MDD: A Unified Model-driven Design Framework for Embedded Control Software

Zhuo Su, Dongyan Wang, Yixiao Yang✉, Zehong Yu, Wanli Chang, Wen Li, Aiguo Cui, Yu Jiang✉ and Jiaguang Sun

***Abstract*—Model-driven methods are widely used in embedded control software development. Current design tools such as Ptolemy-II and Simulink have strong modeling capability but their simulation and code generation functionalities are challenged by the increasing complexity of control requirements. For simulation, emulating the triggering of the actor leads to additional time overhead and speed degradation. For code generation, generating redundant content degrades the code quality. Besides, current tools do not have a unified interface, which makes it difficult to cooperation.**

In this paper, we propose a unified model-driven design framework MDD to facilitate embedded control software development. MDD can support the unification of models built by different modeling tools for high-efficiency simulation and high-quality code generation. And the MDD framework supports the expansion of more modeling tools, and also supports the expansion of more uses, such as unified testing and verification. First, it offers a model intermediate representation (MIR) and several corresponding parsers, which facilitate a unified representation and cooperation for different design tools. Then, based on data flow schedule analysis of the original MIR, intermediate code representation will be generated for optimized code synthesis. Finally, a variety of code translators will synthesize the intermediate code representation into the code of actual use such as code for simulation and code for deployment. For evaluation, we enhance two widely used design tools in industry, Ptolemy-II and Simulink, and apply them on the implementation of several benchmark models and a real-world self-driving control software of our industrial collaborator. Using MDD can help reduce their simulation time by 98.9% and 92.6%, the generated code by 99.7% and 69.9% in the number of lines, and 94.3% and 34.3% in code execution time, respectively.

***Index Terms*—Model-driven design, model simulation, code generation.**

I. Introduction

Model-driven design is widely used in embedded software development, especially in safety-critical applications [1], [2]. Model-based simulation, verification, and code generation can minimize system errors and ensure safety [3]. However, with the rapid development of industrial Internet of Things technology, embedded control software has become more complex, which brings higher requirements for design tools, especially for the simulation and code generation efficiency [4].

Simulink [5] is currently one of the most widely used model-driven design tools in embedded control software development. It relies on the powerful data calculation and processing capabilities of MATLAB to support rich modeling and simulation functionalities. Ptolemy-II [6] is a heterogeneous system design tool developed by the University of Berkeley, which has got much attention in industry and academia [7], [8], [9]. It has rich modeling semantics, supports hierarchical and state machine modeling of embedded systems. Both of them can also automatically generate C code. Because of the powerful modeling capabilities, it is difficult for them to generate efficient code for complex models. For example, the C code generator of Ptolemy-II can only generate code that completely simulates data transmission. In the meanwhile, most of the codes generated by Ptolemy-II are used for data reception and processing. Even for a small model, the C code generator of Ptolemy-II will generate tens of thousands of lines of code, which is difficult to be used in embedded device directly. Figure 1 shows an accumulator model built using Ptolemy-II. Even for such a small model, Ptolemy-II will generate 11371 lines of code. In fact, the logic of the accumulator is only to add the output of the previous time and the input of this time.

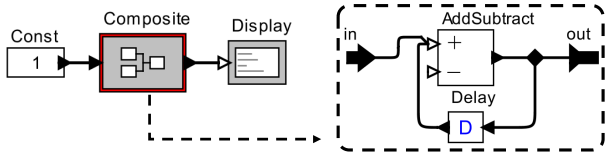

Fig. 1. An example of a simple Ptolemy-II model. It is an accumulator model completed by a Delay actor. The composite actor named Composite is a packaged accumulator.

Not only that, when we were developing our project, we found that we had to simulate the model for a long period of time in order to fully verify the correctness of the model. In addition, we also found that simulation in code takes much less time than simulation in modeling tools. In order to reduce the time cost, simulating directly using code may be an acceleration method. Moreover, we also found that it is much easier to build a model equivalent to Simulink using Ptolemy-II. This is because Ptolemy's model expression ability is more abundant. For example, when building branch logic, Ptolemy-II can be completed with fewer components. Therefore, we hope to reduce the cost of model-driven development through the collaborative development of multiple modeling tools.

Z. Su, Y. Yang, Z. Yu, Y. Jiang, and J. Sun are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mail: suzcpp@gmail.com).

D. Wang is with the Information Technology Center, Renmin University of China, Beijing 100872, China (e-mail: wdy@ruc.edu.cn).

W. Chang is with the Department of Computer Science, University of York, York, YO10 5DD United Kingdom (e-mail: wanli.chang@york.ac.uk).

W. Li is with the HUAWEI Technologies, Co. LTD., Hangzhou 310000, China (e-mail: coco.liwen@huawei.com).

A. Cui is with the HUAWEI Technologies, Co. LTD., Shanghai 200120, China (e-mail: ag.cui@huawei.com).

Yixiao Yang and Yu Jiang are the corresponding authors.

More specifically, for an efficient model-driven design of embedded control software, there are three challenges.

Challenge 1: How to leverage existing models of different tools and support the collaborative development of multiple design tools . At present, the only standard interface that can be used to dock multiple design tools is Functional Mock-up Interface (FMI) [10]. It packages different models into a specific code format named Functional Mock-up Units (FMU) [11], [12] for co-simulation. However, FMU only contains the code generated by the external tools, and the code is difficult to be used in subsequent processes. Thus, there is still no interface for model reuse or data sharing between different design tools. Many researchers also try to use model converters to convert models between different design tools [13], [14], [15]. However, there are many unique characteristics that are difficult to convert from one to another. Coupling the semantics of different models into a unified high-level semantics is also a way to combine multiple modeling tools. Both ModHel'X [16] and Ptolemy-II are tools that combine different model semantics to support heterogeneous modeling and simulation. They all use Token to transfer data among actors and model of computation (MoC). The main difference between them is that ModHel'X actors are executed by observation, while Ptolemy-II actors are executed by trigger. However, their coupling method makes scheduling analysis difficult, so that no code generator can realize the code generation of such a complex heterogeneous system.

Challenge 2: How to accelerate the simulation speed of complex models. Model simulation is to execute each actor one by one according to the predefined semantics [17], [18], [19]. It tends to be iterated many times to simulate the real-time execution of the software. Two main factors affect the efficiency of the simulation. On the one hand, in modern tools, the types of ports are polymorphic, thus, the port type needs to be determined during the execution process and the determination of the port type causes extra time overhead [20]. For example, AddSubtract actors can perform different operations according to different input data types. On the other hand, model scheduling is a dynamic process. Actor scheduling for complex models is usually uncertain. Therefore, dynamic scheduling will cause additional time overhead.

Challenge 3: How to generate high-efficient code for those target devices with limited resources. Due to the complexity of model semantics, those tools usually generate highly redundant code with low execution efficiency [21], [22], [23]. For example, in Ptolemy-II, the Branch-Actor will pass data to different ports according to conditions, which will affect the execution logic of subsequent actors. When the branches are nested and crossed, the scheduling relationship of the model is difficult to sort out. The traditional code generation method will add the code of the branch condition to the code of the actor when an actor is affected by the branch. However, while the branch combination is complicated, it will lead to an explosion in the number of execution conditions of an actor. Similar problems happen in Simulink. How to reduce branch explosion is the key to generate high-efficient code.

In this paper, we propose a unified model-driven design framework MDD for the three challenges. The framework is mainly divided into three layers. 1) The first is the model parse layer, which is mainly used to interface with other design tools. The model parser converts different types of models into the MIR, and different tools need different parsers. 2) The second is the schedule convert layer, which takes the MIR as input, analyzes the control flow logic relationship of the entire model based on the data flow analysis of the actors, and outputs the intermediate code representation. 3) The third is the code translate layer, which is responsible for optimized code generation of different targets, such as accelerated simulation, high-efficient execution, or comprehensive testing, etc.

For evaluation, we use MDD to enhance two widely used design tools, Ptolemy-II and Simulink, and apply them to the implementation of build-in benchmark models and real-world self-driving control software of our industrial collaborator. We conduct quantitative analysis from two aspects: speed of simulation and quality of code generation. Using MDD can help reduce their simulation time by 98.9% and 92.6%, the generated code by 99.7% and 69.9% in the number of lines, and 94.3% and 34.3% in code execution time, respectively. In addition, we also demonstrate how to use MDD to conduct collaborative development of different design tools.

II. Related Work

A. Model-driven Design

Model-driven development has been widely used in the design and implementation of embedded control software, which mainly consists of three parts: model construction, model simulation and code generation [24], [25], [26], [27]. There are many widely used design tools, such as Simulink, SCADE, Tsmart and Polychrony [5], [28], [29], [30], [31]. For example, Lorenzo used Simulink to implement an electric vehicle brake hybrid strategy system [32]. They used the FMI provided by Simulink and Simcenter AMESIM to complete the simulation of the brake model and the vehicle control model. Bagheri established an adaptive rail-based control system, with the hierarchical modeling capability of Ptolemy-II [33]. In addition, Zhang has specially implemented a design tool Tsmart that supports simulation and code generation in order to realize the design of the multi-functional vehicle bus controller chip [31]. Most of them use data flow and state machine to express the control logic. The data flow model is composed of actors, junctions and connections between actor ports. Actors are used to receiving data, process data, and send data. The junction can transmit a piece of data to multiple targets at the same time, and also can indicate the merging of data at the end of the data branch. The connection between the ports is used to transfer data. The state machine model is composed of different states and transitions between states. The state machine model is often packaged as a composite actor with ports and used in the data flow model.

There are also many model-driven development works based on UML. These works are usually devoted to abstraction and description of system functions. For example, the DaVinci tool set [34], which is most widely used in the industry in the design of automotive embedded systems, uses graphical UML diagrams for the architecture design of the entire vehicle

system, and also supports the generation of architecture-level code for the vehicle control system. However, every functional unit in the system is not implemented in code. For another example, Enrici et al. tried to use UML to describe parallel communication systems to develop distributed systems [35]. There are many similar UML-based model-driven development work [36], [37], [38], but few works on modeling and code generation for the specific behavior of the system.

B. Model Simulation

Most design tools such as Simulink and Ptolemy-II mainly adopt the event triggering semantics for model simulation [6], [5]. For the data flow model, the model must be topologically sorted according to the data transfer relationship. When an actor is executed, it will read the data of its input port, process the data according to the function of the actor, and transfer the result to the output port. The output port will pass the data to its subsequent actors. When the actors of the same topological sorting layer are all executed, the actors of the next topological layer that can receive data are executed. The execution process proceeds backward layer by layer until the data can no longer be passed back. At this time, an iterative execution of the model is completed. The simulation of the entire model may be completed by multiple iterations.

For the state machine model, at first, the model is set to the default initial state, and every time the state machine is triggered, it is evaluated whether the conditions on the transition from the current state to other states are satisfied. The state jump is performed if it is satisfied. Of course, there may be statements that need to be executed in the state itself and the transition, such as assigning a value to a variable or initializing a clock. When the state machine is used as a composite actor in the data flow model, it will be regarded as an ordinary actor to trigger the internal logic of the state machine once. The reading and assignment of input and output port data are completed by the execution statement on the state and transition.

C. Code generation

Code generation is used to convert the model into code that can be used for deployment. Since most of the time the model will be a dynamic system, the execution order of the actors in the model may be changeable. Therefore, before generating code, it is necessary to perform schedule analysis on the model. The difficulty of schedule analysis is mainly reflected in the influence of the complex branch structure on the execution logic of the model. In recent years, there are only a few related academic works. Su tried to solve the problem of branch explosion during code generation with branched data flow model of Ptolemy-II [39]. First, it uses the branch marking method for the Ptolemy-II model to mark the branch information of the possible execution logic of each actor. Second, by adopting the method of simplifying the branch mark from back to front, the actors that need to be merged outside the branch can be found. Third, a specific code generation template is used for each actor to generate code from front to back according to the result of the branch mark. It can solve the code generation of models with simple branching logic, but it cannot handle the models with complex branches like branch crossing.

The C code generator of Ptolemy-II [22] first performs topological sorting of the actors to determine the main execution order and then generates codes according to the logic of the simulation. Although the code generated in this way can deal with various complex schedule logic. The code contains complete data transfer and actor execution code, and even a lot of basic library codes based on C language are generated to support data transfer, such as queues and stacks. It leads to a lot of redundant code and poor efficiency of execution, which is difficult to deploy to hardware.

Simulink Coder is a relatively robust code generator, which can support almost any model built by Simulink. However, Simulink adopts a way of avoiding complex branch semantics for modeling, which makes the work of code generation easier, but it becomes very cumbersome to express branch logic in Simulink. The branch logic can only be expressed in three ways: using the If-Else subsystem, embedding code, and using a state machine. Simulink Coder has a certain code optimization function, which can support simple code optimization such as local variable reuse and expression folding. However, the code generated by Simulink Coder contains a lot of code for environment, data and data type definition.

Furthermore, both Ptolemy-II's C code generator and Simulink Coder support expand and pave the hierarchical model as a single-layer model for code generation during schedule analysis. The code generated in this way also loses the original model hierarchical information, and it is difficult to reuse and debug part of the code.

III. MDD Framework Design

We propose MDD, a unified model-driven design framework for embedded control software, which aims to provide collaboration capabilities for multiple design tools, and further improve the efficiency of simulation and the quality of code generation. MDD follows three steps: model parse, schedule convert and code translate.

First of all, the model parse layer will convert different types of models into a unified model intermediate representation. It is able to parse various types of models, which built by existing tools such as Simulink and Ptolemy-II, or other design tools that conform to the basic data flow and state flow simulation semantics introduced in Section II-B. Due to the differences in the details of the model semantics and actor functions supported by those tools, there must be a corresponding model parser for each design tool. After that, the schedule convert layer performs schedule analysis on the MIR, focusing on the data flow logic of the actors. It will generate the intermediate code representation to retain the actor information of the original model, which can facilitate and provide an important reference basis for further processing. Finally, the intermediate code representation is synthesized into codes for different purposes, such as simulation and execution. Different code translators will add necessary function codes in the translation process according to different requirements.

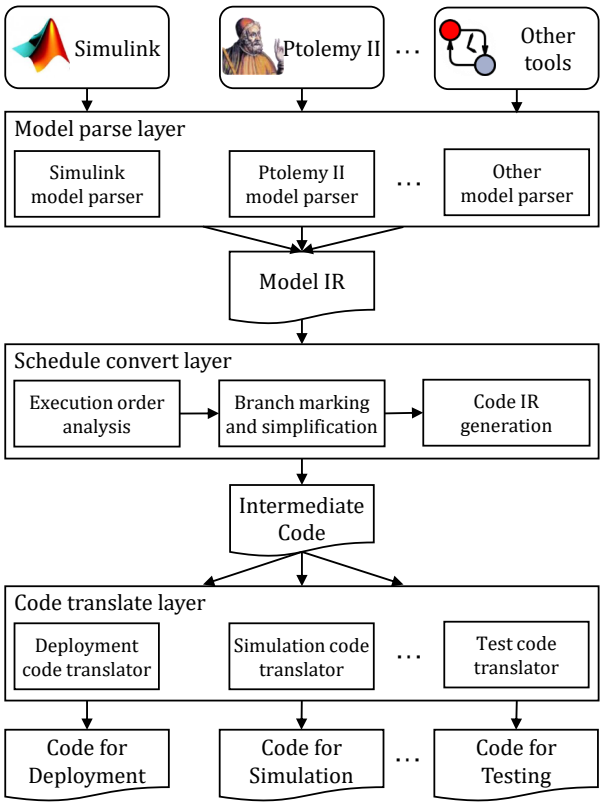

Fig. 2. MDD framework consists of three layers: Model parse layer (Parse the model built by different tools to MIR). Schedule convert layer (Convert the MIR into intermediate code representation). Code translate layer (Translate the intermediate code representation into code for purposes of actual use)

A. Model parse layer

In order to support the collaboration of multiple development tools, the model analysis layer is designed as an interface for various design tools. The model parser takes the model file constructed by its corresponding design tool as input and takes the unified model intermediate representation (MIR) as output. Since the model semantics and actor functions supported by different model-driven design tools are not completely consistent, the MIR is designed into a model description language with a stronger expressive ability to be compatible with them. For example, Ptolemy-II's state machine semantics and Simulink's state machine semantics have certain differences. The state in the Ptolemy-II state machine supports nested data flow composite actors, while Simulink's state machine model semantics supports three kinds of special events: entering event, during event, and exiting event. To be compatible with them, it is necessary to define a new state machine semantics that not only supports the state-embedded data flow composite actor but also supports the three special events. Even if different design tools implement support for the same type of actors, they may be different in the details of the actor's interface or properties. For example, the Addsubtract actor in Simulink supports different types of rounding operations on floating-point data, such as floor and ceiling. But the Addsubtract actor in Ptolemy-II does not support rounding of floating-point data. Therefore, the Addsubtract actors in the MIR should be designed to support rounding operations in different ways. For actors of the same type with conflicting interfaces or attributes in different design tools, design can be done according to different actors. The model intermediate representation adapted to Simulink, Ptolemy and Uppaal is defined as follows.

Definition 1 (model intermediate representation): The model intermediate representation MIR is defined as a tuple $\langle G, A, J, R, P\rangle$. Among them, G is the set of global variables of the model, A is the set of top-level actors in the model, J is the set of top-level data junctions in the model, R is the set of connections between actors' ports and junctions in the model, and P represents the configuration parameters of the model. In particular, the actor a_i contained in A can be a single computing actor e, a composite actor c, or a state machine actor m.

Definition 2 (computing actor): The computing actor e is defined as a tuple $\langle PI, PO, Attr, IS, exe\rangle$. Among them, PI is the set of input ports of the actor, PO is the set of output ports of the actor, $Attr$ is the set of attributes of the actor, IS is the set of internal states of the actor, and exe is the execution logic of the actor. The function of exe is to obtain the data of the input ports PI, assign the output ports PO and update the internal state IS according to the attribute $Attr$ of the actor and the internal state IS.

Definition 3 (composite actor): The composite actor c is defined as a tuple $\langle PI, PO, A, R\rangle$. The representations of A and R in c and A and R in MIR are the same. The representations of PI and PO in c and PI and PO in the calculation actor e are consistent.

Definition 4 (state machine actor): The state machine actor m is defined as a tuple $\langle PI, PO, S, T, l, l_0\rangle$. The representations of PI and PO in m and PI and PO in the calculation actor e are consistent. S represents the state set in the state machine. T represents the set of transitions in the state machine. l represents the current state of the state machine. l_0 represents the initial state of the state machine.

In order to support more design tools, the state s_i is designed to support richer semantics.The state s_i is defined as a tuple $\langle C, M, en, du, ex\rangle$. Among them, C represents the set of composite actors embedded in the state, and M represents the set of state machine actors embedded in the state. Embedded composite actors and state machines are generally used to enrich the expressive capabilities of state machines, and their input and output ports are consistent with the outer state machine actors. en is the logic executed when entering this state from other states. du is the logic executed when the state machine maintains the state without a state transition after a trigger. ex is the logic executed when the state is about to change to other states. Among them, en, du and ex all allow assigning values to global variables, performing other calculations, or calling other functions. Each t_i in the state machine transition set T is defined as a tuple $\langle g, l_{src}, l_{dst}, exe\rangle$, where g represents the conditions under which the transition occurs. l_{src} represents the source state of the transition, and l_{dst} represents the destination state of the transition. exe represents the execution logic in the transition process. For each t_i, its meaning is that when the current state of the state machine is l_{src}, it is judged whether the transition

condition g is satisfied and if it is satisfied, exe is executed, and the state machine's current state is changed to l_{dst}.

MIR supports the coupling of new semantics in two ways. One way is to add attributes to the MIR element to be compatible with the new semantics. Since most tools supporting embedded system development use data flow or state machine semantics, MIR can directly compatible with most tool models by this way [6], [5], [28], [29], [40], [41], [31]. And the other way is to extend the current MIR semantics by adding a new MIR element when the new semantics are quite different from Simulink, Ptolemy, etc. Just like the state machine semantics that is completely inconsistent with the data flow semantics, we can treat the state machine as a computing unit that calculates output data based on input data and insert it into the data flow model in the form of an actor. Other models with different semantics will be packaged as a composite actor and integrated into the data flow model.

Based on the above definitions, MDD reads the elements in the original model and store the corresponding elements in the MIR according to predefined mapping rules. Since the MIR we designed is semantically compatible with Simulink, Ptolemy, Uppaal and other design tools, each element in the source model can be uniquely mapped to the element in the MIR. For example, the "Block" in Simulink will be directly mapped to the "Actor" in the MIR, and the "Property" in the "Block" corresponds to the "Parameter" of the "Actor" in the MIR. For another example, the "Location" in the Uppaal state machine model will be mapped to the "State" in the MIR. To reduce the coupling of the entire framework and improve the scalability of MDD, MIR will be stored in a separate file and then handed over to the schedule converter for schedule analysis.

Fig. 3. An example of parsing from Ptolemy-II model to MIR. The XML file on the left is the Ptolemy-II model and it corresponds to the composite actor in Figure 1. The XML file on the right is the MIR. The gray arrow indicates the mapping relationship between elements.

Figure 3 shows an example of parsing from Ptolemy-II model to MIR. In order to facilitate the demonstration, we omitted redundant elements in the Ptolemy-II model, such as display coordinates and default parameter of "Entity". In the Ptolemy-II model, the connection among ports is expressed by "Relation" and "Link". When converting it to "Relation" in MIR, connectivity analysis is needed to find all input and output ports connected to this "Relation".

B. Schedule convert layer

The schedule convert layer performs data flow based schedule analysis of the MIR to obtain the intermediate code representation, which facilitates the subsequent processes, such as generating high-quality code deployed on embedded devices, high-speed simulation, and automatic testing.

The main method used for schedule conversion is to mark each actor with branch information, and then to simplify branch marks of each actor as much as possible, so as to obtain the possible branch path information of each actor. The specific schedule conversion algorithm consists of three parts. The first part is to analyze the schedule of MIR in accordance with the execution sequence of the simulation logic. The second part is to mark the branch information for each actor and junction in the model and simplify the branch information in the marking process. The third part is to generate intermediate code representation based on control flow according to the branch mark information of each actor and junction in the model.

1) Execution order analysis:

We treat MIR as a directed acyclic graph and determine the execution order of each actor and junction through topological sorting of the graph. We can ignore the junctions which transmit data to multiple targets, because these junctions can not affect the data source of subsequent actors. It should be noted here that for the data flow model with a loop, we have to break the loop by splitting the register-type actors in the loop into two parts, the storage and the fetch. For the data flow model used for code generation, if there is a data loop, there must be a delay, queue, data pool and other registered actors in the loop. Otherwise, once the loop is executed, the model will enter an endless loop. Figure 4 shows an example of breaking the loop. The model on the left contains a Delay actor, which saves the input data, and then outputs the saved data when it is executed next time. The model on the right is equivalent to the model on the left. The Delay actor is decomposed into two parts, a data-writing actor and a data-reading actor. The delay value is stored using an intermediate variable DelayValue.

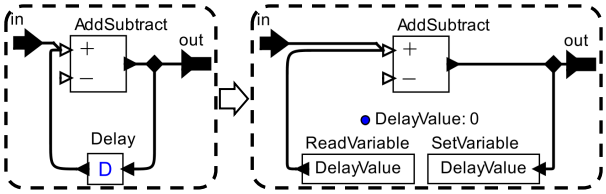

Fig. 4. Break loop Example. The model on the left has a loop. The model on the right is completely equivalent to the model on the left, and does not have a loop.

For topological sorting, we need to find the actors that do not have any input data in the model. These actors are used as the first layer of sorting. Then eliminate the actors of the first layer and all the data connections connected to them, and continue to look for actors and junctions without any input data. Take these newly found actors and junctions as the second layer of sorting. The following layers can be deduced by analogy. If there are no actors without any input data, but there are still actors in the remaining part of the model that have not been eliminated, it means that there is at least one loop in the model. At this time, we need to split the register-type actors in the remaining part of the model into two parts as shown in Figure 4. Then continue topological sorting on the remaining part of the model. We only layer the model, and there is no need to calculate the execution order of the actors in the layer because the execution of the actors in the same layer will not affect each other.

2) Branch marking:

Before introducing the branch marking algorithm, several concepts need to be clarified: the branch ($Branch$) represents a single branch of a branch actor or a junction. The $Branch$ is consists of the branch actor and the branch id (the branch number of the branch actor). The unique representation of $Branch$ is recorded as Id_{Actor}, Id represents the branch id, and $Actor$ represents the branch actor. The branch path ($BranchPath$) represents the different branch execution situations brought about by multiple branch actors, denoted as $\{Branch_1, Branch_2, Branch_3, ...\}$. The branch path table also called the branch mark information of an actor or a junction, represents the collection of all possible branch paths and data sources on the branch path of an actor or a junction, denoted as $\{BranchPath_1 : Src_1, BranchPath_2 : Src_2, BranchPath_3 : Srcr_3, ...\}$. Among them, Src can be an actor or a junction.

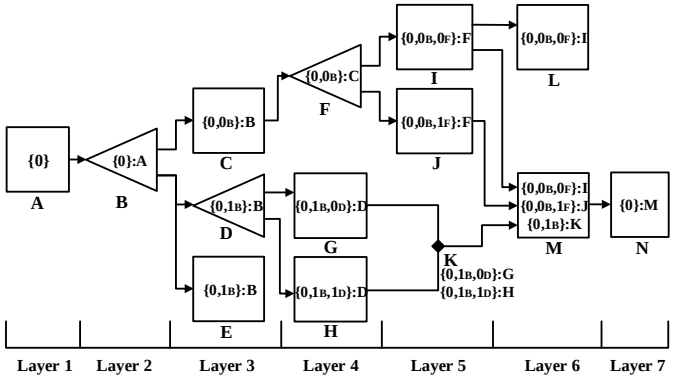

Fig. 5. The result of branch marking. This example model is divided into seven layers by topologically sorting. There are three branch actors B, D and F. And the branch paths of these three actors are combined by junction K and actor M.

The branch marking method is shown in the Algorithm 1, which traverses the entire model layer by layer from front to back and calculates branch information separately for each actor and junction in each layer. Figure 5 is shown as an example of the branch marking algorithm. The $element$ in Algorithm 1 represent an actor or a junction. Line 3 of the Algorithm 1 obtains all the predecessor elements of the current element. If the current element does not have any predecessor elements, the branch of the current element is directly marked as a branch path with empty data, as shown in lines 4-6. If there is a predecessor element, a branch path table is constructed. The branch path table is a map structure. The key stores the branch path, and the value stores the data source elements brought by the key branch path. Next, lines 8-17 traverse all predecessor elements of the current element to obtain their branch path information. Among them, if the predecessor element is not a branch actor, the branch path information of the predecessor element is directly inherited, and the predecessor element is used as the data source of the current branch path of the current element, as shown in lines 15-17. If the predecessor element is a branch actor, the branch information of the predecessor element shall be connected on the basis of the branch path of the predecessor element as a new branch path, as shown in lines 10-14. After all, the branch paths of the current element are obtained, all the inherited branch paths of this element are simplified and then assigned to the branch information of the current element, as shown in lines 18-19.

Algorithm 1: Branch marking method

Input: mls: Model Layers, the topologically sorted data flow model
Output: bis: Branch mark information of all actors and junctions in the model

```
1  for layer in mls do
2      for element in layer do
3          predecessors = getPredecessors(element)
4          if predecessors == ∅ then
5              bis[element] = getEmptyDataBranchPath()
6              continue
7          bps = ∅ // bps: branch paths of element
8          for p in predecessors do
9              bpsOfPredecessor = bis[p]
10             if isBranchActor(p) then
11                 for branchPath in bpsOfPredecessor do
12                     for b in getBranches(p) do
13                         newPath = connect(branchPath, b)
                             // Insert b at the end of branchPath
14                         bps[newPath].addDataSrc(p)
15             else
16                 for branchPath in bpsOfPredecessor do
17                     bps[branchPath].addDataSrc(p)
18         bps = simplifyBranchPath(bps) // Call Algorithm 2
19         bis[element] = bps
20 return bis
```

3) Branch mark simplification:

Branch simplification is used to reduce the condition statement of each element after branch marking. The Algorithm 2 describes in detail the branch simplification algorithm for a single element, corresponding to the simplify BranchPath function on line 18 in the Algorithm 1. The overall idea is to traverse the combination of all branch paths of the current element, find the branch paths that can be merged, and replace them with the merged branch path.

In order to avoid the traversal of meaningless branch path combinations, first, all branch paths need to be sorted from short to long according to the length of the branch path, as shown in the first line of the Algorithm 2. For example,

Algorithm 2: Branch information simplification for a single element(actor/junction)

```
   Input: bps: branch paths of the element
   Output: sbps: simplified branch paths of the element
1  sbps = sortByPathLength(bps)
2  curPos = len(sbps) - 1
3  while curPos > 0 do
4      branchPath = sbps[curPos].key
5      dataSrcs = sbps[curPos].value
6      subbp = branchPath[0:-1]
         // get the sequence except the last Branch of branchPath
7      lastb = branchPath[-1] // get the last Branch of branchPath
8      findPos = curPos − 1
9      bpsSameSrc = ∅
10     bpsSameSrc.append(branchPath)
11     while findPos ≥ 0 do
12         bpFind = sbps[findPos].key
             // the branch path at the current find position
13         dataSrcsFind = sbps[findPos].value
             // the data sources at the current find position
14         subpFind = branchPath[0:-1]
15         lastbFind = branchPath[-1]
16         if len(branchPath)= len(bpFind) and
17           dataSrcsFind == dataSrcs and
18           subbp == subpFind and
19           isSameBranchActor(lastb, lastbFind) then
20             bpsSameSrc.append(bpFind)
21         findPos = findPos - 1
22     if len(bpsSameSrc) > 1 then
23         if canBranchPathsMerge(bpsSameSrc) then
24             sbps.insert(subbp, dataSrcs)
25             sbps.remove(bpsSameSrc)
26             curPos = curPos - len(bpsSameSrc) + 1
27             continue
28     curPos = curPos - 1
29 return sbps
```

$\{0, 0_B, 0_F\}$, $\{0, 1_B\}$, $\{0, 0_B, 1_F\}$ in Figure 5 will be sorted as $\{0, 1_B\}$, $\{0, 0_B, 0_F\}$, $\{0, 0_B, 1_F\}$. Then we match from back to front to find possible merged branch paths. The matching conditions here are: 1) Have the same length of branch path. 2) Have the same data source. 3) Except for the last branch in the branch path, the previous branches are the same, such as $\{0, 0_B, 0_F\}$ and $\{0, 0_B, 1_F\}$. 4) The branch actors of the last branch in the branch path are the same. The pseudo-code of branch path matching is shown in lines 4-21. Lines 9, 10, and 20 are used to store all the branch paths that are matched. Lines 22-27 try to merge the matched branch paths. Merging means that the last branch of all matched branch paths is complementary, such as 0_F of $\{0, 0_B, 0_F\}$ and 1_F of $\{0, 0_B, 1_F\}$ are complementary in the case of two branches, and the same is true for multiple branches. If these branch paths can be merged, the merged branch path will be inserted into the set of current branch paths to continue to participate in the matching. Because the set of branch paths in some cases can be merged multiple times, for example, $\{0, 1_B\}$, $\{0, 0_B, 0_F\}$, $\{0, 0_B, 1_F\}$ finally can Merge into $\{0\}$. Then continue to try new matches until all the branch path combinations are traversed.

The following is an explanation of the entire branch marking and simplification based on the model in Figure 5. First, mark the first layer. Since our default model is in an empty branch at the beginning, we mark the actor A as $\{\{0\}\}$, where $\{\{0\}\}$ represents the first branch and does not With data source. The branch marking is carried out layer by layer, and then actor B in the second layer is marked. Because the branch will affect all subsequent actors, the branch mark must be passed to the subsequent actors, so the actor B must also be marked $\{\{0\}\}$, but at this time B receives the output data from A, so it is recorded under this branch to accept the data of A, which is recorded as $\{\{0\} : A\}$. Next, mark the third layer. Since the actor B has two branches, actor C is affected by the first branch of actor B, and actor C is still under the control of the branch where the original actor A is located, so the actor C should be marked as $\{\{0, 0_B\} : B\}$, which means that it is under the control of the first $\{0\}$ branch and is also under the control of the first branch of B, and receives Data from actor B. In the same way, actor D and E are marked as $\{\{0, 1_B\} : B\}$. The actor D in the third layer and the actor F in the 4th layer will continue to nest branches. The actor G and H in the 4th layer will be affected by the branch of the actor D, and their branch paths will increase the branch of the actor D. Compared with the branch mark of actor C, the actor I and J in the 5th layer will be added an extra branch by the actor F. For example, the branch of actor I is marked as $\{\{0, 0_B, 0_F\} : F\}$, and the branch of actor J is marked as $\{\{0, 0_B, 1_F\} : F\}$. The junction K in the 5th layer will be marked with two branch paths, and the data sources of the two branch paths are different, so they cannot be merged and simplified. The actor L in the 6th layer directly inherits the branch path of actor I. The branch mark of actor M in the 6th layer is somewhat different from the previous actors and junction. The actor M aggregates a total of four branch paths from the actor I, J, and junction K. However, since the original $\{0, 1_B, 0_D\} : K$ and $\{0, 1_B, 1_D\} : K$ of the actor M are complementary on the last branch, they can be merged into $\{0, 1_B\} : K$, so the branch mark of the actor M can finally be simplified into three branch paths. Finally, actor N gathers all branches. According to the Algorithm 2, $\{0, 0_B, 0_F\}$ and $\{0, 0_B, 1_F\}$ can be merged into $\{0, 0_B\}$. $\{0, 0_B\}$ and $\{0, 1_B\}$ can be merged into $\{0\}$ again. And the actor N only receives data from the actor M, so the branch of the actor N is marked as $\{\{0\} : M\}$.

4) Intermediate code representation generation:

In the previous section, we have marked all the actors and junctions in the model. These branch marks are the control flow information of these actors and junctions. With this information, the most direct way to generate code is to generate code layer by layer and actor by actor. If the actor is affected by the branch, the conditional constraint is added before the execution code of the actor. Obviously, it will generate a lot of redundant conditional judgment statements. In order to reduce redundant conditional judgment codes and generate codes with clearer control logic, we construct a code generation table (CGT) for the intermediate code representation.

A complete CGT contains the following elements, as shown in Table I. The construction method of the CGT is described in the Algorithm 3. First, we initialize an empty code generation table CGT, as shown in lines 1-2. Then add CGT elements to the code generation table layer by layer and actor by actor according to the branch information of the actor. It is very important here that each layer will use the CGT calculated by

TABLE I
DESCRIPTION OF ELEMENTS IN CGT

element	attribute	description
$ActorExe$	Actor, DataSrc	Indicates that the $Actor$ receives the data of DataSrc and executes the logic of the $Actor$
$JuncData$	Junction, DataSrc	Indicates that the $Junction$ receives and stores the data of DataSrc
$Branch$	Set of branch path	Indicates the entry condition of the branch path
$Inserter$	Set of branch path	Indicates the insertion position of the subsequent actor that matches this branch paths

Algorithm 3: Intermediate code representation generation method

Input: mls: Model Layers, the topologically sorted data flow model, bis: Branch mark information of all actors in the model
Output: CGT: Code generation table

```
1  CGT = ∅
2  CGT.insertInserter(0, {{0}})
3  for layer in mls do
4      for ele in layer do
5          bpsInfo = bis[ele]
             // ele: actor or junction in each layer
             // bpsInfo: branch mark information of an ele
6          bps = bpsInfo.keys // bps: branch paths of an ele
7          if not isBranchPathsConflict(bps) then
8              for bp in bps do
9                  bpSub = bp
                     // bpSub: subsequence of branch path
10                 while not CGT.findInserter(bpSub) do
11                     bpSub = bpSub[0:-1]
12                 CGT.removeInserterContain(bpSub)
13                 CGT.insertGapBranchAndInserter(bp, bpSub)
14                 if isJunction(ele) then
15                     CGT.insertJunction(bp, ele, bpsInfo[bps])
16                 else
17                     CGT.insertActor(bp, ele, bpsInfo[bps])
18         else
19             CGT.removeInserterContain({0})
20             CGT.insertComBranchPathsAndInserter({0}, bps)
21             if isJunction(ele) then
22                 CGT.insertJunction(bp, ele, bpsInfo[bps])
23             else
24                 CGT.insertActor(bp, ele, bpsInfo[bps])
25     CGT.update()
26 CGT.removeEmptyBranch()
27 CGT. removeAllInSerter()
28 return CGT
```

the previous layer when the algorithm is iteratively executed. In other words, the process of finding the $Inserter$, inserting the $Inserter$, and inserting the $Branch$ of the actor will not affect other actors and junctions of the current layer. After each layer is calculated, the CGT.update() on line 19 of the algorithm sorts out the various CGT elements inserted in this layer, so that they can take effect in the calculation of the next layer. Lines 5-24 deal with the processing code for each actor. First, we obtain the branch path set of the actor or junction, as shown in lines 5-6. Then determine how to modify the code generation table according to the branch path of the actor or junction. If there are no conflicts in all branch paths of the actor or junction, that is, the conditions of these branch paths will not be satisfied at the same time, these branch paths are processed separately, as shown in lines 8-17.

For each branch path, if the $Inserter$ with the branch path can be found directly in the code generation table, the execution code of the actor or data storage code of junction is directly inserted at the $Inserter$ location. If not, the last branch of the branch path is removed as subBranchPath $bpSub$ to find $Inserter$, if it still can't find a branch after removing a branch, remove the last branch as the new subBranchPath until it finds an existing $Inserter$. This iteration is limited because at least the $Inserter$ with $\{\{0\}\}$ will be found at the end. In the iterative process, if an $Inserter$ that satisfies the conditions is found, remove all the $Inserter$s in the branch where the $Inserter$ is located in the code generation table. Then create a multi-layer branch structure at the $Inserter$ position, and add the corresponding $Inserter$ to each branch structure. For example, if the branch path searched for by iteration is $\{0, 0_B, 0_F\}$, and finally an $Inserter$ with $\{\{0\}\}$ is found, then the $Branch$es $\{0, 0_B\}$, $\{0, 0_B, 0_F\}$, $\{0, 0_B, 1_F\}$, $\{0, 1_B\}$, $\{0, 1_B, 0_F\}$, $\{0, 1_B, 1_F\}$ and the corresponding $Inserter$ are inserted in sequence at the position of the $Inserter$. Then the $ActorExe$ of the current actor or $JuncData$ of the current junction will be inserted into the $Inserter$ position corresponding to $\{0, 0_B, 0_F\}$. In view of the conflict in the branch paths of the actors, we directly insert the branches of all combinations of these branch paths and the corresponding $Inserter$s at the $Inserter$ position with $\{\{0\}\}$, as shown in lines 19-24. At the end of the algorithm, remove those $Branch$ elements whose branch content is empty, and remove all $Inserter$s, because they do not generate actual code.

This method is suitable for complex branch crossing situations. For example, the branch path set of an actor is $\{\{0, 1_X\}, \{0, 0_Y\}\}$ (where the branch 1 of the actor X and the branch 0 of the Y actor may be executed at the same time). The three $Branch$es $\{\{0, 1_X\}\&\&\{0, 0_Y\}\}$, $\{\{0, 1_X\}\}$, $\{\{0, 0_Y\}\}$ and the corresponding $Inserter$ will be created. Actors with branch path information of $\{\{0, 1_X\}, \{0, 0_Y\}\}$ or $\{\{0, 1_X\}\}$ or $\{\{0, 0_Y\}\}$ will be inserted into the corresponding position.

```
1  ActorExe: A()                          //Created in layer 1
2  ActorExe: B(A)                         //Created in layer 2
3  Branch ({{0, 0B}})                     //Created in layer 3
4    ActorExe: C(B)                       //Created in layer 3
5    ActorExe: F(C)                       //Created in layer 4
6    Branch ({{0, 0B, 0F}})               //Created in layer 5
7      ActorExe: I(F)                     //Created in layer 5
8      ActorExe: L(I)                     //Created in layer 6
9      ActorExe: M(I)                     //Created in layer 6
10   Branch ({{0, 0B, 1F}})               //Created in layer 5
11     ActorExe: J(F)                     //Created in layer 5
12     ActorExe: M(J)                     //Created in layer 6
13 Branch ({{0, 1B}})                     //Created in layer 3
14   ActorExe: D(B)                       //Created in layer 3
15   ActorExe: E(B)                       //Created in layer 3
16   Branch ({{0, 1B, 0D}})               //Created in layer 4
17     ActorExe: G(D)                     //Created in layer 4
18     JuncData:K = G.out                 //Created in layer 5
19   Branch ({{0, 1B, 1D}})               //Created in layer 4
20     ActorExe: H(D)                     //Created in layer 4
21     JuncData:K = H.out                 //Created in layer 5
22   ActorExe: M(K)                       //Created in layer 6
23 ActorExe: N(M)                         //Created in layer 7
```

Listing 1. CGT generated by MDD (Corresponding to the sample model in Figure 5)

The proof of conversion method from data flow to control flow can be obtained by induction. According to Algorithm 3, the initial CGT is correct. Every time the process of inserting $ActorExe$ and $Branch$ into the CGT is correct (Algorithm 1 ensures that the possible branch path of each actor is correct, and Algorithm 3 ensures that the trigger path of each actor in the code is consistent with its branch path.). Therefore, the conversion and code generation can be proved to be correct. Based on the above algorithm and the calculated branch marks, we can construct a complete CGT for model in Figure 5, as shown in Listing 1. Due to space and typesetting restrictions, we can only give part of steps of constructing CGT. Listing 1 shows the final generated CGT, and the comment indicates when the CGT elements were created.

Then, the intermediate code representation that retains model information will be generated and stored in XML format, based on CGT, the accompanied templates and some information in the original MIR. The intermediate code representation contains not only the control flow information of the model, but also the input and output ports of the actor, the data relation of the ports, and the mapping information of the intermediate code representation to the original model actor. As an actor execution unit, $ActorExe$ in CGT will contain the path of its corresponding actor in the original model, the input and output port information of the corresponding actor, and the actor's own execution logic code. The execution logic code of $ActorExe$ is very concise, and it is generated from our carefully designed template for each different Actor. For example, the execution logic code of the AddSubtract actor in intermediate code representation is similar to $Out = Add1 + Add2 - Sub1$. An example of the intermediate code representation corresponding to the accumulator model in Figure 1 is shown in Figure 6.

Note that the execution logic of each actor is added in the process of generating intermediate code representation, because in the subsequent various code translators, the execution logic of each actor will not change, so there is no need to generate execution logic code for each actor in the subsequent process. And the generation of actor execution logic code only needs to be generated according to the actor's code template. The judgment conditions and the branch type (like If, Else or Switch) of the actor need to be added to $Branch$. For each composite actor in the model, it will be completely packaged into a function. The function will contain the input and output ports of the composite actor and the entire schedule logic composed of actor execution units and branches. In addition, other necessary model elements are also generated into the intermediate code representation, such as data type definitions, global variable definitions, function declarations and other configuration parameters of the model.

C. Code translate layer

The code translate layer converts the intermediate code representation into codes for various purposes of actual use, such as C language codes for deployment on embedded devices, and codes for efficient simulation. These codes for different purposes are basically the same in execution logic but with different facility functions. For example, the C language code of embedded control system generally realizes the three operations of reading port data, processing data and writing port data. In the code used for simulation, the results of model simulation will be given quickly according to the model's timing information according to the requirements of model simulation. It also includes the monitoring of data abnormalities and the collection of port data.

1) Code translation for embedded control device:

To obtain the code for device, the intermediate code representation is traversed recursively from the outside to the inside according to the XML node elements. Each XML element in the intermediate code representation will be generated with a specific code or an empty code. For the function in the intermediate code representation, the function header must be generated according to the input and output information. In order to support the code generation of multiple output ports, the output port will be generated pointer type function parameters. Then the corresponding C code is generated according to the schedule logic of the function. For each of the branch nodes, statements such as If or Switch in C language can be generated according to the type and conditions of the branch. In order to connect the execution logic of the actors, the data source variables of each actor need to be assigned by their precursors. And the data calculated by the previous actor will be transferred according to the connection relation between two actors. For the actor connected to the input port of the composite actor, its data source variable is directly replaced by the variable corresponding to the input port of the composite actor. For actors that are connected back to the output port of the composite actor, it is necessary to assign a value to the pointer variable of the output port.

An intuitive example of the intermediate code representation translated into C code is shown in Figure 6. The intermediate code representation shown is generated by the broken loop model in Figure 4. In the intermediate code representation, the Calculate node is generated from the ActorExe in the CGT. It should be noted that the "DelayValue" parameter in the model corresponds to a global variable in the intermediate code representation, so it is not reflected in the example.

2) Code translation for simulation:

Based on the intermediate code representation with clear logic, the simulation efficiency can be improved by running the code for simulation. According to the model simulation configuration parameters in the intermediate code representation, such as the start time and end time of the model simulation and the execution cycle of the controller, the C code that can be executed iteratively is generated. Specifically, the main logic of the top-level model is packaged into a loop that simulates the passage of time. For the actors in the model, there is no need to process the time information separately. Because in the embedded control system, the model is basically triggered according to a fixed step, and the execution interval of the actors is also fixed. So in fact, timing-related actors often use data queues, data pools and other structures to process data delays. Each function node in MIR will be translated into a function in C code. And the child node ref of Calcualte represents the traceability relationship between the code and

```
<Function Name="Accumulator_Update" Type="FunctionExec">
  <Ref Actor="Composite" Path="Composite"/>
  <Input Type="f32" Name="in">
    <Ref Actor="Composite" Path="Composite.in"/>
  </Input>
  <Output Type="f32" Name="out" Source="AddSubtract.out">
    <Ref Actor="Composite" Path="Composite.out"/>
  </Output>
  <Schedule>
    <LocalVariable Name="ReadVariable_out" Type="f32">
      <Ref Actor="Expression" Path="Composite.ReadVariable.out"/>
    </LocalVariable>
    <LocalVariable Name="AddSubtract_out" Type="f32">
      <Ref Actor="AddSubtract" Path="Composite.AddSubtract.out"/>
    </LocalVariable>
    <Calculate Name="ReadVariable">
      <Ref Actor="Expression" Path="Composite.ReadVariable"/>
      <Output Name="out" Type="f32"/>
      <Expression Expression="out=DelayValue"/>
    </Calculate>
    <Calculate Name="AddSubtract">
      <Ref Actor="AddSubtract" Path="Composite.AddSubtract"/>
      <Input Name="add1" Type="f32" Source="Composite.in"/>
      <Input Name="add2" Type="f32" Source="ReadVariable.out"/>
      <Output Name="out" Type="f32"/>
      <Expression Expression="out=add1+add2"/>
    </Calculate>
    <Calculate Name="SetVariable">
      <Ref Actor="SetVariable" Path="Composite.SetVariable"/>
      <Input Name="in" Type="f32" Source="AddSubtract.out"/>
      <Expression Expression="DelayValue=in"/>
    </Calculate>
  </Schedule>
</Function>
```

```
/* Update function of "Composite" */
void Accumulator_Update(float in, float* out) {
    /* Local variable of "Composite.ReadVariable.out" */
    float ReadVariable_out;
    /* Local variable of "Composite.AddSubtract.out" */
    float AddSubtract_out;
    /* Calculate code of "Composite.ReadVariable" */
    ReadVariable_out = DelayValue;
    /* Calculate code of "Composite.AddSubtract" */
    AddSubtract_out = in + ReadVariable_out;
    /* Calculate code of "Composite.SetVariable" */
    DelayValue = AddSubtract_out;
    /* Output variable assign of "Composite.out"*/
    *out = AddSubtract_out;
}
```

Fig. 6. The intermediate code representation of the example model in Figure 3 and the corresponding translated C code. The intermediate code representation is shown above, and the C code translated from it is shown below.

the model. For an example of the translation function, please refer to Figure 6.

For the requirement of automatic error detection for model simulation, we add data anomaly detection code to the port data of the actor. That is, when the translation actor executes the unit, insert a piece of code after its output port assignment. This code includes data overflow detection, division by zero detection, array out of bounds, data type conversion errors, etc. Of course, these can only provide basic error detection functions. To manually observe and find errors through the waveform diagram of the port output value, we need to output the data of the observation port specified in the model. Since the model simulation is separated from the design tool user interface under the MDD framework, the port data during simulation can be output in a variety of ways such as text and graphics. In addition, if the design tool is independently developed, then the code for data communication with the design tool can be added to the simulation code so that the simulation results can be more conveniently observed on the design tool user interface.

3) Code translation for others:

Except for the simulation and deployment, there are many other actions that can be optimized, such as testing or interaction. For example, traditional automated model-based testing has the same inefficiency problem as the simulation at the model level. Using a code translator to automatically generate the corresponding code for testing can greatly improve the efficiency of model testing. Since the intermediate code representation contains the mapping information of the model actors, once a model error is found, it is easy to locate the error actor. The essence of model automation testing is to generate as many test cases as input to repeatedly execute the model, and in this way to trigger potential errors in the model, such as crashes caused by overflow or wild pointers.

In practice, the industry usually uses actor or code coverage information to guide the generation of test cases. If we want to collect branch coverage information of the model, the feedback code triggered by the branch will be inserted near the branch code when translating the branch node of the intermediate code representation. If we want to collect the coverage information of the model actors, we can add the feedback code triggered by the actor to the actor execution logic code when translating each actor execution unit of the intermediate code representation.

Furthermore, code translation can be used for simulation docking with other design tools. We only need to translate the intermediate code representation into an FMU that conforms to the FMI standard. FMU is supported by many design tools, and the realization of this function only needs to be performed on the basis of code translation for deployment to embedded control devices. We just need to expose the standard input and output ports and pack them in a specific zip file organization.

D. Implementation

The MDD is implemented in C++ language, with 41151 lines of code. It contains three parts, model parser, schedule converter and code translator. The model parser currently contains two parts, the Simulink model parser and the Ptolemy-II model parser. As the model of the Simulink and Ptolemy-II project is saved in XML format, we used the TinyXML library to parse the model file. The schedule converter takes one or more MIR files in XML format as input and takes intermediate code representation files also in XML format as output. We have implemented three code translators for different purposes, which are used for deployment to devices, simulation, and automated testing. We uploaded the tool set for public use of embedded software design in the website. [1]

IV. Evaluations

For evaluation, we conducted experiments on benchmark models and real projects to demonstrate the effectiveness of MDD in simulation acceleration, code generation optimization and development collaboration. We used the benchmark model provided in Ptolemy-II and the equally built Simulink model for quantitative comparison, especially for the speed of simulation, and the size and execution time of the generated code. The benchmark of our experiment contains various types

[1]The MDD tools can be downloaded at: https://github.com/CodeGen456/MDDTools.

of models, including mathematic models, complex branch model, models for different application scenarios and so on. For example, HeteroMK is state machine model. PiSquare and Math are mathematic models but for different operations. Mergeing, Nesting and Crossing are branch models for different function curves. To demonstrate the collaborative capabilities of MDD, we conducted experiments on HUAWEI's real industrial project of self-driving control software.

The use of MDD requires very little preparation. Since MDD is a back-end simulation and code generation framework, it needs to use the GUI of other tools as the front-end. For any model supported by MIR, we need to use the GUI of other tools to build the model. Then take the model file as input and pass it to each layer of MDD through the command line. For example, in our experiment, we use Ptolemy's model file as input. Ptolemy-II Model Parser is first called to generate MIR, then Schedule Converter is used to convert MIR into intermediate code representation, and finally, Code Translator is used to generate code. The Model Parser, Schedule Converter and Code Translator are all used console commands to interact with files as input and output.

A. Can MDD accelerate model simulation?

We used the six benchmark models provided in Ptolemy-II and the manually constructed four models with complex branch structures to compare the simulation efficiency. The corresponding Simulink models with equal semantics are built manually. For all models, we used a frequency of 1000 Hz and a total simulation time of 1000 seconds for simulation.

Since MDD is a decoupled design tool framework, the simulation module is independent of the user interface of the design tool. Writing the simulation results, such as the output of the Display actor, to files is only a way to interact with the user interface. Of course, Socket or shared memory can also be used. These methods will not affect the accuracy of the simulation. However, Simulink and Ptolemy-II do not write the simulation results to the file during the simulation process, so for a fair comparison, we commented out the line of code written to the file in the simulation code generated by MDD. The result is shown in column 3 of Table II.

The results of the third column demonstrate that the simulation based on the generated code has better performance. Compared with Simulink, the simulation time is shortened by 92.6%. Compared with Ptolemy-II, the simulation time is shortened by 98.9%. Furthermore, using MDD to simulate a more complex model with branch structure is even more efficient and can shorten the simulation time of traditional methods by 96.7%. The reason for the acceleration is that Simulink implements the simulation of the model by executing the actors one by one, and each iteration of the model includes the operation of the actor scheduling queue. For the model with branch structure, Simulink must dynamically adjust the actor scheduling queue according to the branch judgment conditions, which results in even worse performance in the simulation. The simulation logic of Ptolemy-II is the same as Simulink. But unlike Simulink, Ptolemy-II will not analyze and fix the type on the port before simulation. In this way, before each actor is executed, it is necessary to judge or even convert the data type passed to the corresponding ports. In addition, Ptolemy-II's implementation of the emulator in Java also led to a slower simulation speed. On the one hand, the simulation process of MDD avoids dynamic schedule analysis and type judgment, because they have been implemented in the schedule conversion stage. On the other hand, since the generated simulation code is based on the C language, and the optimization function of the compiler is used for compilation and optimization, a more efficient simulation program can be obtained by directly executing the machine code.

TABLE II
COMPARISON OF SIMULINK, PTOLEMY-II AND MDD.

Model	Design Tool	Simulation Time (s)	Code Lines	Code Run Time (s)
ClockRamp	Simulink	0.964	77	4.651
	Ptolemy-II	13.27	11063	78.05
	MDD	**0.296**	**17**	**3.416**
HelloWorld	Simulink	0.691	67	9.481
	Ptolemy-II	12.91	11000	173.5
	MDD	**0.152**	**15**	**5.448**
HeteroMK	Simulink	1.328	151	5.221
	Ptolemy-II	89.56	12423	196.2
	MDD	**0.296**	**48**	**3.351**
Math	Simulink	1.106	80	5.813
	Ptolemy-II	30.62	11797	189.6
	MDD	**0.312**	**25**	**4.756**
PiSquare	Simulink	0.998	104	60.42
	Ptolemy-II	27.09	11520	168.9
	MDD	**0.677**	**22**	**46.68**
ScaleCFlat	Simulink	1.045	80	17.47
	Ptolemy-II	15.52	9873	328.5
	MDD	**0.294**	**19**	**13.60**
Mergeing	Simulink	4.765	84	3.566
	Ptolemy-II	17.87	11556	59.46
	MDD	**0.616**	**25**	**2.812**
Nesting	Simulink	4.622	99	3.156
	Ptolemy-II	24.09	12175	64.35
	MDD	**0.218**	**35**	**2.777**
Crossing	Simulink	18.61	93	41.68
	Ptolemy-II	23.35	11864	331.6
	MDD	**0.140**	**27**	**16.42**
Complex	Simulink	10.51	136	1.652
	Ptolemy-II	44.07	12897	Error [2]
	MDD	**0.299**	**59**	**1.389**
Average	Simulink	4.46	97	15.31
	Ptolemy-II	29.84	11616	176.7
	MDD	**0.330**	**29**	**10.06**

B. Can MDD generate more efficient code?

We also used the ten models above to compare the code generation efficiency with Simulink Coder and Ptolemy-II C Code Generator. We know that Simulink can generate code for different hardware platforms. In our experiment, the Intel processor under the Windows system is used as the target device. Since the core logic of the generated code does not

[2]The reason for the runtime error of the code generated by Ptolemy-II for the Complex model is that the code that handles the data transfer at the branch junction does not send the token to the data queue of the input port of the subsequent actor. As a result, an error message of "No more Tokens in the DE Receiver : DEReceiver_Get ($ModelName()__DEReceiver.c)" is printed and the calculation of subsequent actors cannot be performed

have hardware-related instructions, the target platform will not affect the comparison experiment. As for the size of code, since Simulink generates many redundant type definitions and data definitions, we only selected the main file generated by Simulink for line number statistics. In terms of code runtime, we used the same compilation and runtime environment (Window10 x64, Intel x64 processor, Cygwin64 Terminal, GCC compiler) to compile and run these generated codes. In the experiments to compare code efficiency, the standard output instructions are retained in the code, this is why the code running time is longer than the simulation time.

The comparison results are shown in the 4th and 5th columns of Table II. The results demonstrate that the code generator based on MDD can generate shorter and more efficient code. In terms of lines of code, MDD can save 69.9% compared to Simulink, and compared to Ptolemy-II, it can save 99.7%. In terms of code execution time, MDD can save the execution time of 34.3% compared to Simulink, and it can save the execution time of 94.3% compared to Ptolemy-II.

The main reason is that MDD generates more concise code for schedule logic. Simulink generates much redundant code, especially for the model with branch structure. Since Ptolemy-II generated code that completely simulates the execution of the model, it needs a large number of custom c code library to support the discrete event system, such as PriorityQueue for storing port data and HashMap for data search. And Ptolemy-II generated a .h file and a .c file for each actor. Each actor contains three parts of code: reading data from the input port, calculating according to the port data, and outputting data to the output port. This is why the code generated by Ptolemy-II basically reaches tens of thousands of lines. The massive code also consumes a lot of time to execute. [3]

C. Real project study

With the help of developers in HUAWEI, we demonstrate how MDD supports collaborative development of different design tools, during the self-driving control software implementation. The model is the data feedback functional unit of the brake caliper sensors in the vehicle braking system, consists of both the Simulink sub-model and Ptolemy-II sub-model. Since the sensor data feedback functional unit in the model requires complex branch logic, the model logic implemented with Simulink is not intuitive, and we use Ptolemy-II to construct the functional unit.

The model is shown in Figure 7. For the sensor data feedback functional unit, we only built an empty composite actor in Simulink, as shown in the red box in the lower right corner of the Figure 7. The sensor data feedback function unit is implemented as a complete composite actor by Ptolemy-II. Both composite actors have the same settings, such as port names, port data types, and the composite actor name.

After that, we used the Simulink model parser and the Ptolemy-II model parser to analyze the two parts of the model respectively, so that we got the unified model intermediate representation files. Then we used the schedule converter to take the intermediate representation files of these two sub-models as input for overall schedule analysis. The definition of the brake caliper sensors data feedback functional unit exists in the MIR of the two models. This is equivalent to declaring the functional unit in Simulink and defining the functional unit in Ptolemy-II. And an intermediate code representation is generated. Next, we used code translators to generate code for simulation and C code for deployment on the device. Finally, we compiled and ran these codes, and obtained the results of simulation efficiency and code running efficiency of the model built for this collaborative approach.

TABLE III
COMPARISON ON REAL INDUSTRIAL PROJECT.

Design Tool	Actor count	Composite actor count	Simulation Time (s)	Code Lines	Code Run Time (s)
Simulink	53	12	16.33	214	41.05
Ptolemy-II	25	0	157.2	17734	Error [4]
MDD	25	0	**0.749**	**124**	**18.72**

For comparison, we also built equivalent models of the same execution logic on Simulink and Ptolemy-II with more manual efforts. The comparison result is shown in Table III. It can be seen that the simulation and code generation effects achieved by using the MDD framework are better than Simulink and Ptolemy-II. The number of actors used in the two functional units built with Simulink and Ptolemy-II is also counted. Because the representation of Ptolemy-II and Simulink can be inherited by MDD, the actor count of MDD is the same as Ptolemy-II. When modeling the same function, we found that using Simulink to build complex models is indeed more cumbersome than Ptolemy-II. This further proves that code generation that supports complex branch models can save modeling workload, and also demonstrates the importance of collaborative development of different design tools.

V. LESSON LEARNED

In the practice of MDD implementation and model-driven design collaboration with HUAWEI, we have learned some experiences.

Design tools collaboration can improve the efficiency of model-driven development and need more in-depth support. During the development of self-driving control software, there are lots of scenarios, that, when you want to implement a certain function unit in the Simulink model, but the Simulink model expression is inconvenient and you need to use other design tools. Sometimes, you also need to implement the function by a domain expert familiar with other design tools. Design tools collaboration with some unified interface is the best choice. Just as the FMI standard is proposed to solve the problem that different simulation modules constructed by multiple design tools can call each other during the simulation process. But it is also limited to the simulation of the design. Therefore, a design framework that can further integrate functions such as simulation and code generation will break the current limitations of design tools collaboration.

[3]The sample codes generated by Ptolemy-II can be seen at: https://github.com/CodeGen456/MDDTools/tree/main/CodeGeneratedByPtII.

[4]The reason for the runtime error of the code generated by Ptolemy-II for the industrial model is the same as the error in the Complex model.

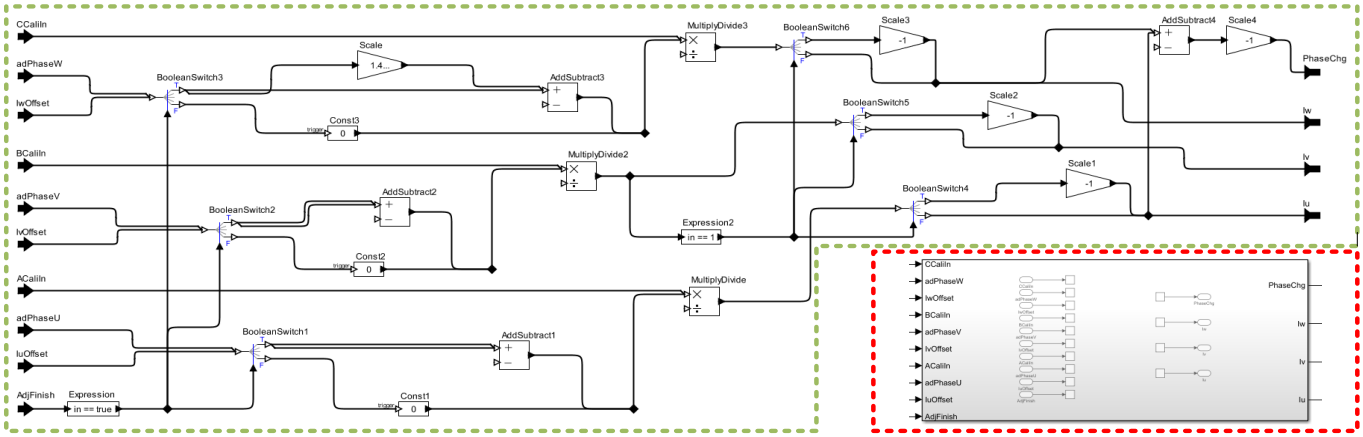

Fig. 7. The vehicle braking system model of HUAWEI's self-driving control software. The red box in the lower right corner shows the declaration of the sensor data feedback functional unit constructed by Simulink. The green box shows the specific implementation of the functional unit constructed by Ptolemy-II.

The intermediate code representation that contains the model mapping information plays an important role in bridging the gap between the model operation and code operation. Firstly, based on the intermediate code representation, code for deployment to the target devices can be generated easily. Because the main logic of the intermediate code representation is expressed in a control flow manner, it is enough to directly translate the intermediate code representation grammatically. Secondly, the intermediate code representation can also generate code for efficient simulation. Because some actors generate code in multiple code locations due to different branch paths, the model mapping information in the intermediate code representation can indicate that these codes come from the same actor. If the data of certain ports need to be observed during simulation, the model mapping information plays a role in traceability. In addition to the purpose of code generation and simulation for target devices, automated testing of models is required in many industrial scenarios. Based on the intermediate code representation generation, the code that supports test case generation and coverage analysis can also greatly improve the efficiency of testing compared to directly testing at the model level. Once the automatic test results find a model error, the model mapping information can directly locate the error to the actor in the model.

Limitations brought by packaging each composite actor as a function for code generation. In the experiment, we found that Simulink expands all the composite actors in the entire model for code generation, which causes the code generated by Simulink to have only one model logic function. Not only that, the code optimization function of Simulink Coder can fold multiple expressions into one expression to save the use of variables on the stack. Therefore, Simulink's code optimizer can optimize the code of the entire model. However, MDD packages each composite actor into a function for code generation, which limits the scope and ability of code optimization. Therefore, in future work, providing users with an option to expand composite actor may be a way to solve this limitation. Another limitation is also caused by MDD packaging each composite component into a function. We know that function call jumps consume more CPU time than executing code sequentially. When MDD generates code for a model with a lot of composite actors, especially when the logic of each composite actor is very simple, a lot of short functions and a lot of function call codes will be generated. This will cause more additional time overhead of code execution. Similarly, we can provide users with an option to expand composite actors, or we also can add inline modifiers to short functions to avoid the function call process.

VI. Conclusion

This paper proposed a unified design framework MDD to optimize the development efficiency of embedded control software, including support for design tools collaboration, more efficient simulation, and higher quality code generation. First, the model parse layer supporting different modeling tools is designed to transform different models into a unified model intermediate representation (MIR). It is extensible, so any model with different semantics can be integrated into MIR through actor packaging. Second, the schedule convert layer converts the MIR that may contain different semantics into a unified intermediate code representation. Especially for the data flow semantic model with complex branching logic, it converts the data flow logic into more concise control flow logic of code as much as possible. More concise code can be generated by the conversion algorithm, which means fewer lines of code and higher execution efficiency. Third, the code translate layer supports code generation for multiple purposes, including code for deployment to the device, code for simulation, code for testing, and so on. Since the MDD simulation process is based on the generated code, it avoids time-consuming processes such as actor triggering and type inference during the simulation process, so the simulation efficiency of MDD is higher than existing tools. Experiments demonstrate that MDD can not only support the collaborative development of Simulink and Ptolemy-II but also can improve their simulation efficiency and reduce the number of lines of generated code, and reduce the execution time of the code. In the future, we will focus on further code optimization with semantic analysis.

REFERENCES

[1] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D. Méndez, A. Moreira *et al.*, "Dealing with non-functional requirements in model-driven development: A survey," *IEEE Transactions on Software Engineering*, 2019.

[2] S. Staroletov, N. Shilov, V. Zyubin, T. Liakh, A. Rozov, I. Konyukhov, I. Shilov, T. Baar, and H. Schulte, "Model-driven methods to design of reliable multiagent cyber-physical systems," in *Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes (MACSPro 2019)*, 2019.

[3] P. Bocciarelli, A. D'Ambrogio, A. Falcone, A. Garro, and A. Giglio, "A model-driven approach to enable the simulation of complex systems on distributed architectures," *Simulation*, vol. 95, no. 12, pp. 1185–1211, 2019.

[4] C. M. Sosa-Reyna, E. Tello-Leal, and D. Lara-Alabazares, "Methodology for the model-driven development of service oriented iot applications," *Journal of Systems Architecture*, vol. 90, pp. 15–22, 2018.

[5] Simulink and Matlab, *Simulink Documentation*. [Online]. Available: https://www.mathworks.com/help/simulink/index.html

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," in *Readings in Hardware/Software Co-Design*, ser. Systems on Silicon, G. De Micheli, R. Ernst, and W. Wolf, Eds. San Francisco: Morgan Kaufmann, 2002, pp. 527–543.

[7] P. Baldwin, S. Kohli, and E. A. Lee, "Modeling of sensor nets in ptolemy ii," in *Proceedings of the 3rd international symposium on Information processing in sensor networks*. ACM, 2004, pp. 359–368.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[9] H. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in ptolemy ii," in *Proceedings of the 2001 American Control Conference.(Cat. No. 01CH37148)*, vol. 6. IEEE, 2001, pp. 4984–4985.

[10] J. Bastian, C. Clauß, S. Wolf, and P. Schneider, "Master for co-simulation using fmi," in *Proceedings of the 8th International Modelica Conference; March 20th-22nd*, no. 63. Linköping University Electronic Press, 2011, pp. 115–120.

[11] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *Proceedings of the 8th International Modelica Conference*. Linköping University Press, 2011, pp. 105–114.

[12] C. Gomes, B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere, "Semantic adaptation for fmi co-simulation with hierarchical simulators," *Simulation*, vol. 95, no. 3, pp. 241–269, 2019.

[13] P. Filipovikj, N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, and H. Lönn, "Simulink to uppaal statistical model checker: Analyzing automotive industrial systems," in *International Symposium on Formal Methods*. Springer, 2016, pp. 748–756.

[14] Y. Yang, Y. Jiang, M. Gu, and J. Sun, "Verifying simulink stateflow model: timed automata approach," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 852–857.

[15] Z. Lu, R. Wang, and Y. Guan, "Formal verification of discrete event model," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things*, 2020, pp. 3–4.

[16] F. Boulanger and C. Hardebolle, "Simulation of multi-formalism models with modhel'x," in *International Conference on Software Testing*, 2008.

[17] D. A. Adams, "A computation model with data flow sequencing." Stanford University, 1969.

[18] C. Ptolemaeus, *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014, vol. 1.

[19] H. Klee and R. Allen, *Simulation of dynamic systems with MATLAB and Simulink*. Crc Press, 2016.

[20] E. A. Lee and Y. Xiong, "A behavioral type system and its application in ptolemy ii," *Formal Aspects of Computing*, vol. 16, no. 3, pp. 210–237, 2004.

[21] H. Hanselmann, U. Kiffmeier, L. Koster, M. Meyer, and A. Rukgauer, "Production quality code generation from simulink block diagrams," in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*. IEEE, 1999, pp. 213–218.

[22] G. Zhou, M.-K. Leung, and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," in *International Conference on Embedded Software and Systems*. Springer, 2007, pp. 193–206.

[23] T. Miyazaki and E. A. Lee, "Code generation by using integer-controlled dataflow graph," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. IEEE, 1997, pp. 703–706.

[24] T. Z. Asici, B. Karaduman, R. Eslampanah, M. Challenger, J. Denil, and H. Vangheluwe, "Applying model driven engineering techniques to the development of contiki-based iot systems," in *Proceedings of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things*. IEEE Press, 2019, pp. 25–32.

[25] K. Jahed and J. Dingel, "Enabling model-driven software development tools for the internet of things," in *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*. IEEE Press, 2019, pp. 93–99.

[26] F. Pasic, "Model-driven development of condition monitoring software," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, 2018, pp. 162–167.

[27] Y. Jiang, H. Song, Y. Yang, H. Liu, M. Gu, Y. Guan, J. Sun, and L. Sha, "Dependable model-driven development of cps: From stateflow simulation to verified implementation," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 1, p. 12, 2018.

[28] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[29] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 03, pp. 261–303, 2003.

[30] F. Balarin, P. Giusto, A. Jurecska, M. Chiodo, C. Passerone, E. Sentovich, H. Hsieh, L. Lavagno, B. Tabbara, A. Sangiovanni-Vincentelli *et al.*, *Hardware-software co-design of embedded systems: the POLIS approach*. Springer Science & Business Media, 1997.

[31] H. Zhang, Y. Jiang, H. Liu, M. Gu, and J. Sun, "Tsmart-bipex: An integrated graphical design toolkit for software systems." in *D&P@ MoDELS*, 2016, pp. 32–39.

[32] L. Berzi, T. Favilli, M. Pierini, L. Pugi, G. B. Weiß, N. Tobia, and M. Ponchant, "Brake blending strategy on electric vehicle co-simulation between matlab simulink® and simcenter amesim™," in *2019 IEEE 5th International forum on Research and Technology for Society and Industry (RTSI)*. IEEE, 2019, pp. 308–313.

[33] M. Bagheri, M. Sirjani, E. Khamespanah, N. Khakpour, I. Akkaya, A. Movaghar, and E. A. Lee, "Coordinated actor model of self-adaptive track-based traffic control systems," *Journal of Systems and Software*, vol. 143, pp. 116–139, 2018.

[34] V. I. GmbH, *DaVinci Developer*. [Online]. Available: https://www.vector.com/us/en-us/products/solutions/autosar-classic/

[35] A. Enrici, L. Apvrille, and R. Pacalet, "A uml model-driven approach to efficiently allocate complex communication schemes," *Springer International Publishing*, 2014.

[36] T. Schattkowsky, J. H. Hausmann, and G. Engels, "Using UML activities for system-on-chip design and synthesis," in *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4199. Springer, 2006, pp. 737–752.

[37] G. Vanwormhoudt, M. Allon, O. Caron, and B. Carré, "Template based model engineering in uml," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 47–56.

[38] R. Ahmadi, E. Posse, and J. Dingel, "Slicing uml-based models of real-time embedded systems," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 346–356.

[39] Z. Su, D. Wang, Y. Yang, Y. Jiang, W. Chang, L. Fang, W. Li, and J. Sun, "Code synthesis for dataflow based embedded software design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[40] H. Zhang, Y. Jiang, H. Liu, H. Zhang, M. Gu, and J. Sun, "Model driven design of heterogeneous synchronous embedded systems," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 774–779.

[41] Y. Jiang, H. Song, H. Kong, R. Wang, and L. Sha, "Safety-assured model-driven design of the multifunction vehicle bus controller," *IEEE Transactions on Intelligent Transportation Systems*, 2017.