Towards Dependable Model-Driven Design of Low-Level Industrial Automation Control Systems

Nan Zhou, Di Li, Valeriy Vyatkin, Victor Dubinin and Chengliang Liu

***Abstract*—Recent technological advances and manufacturing paradigm evolutions in industrial settings will dramatically increase the complexity of automation control systems. Traditional solutions to the software development of low-level control kernels (e.g., numerical control kernel, motion control kernel, real-time communication tasks) are unable to properly cope with such complexity due to inadequate level of abstraction and challenges for dependability. This paper presents a formal semantics integrated model-driven design approach as a holistic solution. A domain-specific modeling language (DSML) is specified based on the adaption of IEC 61499 architecture, along with the extensions of task modeling, task-to-resource allocating and non-functional specification. Both formal structural and behavioral semantics of the proposed DSML are then explicitly defined. A meta-programmable environment is developed for flexible modeling, verification and code generation. Design-time formal verification is achieved by automated model transformation. A case study is demonstrated on implementing a prototype computer numerical control (CNC) system using the proposed solution.**

***Note to Practitioner*—The low-level automation control system in the modern manufacturing scenarios require more agility while respecting strict timing constraints. Handling such a complexity with manual coding is getting harder and lower efficient. The domain-specific modeling language and the supporting development environment presented in this paper aims to enhance the level of automation, flexibility and dependability of the whole design process. For the proposed DSML, the syntax is defined as meta-models while the semantics is integrated through model transformation and model annotation. These definitions are implemented as external rules for a meta-programmable environment to establish our proposed development tool. The finding and insight from this paper can be used to enhance the efficiency and dependability during the developments process of common control kernels, such as CNC kernel, motion controller software, etc.**

***Index Terms*—Industrial Automation Control System; Model-Driven Engineering; Domain-Specific Modeling Language; IEC 61499; Formal Verification.**

I. Introduction

Modern industrial automation and control systems are dominated by distributed mechatronic devices. These devices perform functions according to the decisions from their built-in controllers, ranging from programmable logic controllers (PLC), robot controllers, numerical controllers, general motion controllers, etc. Typically, these controllers should be embedded with real-time systems for executing time-critical automation control software, including user-specific applications and a control kernel. User-specific automation applications are concerned by controller users while the low-level control kernel is provided by controller vendors.

This paper is mainly focusing on the development aspect of control kernels, which typically feature some common functions across various scenarios of industrial automation control systems, such as motion control, drive communication, etc. Traditionally, the required functions of control kernel are coded in a manual way. Then, the control kernel is supplied along with the controllers by vendors in a closed way, providing limited extensibility and proprietary application programming interfaces (API). Therefore, system integrators may be stuck to vendor-specific solutions when developing automation applications, leading to restricted possibilities of rapid product evolutions. Since the modern manufacturing paradigm is shifting from mass production to mass customization, industrial automation and control systems will be facing ever changing demands regarding their functions and structures. Being adaptive to these demands while respecting rigorous non-functional constraints (e.g. hard real-time capability, safety, stability) in shorter time-to-market cycles requires higher agility of the control kernels. These challenges raise the complexities of the software development process. Taming such complexities with the code-centric approaches is a time-consuming and error-prone process due to inadequate level of abstraction and dependability. Thus, adoption of new design methodologies is imperative.

Under the mass customization manufacturing paradigm, the control kernels should adopt a component-based and reconfigurable pattern in the distributed context, as pointed out in [1]. Related works in this domain usually place the emphasis on the architectural design and the specification of component models, as can be seen in [2], [3] and [4]. The development methodology, however, still remains quite primary from the perspective of abstraction and automation level. For example, when developing control software in the context of OROCOS[1] (an open-source low level framework for machine and robot control) or LinuxCNC[2] (an open source CNC machine controller), the only alternative is manual coding with specific low level programming languages. Validation and tuning of non-functional properties must be performed repeatedly on the

Nan Zhou and Di Li is with the School of Mechanical and Automotive Engineering, South China University of Technology, Guangzhou, 510640 China (e-mail: menanchow@mail.scut.edu.cn, itdili@scut.edu.cn)

Valeriy Vyatkin is with Luleå University of Technology, Sweden, 97187, ITMO University, Russia, and Aalto University, Finland, 02150 (e-mail: vyatkin@ieee.org)

Victor Dubinin is with the Department of Computer Science, University of Penza, Penza, Russia (e-mail: victor_n_dubinin@yahoo.com)

Chengliang Liu is with the Institute of Mechatronics, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: chlliu@sjtu.edu.cn)

[1] www.orocos.org

[2] www.linuxcnc.org

targeted platforms. To overcome such drawbacks, advanced developing language and technology should be adopted, with the following properties supported:

- *Model based*. This is the basic requirement as it can enhance the abstraction level of the development process.
- *Vendor independent*. The requirement can ensure that our proposal is not bounded to specific proprietary technologies and platforms.
- *Formal semantics integrated*. This requirement is aiming at facilitating early verification to improve the dependability of the modeled artifacts.

Model-driven engineering (MDE) is regarded as one of the most promising solutions to modern complex software development since it can ensure high abstraction level, modularity as well as flexibility. The Unified Modeling Language (UML) defined by the Object Management Group (OMG) is widely applied in the MDE research. Nevertheless, as a kind of general modeling language, UML lacks the supports of depicting and programming in the control engineering's way. Contrarily, domain-specific languages (DSL), or more specifically modeling languages (DSML), aim at specific application fields. DSML can provide intuitive modeling facilities in the direct notations of domain knowledge. Thus, it's developer-friendly. Given that developers in the industrial control community usually have inadequate expertise on software engineering, this paper adopts the DSML technology.

Specification of a DSML starts from conceptualizing common knowledge of the concerned domain and the results turn into the syntax definitions of the DSML, which are also termed as meta-model definitions. The abstraction level and scope of the conceptualization process will determine how applicable the resultant DSML can be in the concerned domain. In this paper, the architecture of IEC 61499 [5] and its elements are utilized as the means for domain conceptualization. Although IEC 61499 is primarily the standard for modeling application logics intended to be executed on networks of PLCs, this paper takes a step further by adopting it for lower-level control framework design in broader fields, such as general motion control, numerical control, etc. Stringent timing and resource constraints should be meet in these fields, which are out of the scope of IEC 61499. Therefore, the concepts of execution tasks and rule-based non-functional constraints are defined in the syntax domain as the extensions of IEC 61499.

Inconsistency among multiple, interrelated models in terms of structures and behaviors may deteriorate dependability of the design process and the final executables. Hence, some correct-by-construction methodologies should be employed to facilitate automated validation. Formal semantics definitions of DSML are the initial step in this direction. Particularly, two types of semantics are involved: structural and behavioral semantics. Structural semantics aims to describe the meaning of the models in terms of the structure of model instances while behavioral semantics is about dynamic characteristics when executed. In this paper, both these semantics are explicitly specified.

The remaining part of this article is organized as follows. Section II briefly introduces related works. The overall methodology adopted in this paper is illustrated in Section III. Following that, the syntactical and semantic definitions are given out in Section IV and Section V, respectively. Section VI discusses a model transformation approach for integrating formal verification of domain models. A proof-of-concept case study on implementing a CNC system is demonstrated in Section VII. Finally, Section VIII concludes the paper and outlines the future work.

II. Related Works

A. Modeling of Industrial Automation Control Systems

This paper proposes adapting existing standardized high level DSMLs for the fields of lower-level automation control, because such a strategy can reuse knowledge and well-proven practices to ensure acceptances by developers. For the high-level user-specific applications which are executed on top of control kernels, there have been amounts of standardized DSLs available, and most of them are model-based.

In the world of PLC-based automation control programming, there are two predominant international standards: IEC 61131-3 [6] and IEC 61499. In this paper, IEC 61499 is regarded as a more appropriate choice for adaptation, since it specifies a systematic approach for modeling distributed control system based on hierarchical event-driven function block (FB) architecture. Besides, IEC 61499 provides inherent supports of reconfiguration of FB networks. These features are desirable for modeling component-based reconfigurable control systems. A comprehensive review on the recent applications of IEC 61499 in the industrial automation domain can be referred to [7]. In [8] IEC 61499 is utilized to model the control software of machine tools. With regard to low-level control, functions of PLC/CNC control and real-time communications [9] are encapsulated in FBs. In [10] a novel layered CNC architecture is proposed based on IEC 61499. Since their implementations relied on Java-based technologies, a dedicated processor is assumed to be required for ensuring real-time capabilities. These practices prove the feasibility of IEC 61499 in the low-level control software design.

For the vision of dependable domain-specific modeling to become reality, a design environment is necessary. Such a development environment with efficient correct-by-design mechanisms integrated is critical for the availability and success of a DSML. There are several active IEC 61499 modeling tools available for developers, as introduced in [11]. However, most of these tools lack flexible support for design-time verifications and corrections of semantics ambiguities and errors. Furthermore, most of these tools require specific run-time environments for executing models. As per this pattern, behavior semantics of models only implicitly emerge from these manually-coded software, leading to the problem that validation is only possible in the late stage.

Besides, most of the available IEC 61499 modeling tools are built on such an infrastructure where syntax representations and semantics analysis are hard-coded. Any updates of syntactical and semantic rules will result in heavy programming and re-compiling of these tools. Such an inflexibility will lead to problems on model managements and exchanges, which may further affect the dependability of models. These problems

are also identified by Dai et al. [12]. An extensible approach based on ontology is then proposed by them. However, their work did not propose or exploit behavioral semantics of IEC 61499 in a formal way. Instead, it still relies on the underlying execution environment. In this paper, a meta-programmable and extensible modeling environment is developed to eliminate such pitfalls existing in the current tools. The generic modeling environment (GME) [13] is introduced as the basic framework of this tool, and both syntactical and semantic definitions are encoded as external rules in the format of extensible markup language (XML) files.

B. *Verification of Industrial Automation Control Systems*

Formal methods are a well-proven solution to early validation and verification during the IEC 61499 based control system design process. Usually a specific type of formal model or language should be selected and then transformation rules between source models and the formal counterparts are established, such as the NCES (a Petri Nets variant) model in [14], abstract state machine in [15], Esterel (a synchronous language) in [16]. These works mainly contribute to verification against correctness of logical sequences. In [17] timed automata are employed for verifying temporal behavior of FBs and FB network from the nonfunctional perspective. In their work, a scheduler is modeled for verifying the temporal behaviors of FB networks. However, in real-time control the basic scheduled entities of operating systems should be modeled and verified as well, which was not addressed in the work presented in [17].

Mapping IEC 61499 FB networks to scheduled tasks is necessary for specifying real-time constraints. Doukas and Thramboulidis propose in [18] the concept of Event Paths (`EP`) as segments of application to capture real-time requirements using the notation of deadlines and priorities. A similar concept called Event Chain (`EC`) is introduced by Zoitl et al. in [19]. However, the timing constraints are only applied in the implementation phase. Model-level verification of schedulability is therefore impossible. In [20] Lindgren et al. use `EC` for introducing an informal real-time semantics to enable model-level analysis of schedulability-related properties. The timing analysis relies on the prerequisite implement of stack resource policy scheduler.

In this paper, we introduce a concept of execution task attributed with rule-based timing constraints as the design-time extension in the context of IEC 61499. With these extensions available, modular and hierarchical mapping between the proposed DSML and formal models can be set up. In our current work, we use time automata for verifying functional and non-functional properties. Since the behavior semantics of time automaton is defined by labelled transition system (LTS) mathematically, the paper currently leverages LTS as the basis for the formal behavior descriptions of FB, FB networks as well as execution tasks to maintain semantics equivalence during model transformation.

To guarantee the rationality of our formal behavior semantics definitions, some readily available execution models of FB and FB networks proposed in the existing works will be directly adapted and represented in LTS. Currently we prefer the buffered sequential execution model [21] due to the fact it is used in the most industrially adopted tools, such as NxtControl and 4DIAC. A complete formal behavior semantics specification of our DSML is attained by composing behavioral semantics of execution task and these existing formal execution semantics. In addition, the object constraint language (OCL) is introduced for formal specifications of structural semantics. In this way, both structural and behavioral semantics can be integrated in the proposed DSML-based approach on a formal basis, thus ensuring dependability of model artifacts at the early design stage.

To the best of the authors' knowledge, no related work exists for comprehensively integrating corrections and verifications of structural and behavioral semantics during the modeling process of low level control system with IEC 61499. In [22] and [15] both structural and behavioral semantics are considered, but the work takes synchronous semantics as a prerequisite. This semantic model requires dedicated compiler and has minor applications currently. This paper presents an initial step toward this end on the basis of the sequential execution model, which is widely adopted among existing IEC 61499 applications. Two principles are followed in our work. The first one is to reuse as much as possible existing well-proven practices on formal syntax and semantics about IEC 61499, which has been stated above. The other one is to leverage techniques and tools with industrial maturity to set up supporting tool-chains for dependable domain modeling. For the second principle, the model-integrated computing (MIC) approach [23] and the supporting tool-suites [13] are involved. MIC represents a model-based design paradigm for embedded real-time system using DSML. The tool UPPAAL for formal verification is integrated via model-to-model transformation. By following these two principles, the paper achieves the contributions as follows:

- A domain-specific modeling language with complete abstract syntax based on IEC 61499 is proposed for efficient development of low-level industrial control systems;
- Both formal behavioral and structural semantics are defined for the proposed DSML for dependable design of domain models, and a model transformation method is proposed to automate formal verification of domain models;
- A meta-programmable environment for modeling, verification and code generation is established, with which a prototype CNC system is implemented to prove the feasibility of IEC 61499 in low level control system design.

III. Overview of Methodology

This section introduces the overview of our proposed solution to dependable model-based design of low level industrial automation control systems. Basic concepts and features of MIC-based meta-modeling are also briefly introduced.

A. *Overview*

Our proposed solution is based on the MIC approach and its related tools. The framework for implementing model-

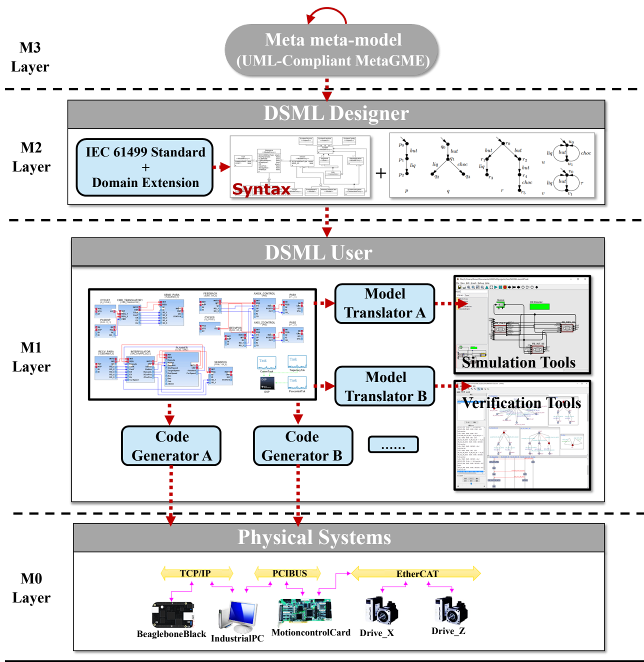


Fig. 1. The architecture of DSML in the OMG four-layered architecture

based design in the context of MIC can be depicted with the classical four-layered architecture proposed by OMG, as shown in Fig. 1. In the *M0* layer, industrial automation systems can be implemented with domain models provided in the *M1* layer. Domain models should conform to specific syntax and semantics defined in the *M2* layer. Prior to specifying these definitions, analysis of domain concepts should be conducted to identify the scope of the DSML. In our solution, the reference architecture and elements of the IEC 61499 standard, along with domain-specific extensions, are adopted as the guidelines for meta-modeling. These meta-models are conformed to a self-explanatory meta-metamodel called MetaGME, which has the basic elements compliant to the UML class diagrams. The meta-metamodel provided by the MIC tools is defined in the *M3* layer.

In detail, there are two roles of developers in our proposal. The first one is DSML developer mainly involved in the *M2* layer, and the other one is (low level) control system developer in the *M1* layer. In the *M2* layer syntactical elements are described using UML-compliant class diagrams while structural semantics are defined by OCL rules. OCL is a declarative language for describing rules that apply to models in UML notation, which makes it suitable for integration in our UML-compliant meta-models. Behavior semantics are then formulated accordingly. Since there are no tools for verifying the behavior semantics of our domain models directly, a model translator is required to set up the mappings between them and other formal models. Similarly, integrations of model simulators or code generators can be achieved by developing the corresponding model translators. These translators are developed based on the syntactical elements in the *M2* layer.

In the *M1* layer, developers may design their own systems or applications using the domain models defined in the upper layer. Runtime validation of OCL rules is enabled in this layer to detect inconsistencies of domain models instantly. The achieved model artifacts will be taken as inputs for model transformations or code generations.

TABLE I
TYPICAL KINDS OF METAGME NOTATIONS

MetaGME concepts	Roles	Visualization
Model	Compositional containers	Rectangle
Atom	Primitive objects	Rectangle
Reference	Pointers to MetaGME objects	Rectangle
Set	Aggregate containers	Rectangle
Connection	Non-Aggregate relation	Rectangle
Containment	Aggregate association	Solid Line (*Ended with Diamond*)
Aspect	Logical visual partitions	Rectangle
FieldAttribute	Properties of objects	String-typed Attribute
Inheritance	Inheritance association	Triangle
First Class Objects	Abstract of entities	Rectangle
Proxy	Alias of specific object	Rectangle

The tool architecture for applying the MIC approach is detailed in [13]. Specifically, the whole architecture is based on the Generic Modeling Environment (GME), which is a meta-programmable and configurable tool for definition and composition of domain-specific models. Due to these features, GME can be used interchangeably for meta- and domain-specific modeling by simply replacing the external definition rules file. The XML-based file is termed as `paradigm` in the context of MIC. In our case, the resultant paradigm generated in the *M2* layer contains all the necessary syntax and structural semantics definitions. Incompatibilities of models after modifications or upgrades of the paradigm can be automatically detected by GME to eliminate any possible ambiguities. The developed model translators can be integrated in the GME as plug-ins and they are invoked manually in the *M1* layer.

B. Metamodeling with GME

This subsection briefly introduces the MetaGME notations and some related featurs of GME as background knowledge for the following syntax definitions. Comprehensive introductions on the details of MetaGME can be found in [24].

MetaGME is an extension of UML class diagrams with OCL constraints to imply the abstract syntax specified by the metamodel. Therefore, MetaGME shares many similarities with Meta Object Facilities (MOF), the meta-modeling language of UML. Compared with MOF, MetaGME enables modeling of separated concerns by introducing ”Aspect” [25] for viewpoint notation. However, this concept will not be elaborated in our syntax definitions due to page limits. Basic concepts of MetaGME notations as well as their visualization styles are illustrated in TABLE I. In the context of MetaGME, most of the required information for meta-modeling and CASE tool implementation suggested in [26] is provided as default attributes of metamodels. All the metamodels defined with MetaGME contain predefined graphical presentation preferences and spatial position information, which can be customized and are handled by the GME tool implicitly during domain model design.

IV. Syntax Definitions and Meta-modeling

This section discusses the syntax definitions of our DSML. Firstly the formal syntax is given using the set theory, then the corresponding meta-models are described in the context of MIC. This paper doesn't elaborate on the definitions of basic FB (BFB), composite FB (CFB) and service interface FB (SIFB) because they have been extensively studied in related works. We focus on formalizing system configuration, task models, FB-to-task allocation and non-functional constraints.

The complete definitions start with the system model, as it's the topmost element in the IEC 61499 standard.

***Definition** 1 (System Model):* A system model Sys is a 4-tuple defined as: $Sys = \langle Dev, Seg, A_{sys}, L\rangle$, where:

- Dev is a non-empty finite set of device models;
- Seg is a link segment set containing various kinds of inter-device communication patterns. For example, in our current works, we define several attributes for EtherCAT segment, including transmission delay and jitter as well as synchronization error;
- A_{sys} denotes an application composed by IEC 61499 FB networks (FBN) in the scope of system model layer. A_{sys} is usually platform-agnostic in this layer and it can be refined as a set of application segments with respect to specific task models;
- L denotes a 3-arity relation linking the device models and the link segments, $L \subseteq \{\langle dev, seg, dev'\rangle \mid dev, dev' \in Dev, seg \in Seg, dev \neq dev'\}$.

***Definition** 2 (Device Model):* A device model in our DSML, dev, is defined as a 4-tuple: $dev = \langle DPara, Res, \Gamma, TR\rangle$, where:

- $DPara$ is a set of device specific parameters, e.g. communication settings and IEC 61499 management ID;
- Res is a finite set of resource models;
- Γ is a non-empty finite set of task models;
- TR stands for an allocating function between Γ and Res, $TR : \Gamma \rightarrow Res$, meaning that a task will be executed in a specific resource model.

***Definition** 3 (Resource Model):* A resource model defined in our DSML, res, is a 3-tuple: $res = \langle RPara, P, \Gamma_{res}\rangle$, where:

- $RPara$ is a set of parameters, such as the type of operating system, maximum RAM size and the type of IEC 61499 resource;
- P is an execution policy of FB network, currently only the buffered sequential execution model (BSEM) [21] is supported in our proposal;
- Γ_{res} denotes a set of tasks allocated to the resource.

The resource model is the abstraction of a hardware computing unit with several executing tasks. Such a definition is different from what it is in the IEC 61499 standard.

***Definition** 4 (Application Model):* The aforementioned concept A_{sys} and A'_i in a specific resource model i are application models A, which is defined as a 3-tuple: $A = \langle FBI, EC, DC\rangle$, where FBI is a finite set of FB instances; EC is an event connection function, $EC : \bigcup_{i\in FBI}(\{i\} \times EO^i) \rightarrow \bigcup_{d\in FBI}(\{d, \varepsilon\} \times (EI^d \cup \{\varepsilon\}))$; DC is a finite set of data connections, $DC : \bigcup_{i\in FBI}(\{i\} \times DI^i) \rightarrow \bigcup_{d\in FBI}(\{d, \varepsilon\} \times (DO^d \cup \{\varepsilon\}))$. We adopt the definitions of output/input events EI, EO and variable sets of output/input data DI, DO in [27] with a few adaptions.

***Definition** 5 (Variables):* Let DI, DO, and IV be sets of (names of) input, output and internal variables of FB, respectively. Let us denote $\overline{DI}$, $\overline{DO}$, and $\overline{IV}$ sets of variable states. For example, $\overline{DI} = \bigcup_{di\in DI}(\{di\} \times D^{di})$,where D^{di} is a domain of the variable di. Therefore, a "valued" variable is a pair in the form $\langle name, value\rangle$, e.g. $\langle di, v^{di}\rangle \in \overline{DI}$. Below we use the denotation $\overline{[\ ]}$ for a set of tuples which contains all combinations of variables' values (tagged by names of variables). For example, $\overline{[DI]} = \prod_{di\in DI}(\{di\} \times D^{di})$.

***Definition** 6 (System Configuration Model):* The concept of system configuration model K is defined as a set of device configuration models: $K = \bigcup_{i\in Dev} Conf_i$.

***Definition** 7 (Device Configuration Model):* The concept of device configuration model $Conf_{dev}$ for a specific dev is defined as a finite set of mapping functions: $Conf_{dev} = \bigcup_{j\in Res_{dev}} conf_j,\ conf_j : {\Gamma_j}' \rightarrow A'_j$
where:

- Γ'_j is a set of allocated task models in a specific resource model j;
- A'_j represents the allocated parts of application model in a specific resource model j;

Given $A_{sys} = \langle FBI_{sys}, EC_{sys}, DC_{sys}\rangle$, we define $A'_i = \langle FBI'_i, EC'_i, DC'_i\rangle$, $FBI'_i = FBI_i \cup FBI_{CI}$, $EC'_i = EC_i \cup EC_{CI}$, $DC'_i = DC_i \cup DC_{CI}$, where $FBI_i \in 2^{FBI_{sys}}$ is a set of allocated application parts from the system layer, $EC_i \in 2^{EC_{sys}}$, $DC_i \in 2^{DC_{sys}}$, while FBI_{CI} represents a set of platform-specific FB for inter-task communications and EC_{CI}/DC_{CI} are related connections;

***Definition** 8 (FB Instance):* A FB instance f is defined as a 2-tuple: $f = \langle n, T\rangle$, where n represents the unique name of f; T denotes the type definition of f, $T \in T_B \cup T_C \cup T_S$, which denotes the type definitions of BFB, CFB and SIFB respectively; The concrete formal definitions of FB types, along with the components of FB types, such as interface, ECC, algorithm, can be referred to [27].

***Definition** 9 (Task Model):* A task model in our DSML, tsk, is a 6-tuple defined as: $tsk = \langle TPara, I_e, O_e, C_I, C_O, NFR\rangle$,
where:

- $TPara$ is a set of real-time parameters, including priority, deadline, worst case execution time, period, etc.;
- I_e denotes a set of external inputs i_e of tsk, i_e is implemented as synchronization module (semaphore, event, etc.);
- O_e denotes a set of external outputs o_e of tsk, o_e is also implemented as synchronization module.
- C_I represents an external input connections function. For a task model tsk associated with a resource model j, the concerned part of application $A' = conf_j(tsk)$, let f_s be a FB instance and ei be an input event port of the interface list in f_s, we define $C_I : I_e \rightarrow \{\langle f_s, ei\rangle \mid f_s \in FBI^{A'} \wedge ei \in EI^{f_s}\}$.
- C_O represents an external output connections function. For a task model tsk associated with a resource model j, the concerned part of application $A' = conf_j(tsk)$, let

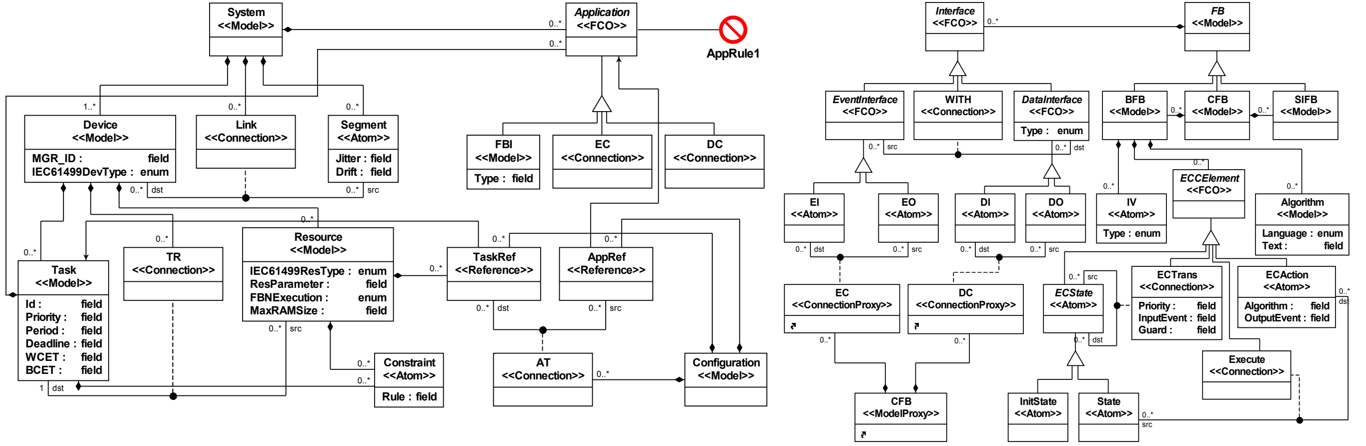


Fig. 2. The meta-models of our DSML in MetaGME notation (core parts)

f_e be a FB instance and eo be an output event port of the interface list in f_e, we define $C_O : \{\langle f_e, eo\rangle \mid f_e \in FBI^{A'} \wedge eo \in EO^{f_e}\} \rightarrow O_e$.

- NFR represents a set of non-functional requirements.

The core meta-models of our DSML are shown in Fig.2. Domain models defined in the M1 layer are concrete instances conformed to these meta-models. To be more specific, the left part of Fig.2 describes the core parts of our meta-models related with the formal definitions stated in this section, while the right part is the meta-models related to the function block type definitions. The rectangle symbols are classes defined according to the MetaGME paradigm, representing various kinds of syntactical elements.

Most of the MetaGME concepts are involved in our proposal, including *Model, Atom, FCO, Connection, Proxy* and *Reference*. Their functions are introduced in TABLE I. *Atom* and *Model* are mainly leveraged in our DSML for modeling the syntactical units. For example, we use *Model* to describe *System, Device, Resource, FB, System Configuration, Task*, and *Atom* is mainly used to specify *Variables*. The parameters of device, resource and task models are defined as attributes of the meta-models. *FCO* is utilized for defining *Application*, because it is regarded as a virtual concept in our approach, meaning that the concrete models contained in the system/task model are FBs and event/data connections instead. *Reference* is adopted for describing the counterparts of tasks or applications in multiple layers. *Proxy* is an auxiliary concept for describing the same object in the complicated meta-models. We employ *Connection* for specifying relations and functions in the formal syntax definitions. Specifically, we use this concept to model *Device Configuration* and *Task-Resource* allocating function (ref. to AT and TR class in Fig.2). Although MetaGME only allow metamodeling of relationships with a "Class-body-end" [28] representation, the mechanism of integrating user-defined attributes make it flexible enough for our DSML proposal. For example, one-to-one mapping can be simply achieved by setting cardinality constraints attribute, avoiding too much OCL rules specification.

Non-functional requirements are defined as *Atoms* contained in the task models. The textual attribute named "Rule" is used for specifying the concrete requirements in the form of Timed Computation Tree Logic (TCTL). Similarly, the structural semantics is defined as OCL rules associated with these class diagrams, represented by the circular symbol in Fig. 2. Due to space limits, not all the OCL rules are shown. These rules will be further elaborated in the following sections.

Based on the meta-models, the domain-specific modelling environment can be established automatically via the build-in MetaGME interpreter. During the modelling process, the type repository for device, resource as well as FB should be defined firstly. The system model can be composed using the model instances with respective to the types. The instantiation mechanism is also guaranteed by the GME tool.

V. Formal Semantic Definitions

In this section, both the structural and behavioral semantics are discussed formally. These semantics are described in terms of the concepts defined in Section IV.

A. Structural Semantics

This sub-section adopts OCL for formal specification of structural semantics. During the meta-modeling process, OCL equations can be adhered to related meta-models directly. In additions, the GME tool provides the options for determining how and when these OCL equations will be evaluated during the modeling process. For example, a specific OCL rule can be checked anytime automatically, or manually by user invocations, by setting the priority and triggering conditions of the OCL rule in the meta-model domain. Therefore, simple semantics analysis can be flexibly realized for the end users in the *M1* layer. Upgrades of the structural semantics will then require no modifications of the model artifacts of users. Therefore, dependability of models from the perspective of structural constraints can be attained in an extendable way. In [12], a similar configurable approach is proposed based on the ontology technology, which however requires additional transformation among different tools. Therefore, dynamic semantics analysis during the modeling process is not supported.

TABLE II
PARTS OF OCL EQUATIONS FOR STRUCTURAL SEMANTICS SPECIFICATIONS

<table>
<tr><th>No.</th><th>Constrained Object</th><th>Description</th><th>OCL Equation</th></tr>
<tr><td>(1)</td><td>ECC</td><td>Each ECC can have no more than one initial state.</td><td>self.parts(InitState)→size ≤ 1;</td></tr>
<tr><td>(2)</td><td>ECC State</td><td>Each EC state must have at least one entry and one exit ECTrans.</td><td>self.connectedFCOs(”dst”,ECTrans)→size >0 and self.connectedFCOs(”src”,ECTrans)→size >0;</td></tr>
<tr><td>(3)</td><td>ECTrans</td><td>No identical EC transition condition is allowed.</td><td>let parent:gme::Model = self.parent() in
  let sibling= parent.parts(ECTrans) in
    let guardcondition = sibling→select(a:ECTrans | a.Guard = self.Guard) in
    guardcondition→size ≤ 1;</td></tr>
<tr><td>(4)</td><td>Interface</td><td>The name should be unique in the FB.</td><td>let parent:gme::Model = self.parent() in
  let sibling= parent.atomParts()→select(a | a.kindName = self.kindName) in
    let nameofInf = sibling→select(a | a.name = self.name) in nameofInf→size ≤ 1;</td></tr>
<tr><td>(5)</td><td>Input Variable</td><td>The datatypes of both connected ports should be compliant.</td><td>let associateds = self.connectedFCOs(”src”,DC) in
  associateds→forAll(obj:DO | obj.Type = self.Type);</td></tr>
<tr><td>(6)</td><td>Application</td><td>The role of FB must be ”Instance” in Application.</td><td>self.parts()→select(p | p.kindName = ”FB”)→forAll(p | p.isInstance() = true)</td></tr>
</table>

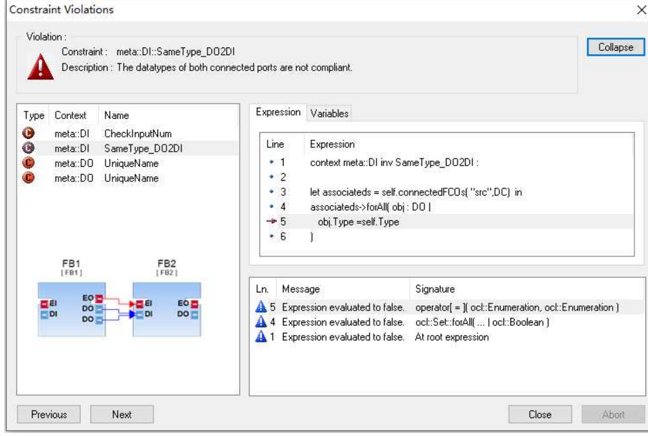


Fig. 3. Warning information of incompatible connection (STRING to BOOL)

In our current progress, two types of constraint rules are defined: cardinality constraints and type constraints. Cardinality constraints deal with a series of constraints on the valid amount of specific elements, while the type constraints limit the usage of types in the specific context, such as data connections in FBN, role of model in an application, etc. These constraints rules are mainly derived from the specification in the IEC 61499 standard. Parts of the developed rules are illustrated in Table II, where the build-in key word ***self*** denotes the constrained object itself. An example is described to illustrate how structural semantics analysis works in the context of MIC. The example application model is shown in Fig.3, where

$$A = \langle\{\texttt{FB1},\texttt{FB2}\},\\ \{\langle\texttt{FB1},\texttt{EO}\rangle \mapsto \langle\texttt{FB2},\texttt{EI}\rangle, \langle\texttt{FB2},\texttt{EO}\rangle \mapsto \langle\varepsilon,\varepsilon\rangle\},\\ \{\langle\texttt{FB2},\texttt{DI}\rangle \mapsto \langle\texttt{FB1},\texttt{DO}\rangle, \langle\texttt{FB2},\texttt{DI}\rangle \mapsto \langle\texttt{FB1},\texttt{DO}\rangle, \langle\texttt{FB1},\texttt{DI}\rangle \mapsto \langle\varepsilon,\varepsilon\rangle\}\rangle.$$

Several incompliances against IEC 61499 exist in this example. Such as multiple ports with the same name, redundant input connections for a variable ports. Regarding the type checking case, we let the data type for `FB1.DO` to be $STRING$ while `FB2.DI` as $BOOL$. According to the rules in Table II, four warnings related to structural semantics will be thrown after evaluations. The total rules are listed on the left of the warning information interface. The information on the top and right of the dialog box indicates the detailed explanations and evaluation results. The domain modelers can then efficiently locate the errors in their models.

B. Behavioral Semantics

This sub-section focuses on the formulation of formal behavior semantics to define the state evolutions of domain models when executed. The behavior semantics of FB, FBN based application as well as the task model are formalized with input/output LTS (IOLTS) [29]. Particularly, the formal execution models for BFB and FBN in [21] are leveraged.

IOLTS is a derivative of LTS, which is composed of system states and transitions between states. Transitions in a LTS are labelled with actions to describe the behaviors when the transitions fire. A labelled transition system is a 4-tuple $LTS = \langle Q, q_0, L, \rightarrow\rangle$, where Q is a non-empty set of states; $q_0 \in Q$ denotes the initial state; L is a set of actions that a LTS can perform; $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ means the transitions between states with the execution of certain actions, τ denotes the internal actions of a LTS, and $\tau \notin L$. $\langle q, a, q'\rangle \in \rightarrow$ is often written in the form of $q \xrightarrow{a} q'$.

IOLTS extends the actions of LTS by clarifying its input actions and output actions between the system and its environment. IOLTS can be defined as a five-tuple $\langle Q, q_0, L_I, L_O, \rightarrow\rangle$. L_I is a set of input actions while L_O the output actions, $L_I \cap L_O = \varnothing$. In our proposal, we define L_I as observable trigger conditions while L_O as observable emitted objects or effect functions.

In existing work around the behavior semantics of IEC 61499, finite state machine (FSM) is a widely adopted formal model. LTS shares many similarities with finite state machine mathematically. FSM is mainly used for validating sequential correctness according to specific rules while LTS may be more suitable for checking non-functional properties, such as liveness, safeness, etc. The intrinsic difference is that LTS does not necessarily have finite set of states and finite set of transitions like finite state machine. Therefore, a FSM can be seen as a LTS by mapping their counterparts respectively. Such a feature can facilitate reuses of the achievements on this topic for ensuring the rationality of our proposal.

***Definition** 10 (Behavior Semantics of BFB):* Given a BFB instance $B = \langle n_B, T_B\rangle$, with $T_B = \langle intf, ECC, IV, Alg\rangle$, (herein, $intf = \langle EI, EO, DI, DO, WI, WO\rangle$ is the interface definition of T_B, IV is a set of internal variables while Alg denotes a set of algorithms. Below, the objects relating to instance B will be marked with the superscript B. Further explanations regarding these concepts can be referred to [27]), the corresponding behavioral semantic model of B is defined

as:

$$S_{BFB} = \langle Q^B, q_0^B, L_I^B, L_O^B, \rightarrow^B \rangle,$$

where

- $Q^B = \{\langle l^B, v^B \rangle \mid l^B \in ES, v^B \in \overline{[DI]} \times \overline{[IV]} \times \overline{[DO]}\}$ is the state space of B, herein $l^B \in ES$ is the active state in the state set of ECC represented with ES while v^B is tuple containing concrete values of all variables.
- $q_0^B = \langle l_0^B, v_0^B \rangle, q_0^B \in Q^B$ is the initial state of B;
- $L_I^B = \{\langle ei^B, \overline{di^B} \rangle \mid ei^B \in EI \cup \{1\}, \overline{di^B} \in \overline{[DI]}\}$, EI is an input event set of B, 1 represents "always true";
- $L_O^B : ES \times VV \rightarrow ((\{B\} \times EO)^* \cup \{\varepsilon\}) \times VVO$, $VV = \overline{[DI]} \times \overline{[IV]} \times \overline{[DO]}$, $VVO = \overline{[IV]} \times \overline{[DO]}$, the * operation on the set $\{B\} \times EO$ represents a set of all finite strings of elements from this set;
- $\rightarrow^B \subseteq Q^B \times (L_I^B \cup L_O^B \cup \{\tau^B\}) \times Q^B$ represents a state transition of B.

The operational semantics of BFB can be defined as:

$$\begin{cases} \langle l^B, v^B \rangle \xrightarrow{\langle \varepsilon, \overline{di^B} \rangle} \langle l^B, v^B \rangle, & \textcircled{1} \\ \dfrac{EnableTrans(l^B, ei^B, \overline{di^B}, l^{B'}) = true}{\langle l^B, v^B \rangle \xrightarrow[L_O(l^{B'}, v^B)]{\langle ei^B, \overline{di^B} \rangle} \langle l^{B'}, v^{B'} \rangle}, & \textcircled{2} \\ \langle l^B, v^B \rangle \xrightarrow{\tau^B} \langle l^B, v^B \rangle. & \textcircled{3} \end{cases}$$

The case ① means that when only input variables arrive without occurring of input events, the state space of FB remains unchanged, neither output events nor output variables will be send, herein ε represents the absence of objects. The final case means that the state of B remain unchanged when internal actions perform. Specifically, transitions of IEC 6499 ECC Operation State Machine (OSM) during the execution process of BFB are treated as the internal actions. The case ② means that when the predicate $EnableTrans(l^B, ei^B, \overline{di^B}, l^{B'})$ holds under the inputs of ei^B and $\overline{di^B}$, the state of B will evolve from $\langle l^B, v^B \rangle$ to $\langle l^{B'}, v^{B'} \rangle$, with output action $L_O(l^{B'}, vv^B)$ being executed. To explain the second rule, we firstly define:

$$EnableTrans(l^B, ei^B, \overline{di^B}, l^{B'}) \triangleq Val(ei^B) \wedge g^{t^{ei^B}}(\overline{di^B})$$

where $ei^B \in EI \cup \{1\}$, $\overline{di^B} \in \overline{[DI]}$, $Val : EO \cup EI \cup \{1\} \rightarrow \{true, false\}$, g^t is a guard function associated with an ECC transition t and t^{ei^B} is an ECC transition t labelled with ei^B going out. Here, $g \in G$, G represents a set of all possible guard functions $g : \overline{[DI]} \rightarrow \{true, false\}$, $t \in T$ represents an element in the ECC transition set T, $T \subseteq ES \times (EI \cup \{1\}) \times G \times ES$. Specifically, $Val(1) = true,\ g(\emptyset) = true$.

We currently assume that only input data is involved in the guard conditions, and a transition is related to one input event at most. Besides, multiple transitions between two states with identical input events are not allowed.

Then, we define $CA \subseteq ES \times K$ for relating ECC states and EC actions K, $K = (Alg \cup \{\varepsilon\}) \times (EO \cup \{\varepsilon\})$, Alg represents a set of all algorithm functions alg, $alg : VV \rightarrow VVO$. Then, the function L_O^B, which updates variable states and generates sequences of output events, can be defined as $L_O^B : ES \times VV \rightarrow (\{B\} \times EO)^* \times VVO$ and calculated by the following algorithm:

inputs: l_B is a current state of ECC, and v^B is a tuple of current states of all variables of BFB instance B.
outputs: z is a sequence of event outputs from which signals will be outcome when executing the algorithm, and x is a resulted state of internal and output variables of the BFB instance.
1: $z \leftarrow \varepsilon$ // Construct an empty sequence
2: **foreach** $\langle l_B, k_{l_B} \rangle \in CA$ // Order of actions is not considered currently
3: **if** $alg^{k_{l_B}} \neq \varepsilon$ **then**
4: $x \leftarrow alg^{k_{l_B}}(v^B)$;
5: **end if**
6: **if** $eo^{k_{l_B}} \neq \varepsilon$ **then**
7: $Val(eo^{k_{l_B}}) \leftarrow true$; // Set output event
8: $z \leftarrow z + \langle B, eo^{k_{l_B}} \rangle$; // Add the output event to the sequence, "+" means concatenating a new character to the tail of a string
9: **end if**
10: **end foreach**
11: **return** $\langle z, x \rangle$

An execution procedure of a BFB instance B under the event and data input is defined as a function of run-to-complete steps consisting of multiple state transitions. It can be calculated by the following algorithm:

$$RTC_Step^B : EI \times VV \rightarrow (\{B\} \times EO)^*$$

assumption: 1)there is an initial execution queue $z_0 \in QE^*$ before starting the algorithm; 2) l^{B^+} is used both as an ordinary variable and as a bounded variable in logical expressions with a quantifier; 3) when calculating logical expressions with a quantifier, the destination ECC state is stored in l^{B^+}.
inputs: $ei^B \in EI$ is an event input with a signal to be active; $l_0^B \in ES$ is a current state of ECC from which it is run; $v_0^B \in VV$ is a tuple of current states of all variables of BFB instance B.
outputs: z is a sequence of event outputs from which signals will outcome when executing the possible ECC transitions.
1: $l^{B^-} \leftarrow l_0^B$; $v^B \leftarrow v_0^B$; $z \leftarrow z_0$ // Set initial conditions
2: **if** $\exists\ (l^{B^-}, ei^B, l^{B^+}) \in T : EnableTrans(l^{B^-}, ei^B, \overline{di^B}, l^{B^+})$ **then**
3: $Val(ei^B) \leftarrow false$ // Reset input event
4: **do**
5: $l^{B^-} \leftarrow l^{B^+}$ // Update current active ECC state
6: $\langle e, vvo \rangle \leftarrow L_O^B(l^{B^-}, v^B)$ // Run output action upon transition
7: Update values of v^B according to vvo by matching names of variable
8: $z \leftarrow z + e$ // Gather output event sequences
9: **until** $\exists\ (l^{B^-}, 1, l^{B^+}) \in T : EnableTrans(l^{B^-}, 1, \overline{di^B}, l^{B^+})$
10: **end if**
11: **return** z

Since we mainly concern the reachability of specific ECC states of FBs, we simply consider the state transition behaviors occurring under specific inputs. The comprehensive procedures including the data buffering mechanism, the transitions of OSM during execution and the formalized running process of algorithms can be referred to [30].

Before specifying the behavioral semantics of application model in the context of BSEM, the basic element of the buffered queue during execution should be identified.

***Definition** 11 (Execution Queue Element):* Given an application model $A = \langle FBI, EC, DC \rangle$, a set of execution queue elements is defined as: $QE = \bigcup_{j \in FBI} (j \times EO^j)$. EO^j is a set of output events in the interface list of FB instance j. Then, the execution queue can be defined as QE^*, where the * operation represents a set of all finite strings of elements from the set QE including the empty string. Let $eq = e_1 e_2 ... e_n (n \geq 0)$, $eq' = e_1 e_2 ... e_k (k \geq 0)$ be the substrings of QE^*, some related help functions of QE^* are defined as follows:

- $Dequeue : QE^* \rightarrow QE \times QE^*$, this function pops up the first element in the queue;
- $Enqueue : QE^* \times QE^* \rightarrow QE^*$ this function pushes sequences into the tail of queue;

- $Empty : QE^* \to \{true, false\}$, it can be calculated by the rule:

$$\begin{cases} Empty(eq) = true, if\ n = 0 \\ Empty(eq) = false, if\ n > 0 \end{cases}$$

- $SelectActiveFB : QE^* \times \overline{[DO]} \to \{\varepsilon\} \cup FBI \times EI \times \overline{[DI]}$, it can be calculated by the following algorithm:

assumption: the function of the values of FB instances output variables, and the function WI are considered known in the algorithm and not passed as parameters
inputs: $eq \in QE^*$ is the execution queue
outputs: k is an identifier of the active FB instance; ei^k is an event input with a signal to be processed; D is a tuple of valued input variables associated with ei^k.
1: **if** $\neg Empty(eq)$ **then** // The queue is not empty
2: $\langle e, eq'\rangle \leftarrow Dequeue(eq)$, $e = \langle i, ee^i\rangle, e \in QE$ // Pop up the first queued FB and output event
3: $\langle k, ei^k\rangle \leftarrow EC(i, ee^i)$ // Extract connected FB and input event
4: $di^k \leftarrow WI(ei^k)$ // $WI : EI \to 2^{DI}$, extract the input variables related with the "with" function
5: **foreach** $d^k \in di^k$
6: $\langle f, do^f\rangle \leftarrow DC(k, d^k)$ // Extract connected FB and output variable
7: $D \leftarrow D + \langle di^k, v^{do^f}\rangle$ // v^{do^f} is the value of do^f
8: **end foreach**
9: **return** $\langle k, ei^k, D\rangle$
10: **else return** ε

Definition *12 (Behavioral Semantics of Application):* The execution of an application model involves selection of active FB instance based on event and data propagations. Given an application model $A = \langle FBI, EC, DC\rangle$, with the buffered sequential execution mechanism, the corresponding behavioral semantic model can be defined as:

$$S_A = \langle Q^A, q_0^A, L_I^A, L_O^A, \to^A\rangle,$$

where

- Q^A represents the state space of A, $Q^A = \{\langle k^A, eq^A\rangle \mid k^A \in FBI \cup \{\varepsilon\}, eq^A \in QE^*\}$, k^A is the last active FB;
- $q_0^A = \langle \varepsilon, \varepsilon\rangle \in Q^A$ is the initial state of A;
- $L_I^A = \{\langle k^A, ei^k, \overline{di^k}\rangle \mid k^A \in FBI, ei^k \in EI^k, \overline{di^k} \in \overline{[DI^k]}\}$;
- $L_O^A : FBI \times EI \times \overline{[DI]} \to QE^*$;
- $\to^A \subseteq Q^A \times (L_I^A \cup L_O^A \cup \{\tau^A\}) \times Q^A$ represents a state transition of A.

The operational semantics of an application model can be defined as:

$$\begin{cases} q_0^A \xrightarrow[L_O^A(k^A, ei^k, \overline{di^k})]{\langle k^A, ei^k, \overline{di^k}\rangle} \langle k^A, eq^A\rangle & ① \\ \dfrac{l^A \leftarrow SelectActiveFB(eq^A),\ l^A = \langle k^{A\prime}, ei^{k\prime}, \overline{di^{k\prime}}\rangle}{\langle k^A, eq^A\rangle \xrightarrow[L_O^A(k^{A\prime}, ei^{k\prime}, \overline{di^{k\prime}})]{l^A} \langle k^{A\prime}, eq^{A\prime}\rangle}, & ② \\ \langle k^A, eq^A\rangle \xrightarrow{\tau^A} \langle k^A, eq^A\rangle. & ③ \end{cases}$$

Case ① specifies insertion of event source FB into the execution queue by the task related with this application model. This FB is connected to specific external inputs of the task model. In case ②, the output action function for updating execution queue can be calculated based on the following algorithm:

1: $ee \leftarrow RTC_Step^k(ei^k, v^k)$ // Execute active FB k, gather outputs
2: **return** ee

The execution procedure of an application model A initiated by the input action $\langle f_s, ei^{f_s}, \overline{di^{f_s}}\rangle$ is named as $AppExec$, it can be calculated by the following algorithm consisting of multiple state transitions:

1: $eq^A \leftarrow L_O^A(f_s, ei^{f_s}, \overline{di^{f_s}})$
2: $l^A = SelectActiveFB(eq^A)$
3: **while** $(l^A \neq \varepsilon)$
4: **do**
5: $ee \leftarrow L_O^A(l^A)$
6: $eq^A \leftarrow Enqueue(ee, eq^A)$
7: $l^A = SelectActiveFB(eq^A)$
8: **end while**
9: **return** ee // ee is the queued strings during the latest execution

In the algorithm presented above, Line 1-3 corresponds to the first case of state transition of A while Line 5-8 represent the second case. The returned value of this procedure will be utilized during the execution of a task model.

Definition *13 (Behavior Semantics of Task Model):* The execution of a task model includes the interactions with underlying operating system and the execution of FB application. Given a task model $t = \langle TPara, I_e, O_e, C_I, C_O, NFR\rangle$, the corresponding behavioral semantic model can be defined as:

$$S_t = \langle Q^t, q_0^t, L_I^t, L_O^t, \to^t\rangle$$

where

- Q^t represents the state space of t, $Q^t = S \times Q_A^t, S = \{idle, ready, executing, finished, error\}$ and Q_A^t is the state space of related application model A_τ ;
- $q_0^t = \langle idle, {q_{A0}}^t\rangle \in Q^t$ is the initial state of tsk, ${q_{A0}}^t \in Q_A^t$.
- $L_I^t = \{\delta\} \cup I_e \cup {L_I}^A$, ${L_I}^A$ is the input actions of A_τ and δ represents the system call from operating system;
- $L_O^t = {L_O}^A \cup O_e$, ${L_O}^A$ is the output actions of A_τ;
- $\to^t \subseteq Q^t \times (L_I^t \cup L_O^t \cup \{\tau^t\}) \times Q^t$ represents the state transition of tsk under certain inputs, and five common cases of transitions may exist:

$$\begin{cases} \langle idle, q_A^t\rangle \xrightarrow{\delta} \langle ready, q_A^t\rangle, & ① \\ \langle ready, q_A^t\rangle \xrightarrow[L_O^A(f_s, ei, \varepsilon)]{i_e} \langle executing, {q_A^t}'\rangle, i_e \in I_e, & ② \\ \dfrac{l^A \leftarrow SelectActiveFB(eq^{A_\tau}),\ l^A = \varepsilon}{\langle executing, q_A^t\rangle \xrightarrow[o_e]{} \langle finished, q_A^t\rangle}, o_e \in O_e, & ③ \\ \dfrac{l^A \leftarrow SelectActiveFB(eq^{A_\tau}),\ l^A = \langle k^{A\prime}, ei^{k\prime}, \overline{di^{k\prime}}\rangle}{\langle executing, q_A^t\rangle \xrightarrow[L_O^A(k^{A\prime}, ei^{k\prime}, \overline{di^{k\prime}})]{l^A} \langle executing, {q_A^t}'\rangle}, & ④ \\ \langle s, q_A^t\rangle \xrightarrow{\tau^t} \langle s, q_A^t\rangle, s \in S. & ⑤ \end{cases}$$

The execution procedure of a task model can be calculated by the following algorithm:

1: $s \leftarrow idle, q_A \leftarrow q_{A0}^t$ // Initial Conditions
2: $P(\delta)$ // Await system call (PV operation, P: pending, V: posting)
3: $s \leftarrow ready$
4: **foreach** $i_e \in I_e$ // The order of i_e is not considered
5: $P(i_e)$ // Await specific external input
6: $\langle f_s, ei^{f_s}\rangle = C_I(i_e)$ // Extract related FB instance and input event
7: $s \leftarrow executing$
8: $Z \leftarrow AppExec(f_s, ei^{f_s}, \varepsilon)$
9: **foreach** z of $Z, z = \langle f_e^z, eo^{f_e^z}\rangle$
10: $o_e \leftarrow C_O(f_e^z, eo^{f_e^z}),\ V(o_e)$ // Extract and post related external output
11: **end foreach**
12: $s \leftarrow finished$
13:**end foreach**

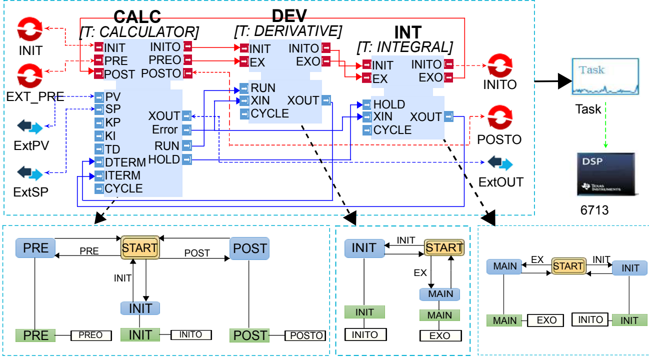


Fig. 4. A PID application modeled with our DSML

where Line 1-3 corresponds to the first case of state transition , Line 4-8 represent the second and the fourth case while Line 9-12 are related to the third case.

The formal behavioral semantics can be the guidelines for the implementation phase activities, such as automatic code generation and execution environment establishment, which will not be elaborated in this article as design phase activities are the main concerns. Since that domain models conformed to the formal behavior semantics described in this section cannot be directly verified using the available tools, a model-to-model transformation approach is therefore required.

VI. Automated Formal Verification Through Model Transformation

In this section, formal verification of models composed by our DSML is discussed. A model checking approach based on the timed automaton (TA) is presented. Particularly, the graphical model checking tool, UPPAAL, is employed for this purpose. Multi-layered time automata networks are automatically generated from the source model in the *M1* layer, facilitating formal verification on schedulability and liveness of task models as well as reachability of FB models. A simple application for implementing PID calculation using our proposed DSML will be described as the illustrative example in this section. The example is shown in Fig. 4.

A TA can be defined as a 6-tuple: $\langle L, l_0, C, V, A, E\rangle$, where L is a set of locations, l_0 is the initial location, C is a set of real-valued clocks, V is a set of Boolean, channel or integer variables defined in the form of $\langle name, value\rangle$, A is a set of actions, $E \subseteq L \times 2^A \times EVAL \times L$ is a set of edges between locations, where $EVAL$ is a set of all possible functions: $eval : [\overline{V}] \times [\overline{C}] \to \{true, false\}$. In the context of UPPAAL, the actions $A = A_{sync} \cup A_{update}$, where A_{sync} denotes the synchronization actions between TA, and A_{update} is a set of updating functions over $[\overline{V}] \times [\overline{C}]$. A TA network $\mathcal{T}$ is simply defined as a set of TA, $\mathcal{T} = \{ta_1, ta_2, ..., ta_n\}$. The semantics of TA can be specified with a LTS [31].

Finally, the complete UPPAAL model can be defined as: $UM = \langle GV, \mathcal{T}, \mathcal{I}\rangle$, where GV is a collection of variables, including clock type, channel type, etc., $\mathcal{I} = \langle PI, \prec\rangle$ is the ordered set of TA model instances PI inherited from $\mathcal{T}$, the orders of TA models determine their priorities.

Algorithm 1: Transforming Tasks Set to UPPAAL model

inputs: A set of tasks Γ related to a device model dev.
output: A populated model $UM = \langle GV, \mathcal{T}, \mathcal{I}\rangle$.

1 **procedure** $TS2TA(\Gamma)$
2 $GV \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$;
3 Create a scheduler TA, $ta_1 := \langle L, l_0, \emptyset, \emptyset, A^{ta_1}, E^{ta_1}\rangle$;
4 $L = \{idle, scheduling, executing\}, l_0 = idle$;
5 $\mathcal{T} \leftarrow \mathcal{T} \cup \{ta_1\}, \mathcal{I} \leftarrow \mathcal{I} \cup \{I^{ta_1}\}$; // Add new instance to the end of $\mathcal{I}$
6 **foreach** $\tau \in \Gamma$ **do**
7   Create a variable id as the index of τ, and two channel variables cv_1, cv_2 as the flags of starting/finishing execution, $GV \leftarrow GV \cup \{id, cv_1, cv_2\}$;
8   Create a compound variable gv containing sub-variables respective to the attributes of $TPara^{\tau}$, $GV \leftarrow GV \cup \{gv\}$;
9   Create two actions a_{sync1}, a_{sync2} awaiting/posting the starting/finishing flag, based on cv_1, cv_2;
10   Create a pair of edges, $e : executing \xrightarrow{a_{sync1}} scheduling$, $e' : scheduling \xrightarrow{a_{sync2},\ eval(gv)} executing$;
11   Create an action a_{sync3} and a pair of edges, $e_1 : scheduling \xrightarrow{eval(gv)} idle$, ${e_1}' : idle \xrightarrow{a_{sync3}} scheduling$; // Awaiting periodic system call
12   $A^{ta_1} \leftarrow A^{ta_1} \cup \{a_{sync1}, a_{sync2}, a_{sync3}\}$;
13   $E^{ta_1} \leftarrow E^{ta_1} \cup \{e, e', e_1, {e_1}'\}$;
14   Create a TA for τ, $ta_{\tau} := \langle L', l'_0, C', V', A', E'\rangle$; $L' = \{idle, ready, executing, finished, error\}, l'_0 = idle$
15   Update A', E' according to τ;
16   $\mathcal{T} \leftarrow \mathcal{T} \cup \{ta_{\tau}\}, \mathcal{I} \leftarrow \mathcal{I} \cup \{I^{ta_{\tau}}\}$; // Adjust orders of elements in $\mathcal{I}$ based on their priorities specified in $TPara^{\tau}$
17 **return** UM

The proposed model transformation approach will produce several UPPAAL models: a task set verification model and a series of FBN verification models respective to each task model. The FBN verification model consists of several TA for verifying FBs in the FBN. During the transformation process, the behavior semantics of FB, FBN and task are derived in the form of time automata. Since they have the same mathematical definitions based on LTS, the verification of these time automata networks can also prove the non-functional properties of the source models, assuming that the model-to-model mapping procedure is correct.

Algorithm 1 shows the main mapping rules for populating a UPPAAL model with the input of a specific device model containing a set of task models. Several Task TA will be created to abstract the temporal execution procedure of the input task set. These tasks are scheduled with a policy called Rate Monotonic Scheduling, which is modeled by the *Scheduler* TA. Comprehensive modeling of RM-based scheduler is out of the scope of this paper, therefore we will not elaborate on it. Algorithm 2 will populate a UPPAAL model from a single task, with internally calling Algorithm 3 for transforming FB instances in the task model to corresponding TA models. The *Event dispatcher* TA model generated by Algorithm 2 abstracts the $AppExec$ function while those by Algorithm 3 is respective to the RTC_Step function. Currently, only BFB instances can be transformed while SIFB instances should be added manually. The generated TA can be utilized in UPPAAL for simulating the logical behaviors. Furthermore, together with the TCTL rule files, formal verification of the concerned non-functional properties can be achieved. These algorithms are implemented using the graph rewriting and transformation language, a model transformation language provided as a part of the MIC toolchain [32].

Algorithm 2: Transforming Task to UPPAAL model

inputs: A task model τ and related application model A_τ figured out by the device configuration model.
output: A populated model $UM = \langle GV, \mathcal{T}, \mathcal{I} \rangle$.
1 **procedure** $T2TA(\tau, A_\tau)$
2 **foreach** $f \in FBI^{A_\tau}$ **do**
3 Create a FB TA ta_f, $ta_f \leftarrow FB2TA(f)$; // Only BFB is supported
4 $\mathcal{T} \leftarrow \mathcal{T} \cup \{ta_f\}$;
5 $\mathcal{I} \leftarrow \mathcal{I} \cup \{I^{ta_f}\}$; // Add new instance to the end of $\mathcal{I}$
6 Create an event dispatcher TA ta_e, $ta_e := \langle L, l_0, \{c_0\}, \{v_0\}, A^{ta_e}, E^{ta_e} \rangle$, $L = \{l_0\}, l_0 = idle$, $A^{ta_e} = \{a_{sync0}, a_{update0}\}$, $E^{ta_e} = \{e_0\}, e_0 : idle \xrightarrow{a_{sync0}, a_{update0}} idle$;
7 $\mathcal{T} \leftarrow \mathcal{T} \cup \{ta_e\}$, $\mathcal{I} \leftarrow \mathcal{I} \cup \{I^{ta_e}\}$; // Add new instance to the end of $\mathcal{I}$
8 **foreach** $ec \in EC^{A_\tau}$ **do**
9 Create a channel variable cv and a integer variable $nIdx$ as the index of ec, $GV \leftarrow GV \cup \{cv, nIdx\}$;
10 Create a location l, $L^{ta_e} \leftarrow L^{ta_e} \cup \{l\}$;
11 Create a synchronization action a_{sync} awaiting the occurrence of cv, and a update action a_{update} for enqueue operation of $nIdx$;
12 $A^{ta_e} \leftarrow A^{ta_e} \cup \{a_{sync}, a_{update}\}$;
13 Create an edge $e : idle \xrightarrow{a_{sync}, a_{update}} l$, and the reverse edge $e' : l \rightarrow idle$;
14 $E^{ta_e} \leftarrow E^{ta_e} \cup \{e, e'\}$;
15 **foreach** $dc \in DC^{A_\tau}$ **do**
16 Create v, $GV \leftarrow GV \cup \{v\}$, v is related to the input variable of dc;
17 **foreach** $i_e \in {I_e}^\tau \, and \, o_e \in {O_e}^\tau$ **do**
18 Update ta_e based on C_I and C_O as lines 9-14 do;
19 **return** UM

Algorithm 3: Transforming BFB to TA

inputs: A BFB model f.
output: A populated model $ta = \langle L, l_0, C, V, A, E \rangle$.
1 **procedure** $FB2TA(f)$
2 **foreach** $es \in ES^f$ **do** // ES denotes the ECC states of f
3 Create a location, l, $L \leftarrow L \cup \{l\}$;
4 **foreach** $ea \in K^{es}$ **do** // K: Actions related to es
5 Create a location, l', $L \leftarrow L \cup \{l'\}$;
6 Create an edge e, a synchronization and an update actions a_0, a_1, based on ea, l, l'; $E \leftarrow E \cup \{e\}$; $A \leftarrow A \cup \{a_0, a_1\}$;
7 **foreach** $et \in ET^f$ **do** // ET denotes the state transitions
8 Create an edge e, a synchronization and an update actions a_0, a_1, based on et and related guard condition;
9 $E \leftarrow E \cup \{e\}$; $A \leftarrow A \cup \{a_0, a_1\}$;
10 **foreach** $fbv \in DI^f \cup DO^f \cup IV^f$ **do**
11 Create a variable v based on fbv, $V \leftarrow V \cup \{v\}$;
12 **return** ta

The example shown in Fig. 4 mainly contains a single task model and a FB application composed by 3 interconnected FB instances. The task model is mapped to a resource model representing the DSP-based hardware platform running the DSP/BIOS operating system. The corresponding ECCs of each FBI are shown in the bottom part of Fig. 4.

By applying the model transformation algorithms stated in this section, several TA models can be achieved, as shown in Fig. 5. Since there is only one task model in the example, the generated task set verification models shown in Fig. 5(a) contains only one TA for the task, and only one pair of transitions exists between the *scheduling* and *executing* location of the *Scheduler* TA. Detailed implementation of the line 15 in Algorithm 1 (e.g., constructions of actions and transitions) is based on a template TA similar to Task TA in Fig. 5(a). Multiple Task TA models will share the same location sets and edge sets, only differing in the value of the id.

Fig. 5(b) represents the event dispatcher TA related to the FB application contained in the task model. The corresponding TA models for each FBI are given in Fig. 5(c). These models are generated with Algorithm 2 and Algorithm 3. In detail, a_{sync0} in the line 6 of Algorithm 2 is corresponding to the synchronization action `"event[front]!"` in Fig. 5(b), which means posting the top element of the execution queue. Meanwhile, $a_{update0}$ in the line 6 of Algorithm 2 is related to the update action `"dequeue(), isInit=1"` in Fig. 5(b), which removes the top element from the execution queue. In Fig. 5(c), the anonymous locations are created by the code snippet from line 4-6 in Algorithm 3. Transition from named locations representing specific ECC states to these anonymous locations will execute synchronization and update actions with respective to sending output event and executing algorithm function. For the code snippet from line 7-9 in Algorithm 3, the source locations of the created edges will be the anonymous locations related with the corresponding source ECC states.

Basically, the first property can be verified for the generated models is the schedulability by checking whether the predicate "**AG** ¬ *deadlock*" holds, which means for all state transition traces, no deadlock will happen. Reachability of specific states of FBs can also be verified with the predicate that "**EF** *FB.fb_state*" is true, which means that exists a state transition trace where the state `fb_state` of `FB` will be eventually reached. Compared with the work of [17], we additionally provide analysis for the schedulability of concurrent tasks, which will be elaborated in the next section.

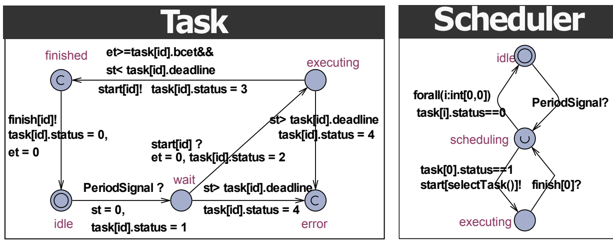
VII. Design of a CNC system: A case study

In this section, a distributed CNC system is modeled and verified using the proposed DSML and supporting modeling environment. Typically, the working process of a CNC system starts with G-Code files parsing and ends in motion control of corresponding axes to generate desired trajectory. The motion control process can be further divided into supervisory control, trajectory planning and axes position control. These functions will be deployed to distributed control nodes in this case study.

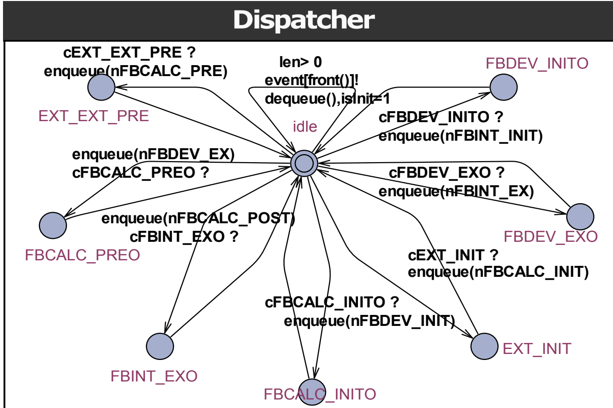
A. System Modeling

During the design process of the example CNC system, the platform, function and configuration aspects are synthetically considered. With respect to the platform aspect, an ARM-based Beaglebone Black Board (referred as *dev1*) and a DSP-based motion control card (referred as *dev2*) is leveraged for executing CNC tasks. For each device model, a single resource model is contained inside (ARM Cortex-A8 for *dev1* and TMS320C6713 for *dev2*). An industrial PC (IPC) is employed for implementing information exchange between these two device models. The IPC is connected to *dev1* via TCP/IP protocol, and *dev2* is plugged into the IPC through PCI bus. Since the IPC device is only used for forwarding motion control commands from *dev1* to *dev2*, modeling of its functions and configuration will not be elaborated in this example.

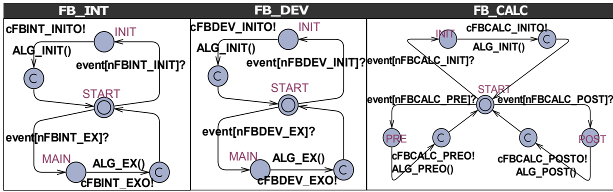
As to the function and the configuration aspect, application development and task allocation are carried out. The platform



(a) The generated TA models by Algorithm 1

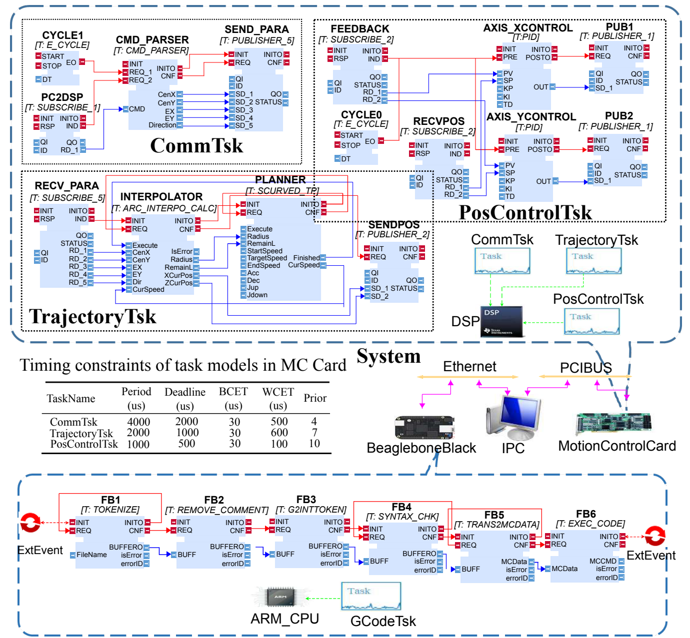


(b) The generated TA model by Algorithm 2



(c) The generated TA models by Algorithm 3

Fig. 5. The generated UPPAAL models for verifying the PID application



Timing constraints of task models in MC Card

TaskName	Period (us)	Deadline (us)	BCET (us)	WCET (us)	Prior
CommTsk	4000	2000	30	500	4
TrajectoryTsk	2000	1000	30	600	7
PosControlTsk	1000	500	30	100	10

Fig. 6. A prototype CNC system and its FB-based task models

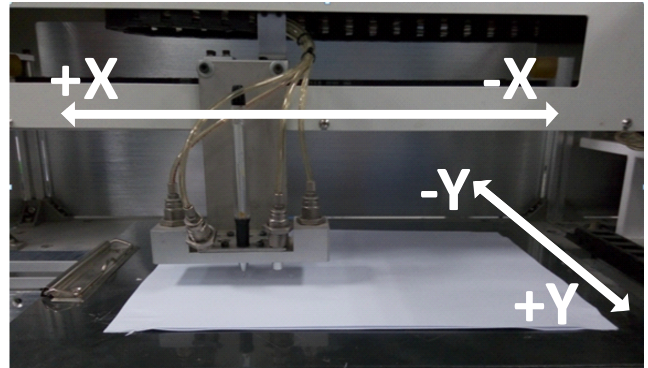


Fig. 7. The established X-Y testbed for the case study

independent application is modeled with the pre-constructed FB libraries in the system layer. The developed application is allocated layer by layer to the task models that will be scheduled on device models. The non-realtime Linux system and real-time DSP/BIOS system are chosen for CPUs of *dev1* and *dev2*, respectively, to execute the contained tasks. In this example, both non-realtime and real-time tasks are defined. The G-Code parsing task is deployed in *dev1* while the motion control tasks are in *dev2*. The overall platform aspect of the final system model and the task models are shown in Fig. 6. The generated control kernel is validated on a prototype X-Y testbed established as Fig. 7 shows.

In this example, the FB instances *CMD_TRANSLATOR1* will interpret motion command packet received from *PC2DSP*. The concrete motion parameters are sent to *TrajectoryTsk* through *SEND_PARA*. The main functioning FBIs in this task are *INTERPOLATOR* and *PLANNER*. The former FBI tackles with post-processing of the intermediate set-points calculated by the latter FBI. These intermediate set-points are a series of desired velocity values generated by the S-Curved velocity planning algorithm contained in the *PLANNER* FBI. Therefore, the pattern of acceleration/deceleration control before interpolation (ADCBI) can be implemented. The processed velocity set-points are then sent to the final output task *PoscontrolTsk*. The aforementioned PID application is utilized in this task in the form of CFB (the FBI *AXISX_CONTROL* and *AXIS_YCONTROL*), which will send the intermediate position to the external drives during each servo cycle.

B. Non-Functional and Functional Validations

The real-time parameters of these task models are also listed in Fig. 6. These settings will be verified according to the criteria defined in the constraint atoms contained in the task models. Parts of these criteria are listed in Table III. In detail, Rule 2-4 for FB TA can verify the reachability of concerned states in the corresponding FBIs.

With respect to verification of task models, both symbolic and statistical model checking methods are adopted in the schedulability analysis, as Rule 1 and Rule 2 for task TA in Table III presents. Symbolic model checking (e.g., Rule 1) is suitable for verifying simple TA models precisely, while the statistical methods (e.g., Rule 2, which estimates probability of whether the predicate "**F** *TrajectoryTsk.error* $\vee$ *CommTsk.error* $\vee$ *PosControlTsk.error*" holds over the range of 0-

200000 unit time) can tackle with complicated models under specific probability and uncertainty. With Rule 2, UPPAAL can estimate the probability of scheduling failure indicated by the error state of tasks. The verification results of Rule 2 for task TA are shown in Table IV. At last, Rule 3 for task TA is mainly related to the reachability property of the execution state for the task model, while rule 4 is related with the safety property.

The list of task parameters given in Fig. 6 can pass all the proposed verification rules, which means that this group of settings can be applied during the implementation phase. Currently the worst-case execution time of FBIs are determined by the profiling plug-in in the DSP/BIOS developing tool. Some advanced model-level timing analysis (e.g., [33] and [34]) may be adopted in future work.

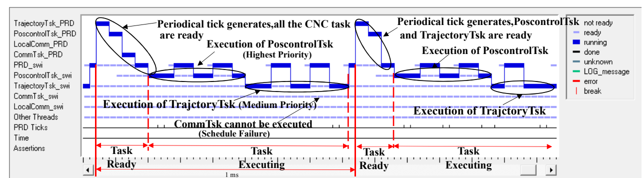
TABLE III
PARTS OF THE VERIFICATION RULES

UPPAAL Models	No.	Verification Rules (in the form of CTL)
Function Block TA	1	**AG** $\neg$ *deadlock*
	2	**EF** *FB_INT.MAIN*
	3	**EF** *FB_DEV.MAIN*
	4	**EF** *FB_CALC.POST*
TaskSet & Task TA	1	**AG** $\neg$ *deadlock*
	2	$Pr[\leq 200000]$ (**F** *TrajectoryTsk.error* $\vee$ *CommTsk.error* $\vee$ *PosControlTsk.error*)
	3	**EF** *TrajectoryTsk.executing*
	4	**AG** *TrajectoryTsk.error* $\Rightarrow$ *TrajectoryTsk.st* $>$*TrajectoryTsk.deadline*

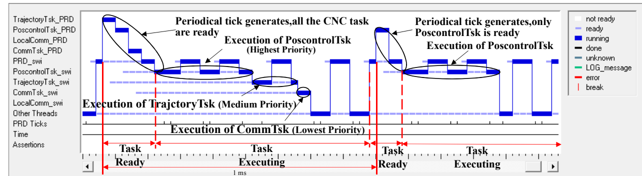
A preliminary implementation of the modeled CNC system is established based on the code generators proposed in [35]. The execution graphs of the real-time tasks in the MC card device with different timing settings are represented in Fig. 8. They are recorded using the Code Composer Studio tool. All the tasks are periodic, which can be implemented by the combination of periodic object (PRD) and software interrupt object (SWI) provided in the DSP/BIOS system. Each shot of the periodic task starts with the triggering of PRD handlers. Then, the PRD handlers will enable the corresponding preconfigured SWI objects, which contain the concrete functions of the task. As can be inferred from the execution graphs, the period settings used by Fig. 8(a) will result in scheduling failure, because the task with lowest priority *CommTsk* cannot be executed normally during every scheduling cycle. Scheduling failures of motion control system will result in delayed response of motion command or even uncontrolled quick stop of motor, causing unpredictable dangers. These settings are also verified according to the rules in Table III, and rule 1 for Task TA return false. The passed task parameter settings

TABLE IV
PROBABILITIES OF SCHEDULING FAILURE IN THE CNC EXAMPLE

No.	Confidence $(1-\alpha)$	Uncertainty (ε)	Probability
1	0.95, α=0.05	0.05	[0, 0.0973938]
2	0.95, α=0.05	0.01	[0, 0.019956]
3	0.99, α=0.01	0.05	[0, 0.0986743]
4	0.99, α=0.01	0.01	[0, 0.0199441]



(a) Execution graph of motion control tasks with the period settings: 4000/1000/1000 us for CommTsk/TrajectoryTsk/PosControlTsk



(b) Execution graph of motion control tasks with the period settings: 4000/2000/1000 us for CommTsk/TrajectoryTsk/PosControlTsk

Fig. 8. Execution graph of the real-time tasks in the MC card device

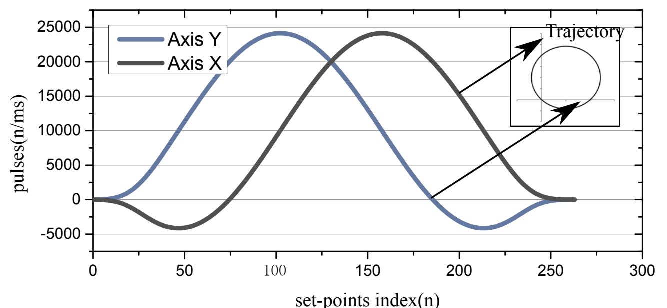


Fig. 9. The output set-points of Axis X and Axis Y and the trajectory

listed in Fig. 6, however, can ensure scheduling correctness of all tasks, as can be inferred from Fig. 8(b). In this sense, our approach can meet the requirement of dependability at the early design stage, which cannot be provided by traditional methods.

The functional correctness is then proved by a use case. A G-Code file for generating circular trajectory is transmitted to *dev1* which executes the parsing task. Following that, the motion control tasks are executed. The sampled position set-points of both Axis X and Axis Y generated by *dev2* are presented in Fig. 9, along with the desired circular trajectory. According to the results of functional and non-functional verifications, the feasibility of our proposal can be proved in terms of the capability of dependable design of a low-level automation control system.

C. Quantitative Assessments of Generated Code

Finally, quantitative assessments of our approach are conducted regarding the capability of taming design complexity. Since there are no common metrics for evaluating domain model complexity, we carry out quantitative assessments of the generated code instead. The generated code of the MIC approach is evaluated, compared with our legacy manual coded motion control kernel software, using several software engineering metrics, including 1) *CountDeclFileCode*, 2) *AvgCyclomatic*, 3) *MaxCyclomatic*, 4) *MaxNesting* and 5)

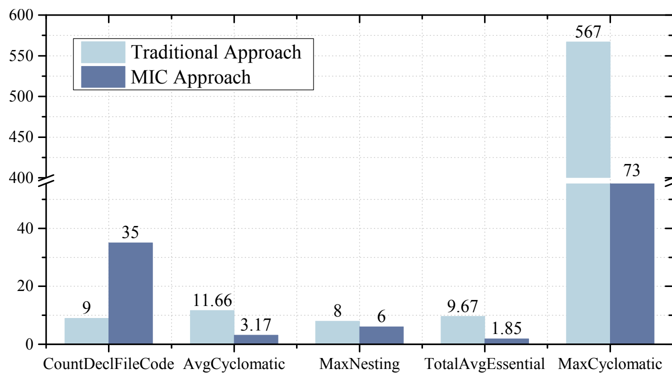


Fig. 10. Metrics for complexity of the source code

TotalAvgEssential. In detail, Metric 1 is the total number of source files; Metric 2 is the average cyclomatic complexity for all nested functions or methods; Metric 3 is the maximum cyclomatic complexity; Metric 4 is the maximum nesting level of control constructs in all functions while Metric 5 is the average essential complexity. These complexity metrics can indicate the modularity and maintainability of the program.

In this paper, the metrics are evaluated using a commercial code analysis tool, Understand. The assessment results are shown in Fig. 10. It is shown that almost all metrics have better values for our approach, meaning that the complexity can be effectively reduced using the proposed MIC approach, in spite of the increase of the total source files. Since our code generator will generate a source and head file for each models, the *CountDeclFileCode* can reflex the complexity of models. Higher modularity is achieved by decomposing the monolithic motion control software into multiple FB-based tasks. Thus, complexity of the main components, such as trajectory planner, interpolator, etc., can be reduced. In this sense, the effectiveness of the proposed solution can be proved regarding industrial automation systems design.

VIII. Conclusion and Future work

This paper presents a model-based solution to design low-level control system applied to the industrial automation domain. The main goal is to enhance the abstraction level, dependability as well as flexibility of the design process. The MIC approach is adopted, and accordingly a domain-specific modeling language and the corresponding modeling environment is developed. The IEC 61499 standard is leveraged as the reference for defining meta-models of our DSML. Both formal structural and behavioral semantics are explicitly specified as the ground of formal model verifications. To ensure flexibility, both syntax and semantics definitions are encoded as external XML files of a meta-programmable modeling environment. Verifications of the modeled artifacts with respect to non-functional and functional properties are attained by a model-to-model transformation approach. A case study is demonstrated to exhibit the feasibility of the proposed solution.

Future work in this direction is envisaged as follows:

- The ontology technology can be integrated in the meta-model layer for achieving more advanced semantic analysis;
- Correctness-proof on the model transformation procedure can be conducted for validation on the preservation of concerned properties;
- Model-based simulation in the early development stage can also be achieved through similar transformation approaches. Particularly, simulation of the communication process is desired.
- Evaluation of other formal languages with possibly higher industrial maturity of supporting tools.

References

[1] G. Pritschow, K.-H. Wurst, C. Kircher, and M. Seyfarth, “Control of reconfigurable machine tools,” in *Changeable and Reconfigurable Manufacturing Systems*, pp. 71–100, Springer, 2009.

[2] V. Lesi, Z. Jakovljevic, and M. Pajic, “Towards plug-n-play numerical control for reconfigurable manufacturing systems,” in *Proc. IEEE 21st Int. Conf. Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, Sept. 2016.

[3] D. Yu, Y. Hu, X. W. Xu, Y. Huang, and S. Du, “An open CNC system based on component technology,” *IEEE Transactions on Automation Science and Engineering*, vol. 6, pp. 302–310, Apr. 2009.

[4] S. Wang and K. G. Shin, “Constructing reconfigurable software for machine control systems,” *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 475–486, Aug. 2002.

[5] *IEC 61499-1:2012, Function Blocks*. Geneva, Switzerland: International Electrotechnical Commission, 2012.

[6] *IEC 61131-3:2013, Programmable Controllers, Part 3: Programming Languages*. Geneva, Switzerland: International Electrotechnical Commission, 2013.

[7] V. Vyatkin, “IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 7, pp. 768–781, Nov. 2011.

[8] F. Serna, C. Catalan, A. Blesa, J. M. Rams, and J. M. Colom, “Control software design for a cutting glass machine tool based on the COSME platform. case study,” in *Proc. IEEE Int. Conf. Automation Science and Engineering*, pp. 501–506, Aug. 2011.

[9] A. Blesa, C. Cataln, F. Serna, J. M. Colom, C. Larrea, and J. M. Rams, “Function blocks for the design of control applications based on EtherCAT fieldbus,” in *Proc. IEEE Int. Conf. Industrial Technology (ICIT)*, pp. 1876–1883, Mar. 2015.

[10] M. Minhat, V. Vyatkin, X. Xu, S. Wong, and Z. Al-Bayaa, “A novel open cnc architecture based on step-nc data model and IEC 61499 function blocks,” *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 3, pp. 560 – 569, 2009.

[11] C. Pang, S. Patil, C. W. Yang, V. Vyatkin, and A. Shalyto, “A portability study of IEC 61499: Semantics and tools,” in *Proc. 12th IEEE Int. Conf. Industrial Informatics (INDIN)*, pp. 440–445, July 2014.

[12] W. Dai, V. N. Dubinin, and V. Vyatkin, “Automatically generated layered ontological models for semantic analysis of component-based control systems,” *IEEE Transactions on Industrial Informatics*, vol. 9, pp. 2124–2136, Nov. 2013.

[13] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, “Composing domain-specific design environments,” *Computer*, vol. 34, pp. 44–51, Nov. 2001.

[14] V. Vyatkin, H. M. Hanisch, C. Pang, and C. H. Yang, “Closed-loop modeling in future automation system engineering and validation,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, pp. 17–28, Jan 2009.

[15] S. Patil, V. Dubinin, and V. Vyatkin, “Formal modelling and verification of iec61499 function blocks with abstract state machines and smv - execution semantics,” *Dependable Software Engineering: Theories, Tools, and Applications*, pp. 300–315, Jan. 2015.

[16] L. H. Yoong and P. S. Roop, “Verifying IEC 61499 Function blocks using Esterel,” *IEEE Embedded Systems Letters*, vol. 2, pp. 1–4, Mar. 2010.

[17] M. Stanica and H. Guéguen, “Using timed automata for the verification of IEC 61499 applications,” in *IFAC Workshop on Discrete Event Systems (WODES’04)*, pp. 375–380, 2004.

[18] G. S. Doukas and K. C. Thramboulidis, “A real-time Linux execution environment for function-block based distributed control applications,” in *Proc. INDIN ’05. 2005 3rd IEEE Int. Conf. Industrial Informatics*, pp. 56–61, Aug. 2005.

[19] A. Zoitl, R. Smodic, C. Sunder, and G. Grabmair, "Enhanced real-time execution of modular control software based on IEC 61499," in *Proc. IEEE Int. Conf. Robotics and Automation ICRA 2006*, pp. 327–332, May 2006.

[20] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L. M. Pinho, "A real-time semantics for the IEC 61499 standard," in *Proc. IEEE 20th Conf. Emerging Technologies Factory Automation (ETFA)*, pp. 1–6, Sept. 2015.

[21] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 Applications, part b: Execution semantics," *IEEE Transactions on Industrial Informatics*, vol. 6, pp. 145–154, May 2010.

[22] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Transactions on Computers*, vol. 58, pp. 1599–1614, Dec 2009.

[23] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, pp. 110–111, Apr. 1997.

[24] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, "Composition and cloning in modeling and meta-modeling," *IEEE Transactions on Control Systems Technology*, vol. 12, pp. 263–278, Mar. 2004.

[25] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.

[26] I. Garca-Magario, R. Fuentes-Fernndez, and J. J. Gmez-Sanz, "A framework for the definition of metamodels for computer-aided software engineering tools," *Information and Software Technology*, vol. 52, no. 4, pp. 422 – 435, 2010.

[27] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 Applications, part A: Modeling," *IEEE Transactions on Industrial Informatics*, vol. 6, pp. 136–144, May 2010.

[28] I. Garca-Magario, R. Fuentes-Fernndez, and J. J. Gmez-Sanz, "Guideline for the definition of emf metamodels using an entity-relationship approach," *Information and Software Technology*, vol. 51, no. 8, pp. 1217 – 1230, 2009.

[29] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, pp. 1–38, Springer Berlin Heidelberg, 2008.

[30] S. Patil, V. Dubinin, and V. Vyatkin, "Formal verification of IEC61499 function blocks with abstract state machines and SMV – modelling," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, pp. 313–320, Aug 2015.

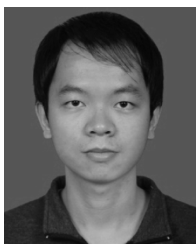
[31] J. Bengtsson and W. Yi, *Timed Automata: Semantics, Algorithms and Tools*, pp. 87–124. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[32] A. Agrawal, "Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck," in *Proc. 18th IEEE Int. Conf. Automated Software Engineering*, pp. 364–368, Oct. 2003.

[33] M. M. Y. Kuo, L. H. Yoong, S. Andalam, and P. S. Roop, "Determining the worst-case reaction time of IEC 61499 function blocks," in *Proc. 8th IEEE Int. Conf. Industrial Informatics*, pp. 1104–1109, July 2010.

[34] L. Lednicki, J. Carlson, and K. Sandström, "Model level worst-case execution time analysis for IEC 61499," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, (New York, NY, USA), pp. 169–178, ACM, 2013.

[35] S. Li, D. Li, F. Li, and N. Zhou, "CPSiCGF: A code generation framework for CPS integration modeling," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1234 – 1244, 2015.



Nan Zhou Nan Zhou received the B.Eng. degree in mechanical engineering and automation from the South China University of Technology, Guangzhou, China, in 2013, where he is currently pursuing the Ph.D. degree with the School of Mechanical and Automotive Engineering. His current research interests include embedded system design theory and methodology, IEC 61499 function blocks, motion control systems, real-time ethernet and formal method for industrial cyber-physical systems.

Di Li Di Li received the B.Eng. and M.Eng. degrees in aircraft engine and power engineering from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1985 and 1988, and the Ph.D. degree in automatic control theory and application from the South China University of Technology, Guangzhou, China, in 1993. She was a Visiting Scholar with The University of Sydney, Sydney, NSW, Australia, and a Senior Research Fellow with Vanderbilt University, Nashville, TN, USA, and in Japan and Singapore. She is currently a Professor with the School of Mechanical and Automotive Engineering, South China University of Technology. She has directed over 50 research projects, including the National High Technology Research and Development Program of China (863 Program) and the National Natural Science Foundation of China. She has authored or co-authored over 180 scientific papers. Her research interests include embedded systems, computer vision, and cyber-physical systems.

Valeriy Vyatkin Valeriy Vyatkin (M'03,SM'04) received Ph.D. degree from the State University of Radio Engineering, Taganrog, Russia, in 1992. He is on joint appointment as Chaired Professor of Dependable Computation and Communication Systems, Lulea University of Technology, Lulea, Sweden, and Professor of Information and Computer Engineering in Automation at Aalto University, Helsinki, Finland. He is also co-director of the international research laboratory of Computer Technologies at ITMO University, St. Petersburg, Russia. Previously, he was a Visiting Scholar at Cambridge University, U.K., and had permanent academic appointments with the University of Auckland, Auckland, New Zealand; Martin Luther University of Halle-Wittenberg, Halle, Germany, as well as in Japan and Russia. His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; and distributed architectures and multi-agent systems applied in various industry sectors, including smart grid, material handling, building management systems, and reconfigurable manufacturing.

Dr. Vyatkin was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012.

Victor Dubinin Victor Dubinin received the Diploma degree in computer engineering, the Ph.D. degree, and Sc.D. degree in computer science from the University of Penza, Penza, Russia, in 1981, 1989, and 2014, respectively. From 1981 to 1989, he was a Researcher, from 1989 to 1995, he was a Senior Lecturer, and from 1995 to 2015, he was an Associate Professor with the University of Penza. Since 2015, he has been a Professor with the Department of Computer Science, University of Penza. He held a Visiting Researcher position with the University of Auckland, Auckland, New Zealand, in 2011, and with the Lulea University of Technology, Lulea, Sweden (2013C2017). His research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.

Dr. Dubinin received DAAD-grants to work as a Guest Scientist at Martin-Luther-University, Halle-Wittenberg, Germany, in 2003, 2006, and 2010.

Chengliang Liu Chengliang Liu was born in Shandong, China, in 1964. He received the B.Eng. degree from the Shandong University of Technology, Shandong, and the M.Eng. and Ph.D. degrees from Southeast University, Nanjing, China, in 1991 and 1998, respectively. He has been invited as a Senior Scholar at the University of Michigan, Ann Arbor, and the University of Wisconsin-Madison, Madison, since 2001. He is currently with the Institute of Mechatronics, Shanghai Jiao Tong University, Shanghai, China. His current interests include mechatronic systems, MEMS/NEMS design, intelligent robot control, web-based remote monitoring techniques, and 3S (GPS, GIS, RS) systems.