080/00207543.2010.504754.

[57] W3C. OWL2 web ontology language structural specifcation and functional-style syntax (second edition) 2012. doi:10.1086/305384.

[58] Groover MP. Levels of automation. Autom. Prod. Syst. Comput. Manuf. 4th ed., Pearson Education India; 2016.

[59] Hoffmann H. Harmony SE: A SysML based systems engineering process. Innov. 2008 Telelogic User Gr. Conf., 2008, p. 1–25.

[60] Lykins H, Friedenthal S, Meilich A. Adapting UML for an object oriented systems engineering method (OOSEM). Proc. 10th Int. INCOSE Symp., 2000.

[61] Alur R, Courcoubetis C, Halbwachs N, Henzinger T a., Ho P-H, Nicollin X, et al. The algorithmic analysis of hybrid systems. Theor Comput Sci 1995;138:3–34. doi:10.1016/0304-3975(94)00202-T.

[62] Lee EA. CPS foundations. Proc. 47th Des. Autom. Conf., 2010, p. 737–42.

[63] Gulwani S, Polozov O, Singh R. Program synthesis. Found Trends® Program Lang 2017;4:1–119. doi:10.1561/2500000010.

[64] OMG. Systems modeling language specification, version 1.5 2017. https://www.omg.org/spec/SysML/1.5/.

[65] Yuan L, Liu Y, Sun Z, Cao Y, Qamar A. A hybrid approach for the automation of functional decomposition in conceptual design. J Eng Des 2016;27:333–60. doi:10.1080/09544828.2016.1146237.

[66] Yuan L, Liu Y, Lin Y, Zhao J. An automated functional decomposition method based on morphological changes of material flows. J Eng Des 2017;28:47–75. doi:10.1080/09544828.2016.1258459.

[67] Chen R, Liu Y, Liu Y, Zhang Z, Ye X, Hu J. An automated generation method of system architecture with component's multi-criterion. Proc. 21st Int. Conf. Eng. Des. (ICED 17), vol. 4, Vancouver, Canada: 2017, p. 357–66.

[68] Sirin E, Parsia B, Grau BC, Kalyanpur A, Katz Y. Pellet: A practical OWL-DL reasoner. Web Semant 2007;5:51–3. doi:10.1016/j.websem.2007.03.004.

[69] Vyatkin V. IEC 61499 as enabler of distributed and intelligent automation: State of the art review. IEEE Trans Ind Informatics 2011;7:768–81. doi:10.1109/TII.2011.2166785.

[70] Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. Sci Comput Program 2008;72:31–9. doi:10.1016/j.scico.2007.08.002.

[71] Musen MA. The protégé project:A look back and a look forward. AI Matters 2015;1:4–12. doi:10.1145/2757001.2757003.

[72] Kuipers B. Qualitative Simulation. Artif Intell 1984;29:289–338. doi:10.1016/B978-1-4832-1447-4.50018-3.

# An automated approach for software and physical co-design of mechatronic systems based on hybrid functional ontology

## Highlights

- A hybrid functional ontology is proposed, which unifies the physical-centric flow-based functional representation and software-centric data/control flow diagram .
- Software and physical designs are linked to the unified functional knowledge by ontologies defined in OWL2 DL.
- Physical structures are automatically evaluated against the execution sequences of the functions based on SWRL .
- Software behaviors are automatically generated from and adapted to the functions of physical subsystems.
- The approach is illustrated on two case studies from different application areas.

No potential conflict of interest was reported by the authors.