Code Synthesis for Dataflow-Based Embedded Software Design

Zhuo Su, Dongyan Wang, Yixiao Yang, Yu Jiang, Wanli Chang, *Member, IEEE*, Liming Fang, *Member, IEEE*, Wen Li, and Jiaguang Sun

***Abstract*—Model-driven methodology has been widely adopted in embedded software design, and Dataflow is a widely used computation model, with strong modeling and simulation ability supported in tools such as Ptolemy. However, its code synthesis support is quite limited, which restricts its applications in real industrial practice. In this article, we focus on the automatic code synthesis of Dataflow, and implement *DFSynth*, a code generator that could support most of the widely used modeling features, such as the expression type and Boolean switch, more efficiently. First, we disassemble the Dataflow model into actors embedded in if-else or switch-case statements based on the schedule analysis, which bridges the semantic gap between the code and the original Dataflow model. Then, we design well-designed templates for each actor, and synthesize well-structured executable C and Java codes with sequential code assembly. Compared to the existing C and Java code generators of Dataflow model in Ptolemy-II, and the C code generator in Simulink, the lines of code synthesized by *DFSynth* are decreased by an average of 99.7%, 81.4%, and 61.9%, and the execution time of the synthesized code by *DFSynth* is also decreased by an average of 76.2%, 56.8%, and 22.7%, respectively.**

***Index Terms*—Code synthesis, dataflow, model-driven design, Ptolemy, Simulink.**

I. Introduction

MODEL-DRIVEN design has been widely adopted in the embedded system domain, and Dataflow is a widely used model of computation with strong modeling and simulation ability [1]–[4]. In Dataflow, various basic functions are packed into actors with input and output ports, such as the Addsubtract actor to perform the addition and subtraction. Advanced features also support composite actors and hierarchical finite-state machines to model the system structure and control logic. Those actors are usually combined in a Dataflow model to describe the whole system.

Dataflow model and its support platforms, such as Ptolemy-II, are attracting increasing attention in both academia and industry [5]–[8]. However, stronger modeling and simulation capability would bring more difficulties for model verification and synthesis. The limited synthesis ability of the code generator would strictly restrict the application scenarios, as most developers want to support not only modeling and simulation-based analysis but also code synthesis-based implementation to reduce the coding efforts.

To improve the model synthesis ability, Edward A Lee developed a C language code generator and a Java language code generator for Dataflow [9], which have been successfully integrated into the famous Ptolemy-II and works well in certain cases. But for more practical embedded system models, there are three main weaknesses in the existing code generators.

1) Many commonly used modeling actors and features are not supported. For example, the Expression actor, which is used to model customized input ports and execute customized expressions and operations, is not allowed in the original code generator. The model with the counter actor, record actor, and limiter actor cannot be handled either.
2) The synthesized code contains lots of redundant files. For example, each actor would be wrapped into two .c and .h files, most of which are redundant. Furthermore, the original code generator synthesizes the code exactly as the execution of the time series and a lot of codes only aims to deal with the data transmission and data-type conversion, which makes the files more overlap. It would synthesize tens of thousand lines of C code for a model with two simple actors, and the huge size of the code makes it even harder to maintain.
3) The synthesized code omits the structure information. For example, the composite actors are ignored and each actor has its corresponding .c and .h files, and the flattened code cannot reflect the hierarchical structures. Both the synthesized C and Java codes pass their values in the form of event queues and data flow of separate actors. The synthesized code needs to separately retrieve all events accumulated on its input port during the execution, perform addition and subtraction

Manuscript received August 27, 2020; revised December 10, 2020; accepted January 16, 2021. Date of publication January 29, 2021; date of current version December 23, 2021. This work was supported in part by NSFC Program under Grant 62022046, Grant U1911401, and Grant 61802223; in part by the National Key Research and Development Project under Grant 2019YFB1706200; and in part by the Huawei-Tsinghua Trustworthy Research Project under Grant 20192000794. This article was recommended by Associate Editor C. Yang. *(Corresponding author: Yu Jiang.)*

Zhuo Su, Yixiao Yang, Yu Jiang, and Jiaguang Sun are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mail: suzcpp@gmail.com; jiangyu198964@126.com).

Dongyan Wang is with the School of Information Science and Technology, Peking University, Beijing 100871, China (e-mail: wdy18@pku.edu.cn).

Wanli Chang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, and also with the Department of Computer Science, University of York, YO10 5DD, U.K. (email: wanli.chang@york.ac.uk).

Liming Fang is with the College of Computer Science and Technology and the State Key Laboratory of Cryptology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: fangliming@nuaa.edu.cn).

Wen Li is with Gödel Labs, HUAWEI Technologies Company Ltd., Hangzhou 310000, China (e-mail: coco.liwen@huawei.com).

Digital Object Identifier 10.1109/TCAD.2021.3055487

operations, and heap the result to the input port of the next actor. Furthermore, the information of logical branch actors such as the BooleanSwitch actor is omitted in the synthesized code.

To satisfy the increasing requirements of more advanced embedded system design and improve the usability of Dataflow as well as its supporting platforms, we need to support more modeling actors and features with a more efficient code generator. It is not possible to use those limited number of actors supported in the current generator during the system modeling process. Furthermore, for many embedded systems with limited memory and computing resource, it is not reasonable to load and maintain tens of thousand lines of unstructured codes. For an efficient code synthesis of the Dataflow model, there are mainly three challenges.

Challenge 1 (How to Synthesize Code for the Logical and Scheduling Relationship of Actors): Dataflow is implemented by passing data as tokens to the input ports of each actor for calculation, and the actor can be executed when it has input data. For the actors that control the direction of Dataflow, it is not able to be handled by the original execution semantics. Taking an example of the BooleanSwitch actor, the data of the input port would be diverted to the TrueOutput port or the FalseOutput port according to the value of the control port. To generate the code with clear logic, the BooleanSwitch actor needs to be converted to a control statement such as if-else; otherwise, only a complex redundant code with token passing would be synthesized.

Challenge 2 (How to Define Templates to Facilitate Code Synthesis for Actors With Dynamic Data Types): Considering that most actors contain both input ports and output ports and perform some calculations on some parameters, so it seems that each actor can be coded to an expression with inputs, outputs, and parameters. However, there are many more complex situations. For example, many actors can define a new port and many ports can be connected to more than one port, and many actors are designed to deal with structured data and data-type conversion. An important actor, named the expression actor, can execute given expressions, and we also need to handle the data-type conversion and data type of the execution result. How to support those situations in a unified template for each actor is a complicated problem.

Challenge 3 (How to Preserve the Structural Information of the Hierarchical Model): In Dataflow, composite actors wrap a part of the computational model into an actor and only the input ports and output ports are publicized. The actor and state allow refinement to model the hierarchical structure of actors. Hence, the synthesized code should also present the structural information for better illustration, e.g., a composite actor or a state machine needs to be synthesized as a function, which could be executed with a function call.

In this article, we propose a strengthened code generator *DFSynth*[1] to address the above challenges and synthesize more efficient and structured code for the embedded system design. It mainly consists of three steps. First, data flow branch-sensitive actors are translated into the if-else statement or switch-case statement. To determine the merge location, branch information on actors is marked and will be backtracked based on the schedule analysis algorithm. Then, well-designed templates are implemented to generate the header files, utility functions, variables, and execution functions for each actor, with the data types in consideration. Finally, the whole model is analyzed layer-by-layer to generate the structured function-call statements. Composite actors and state machine are encapsulated as functions, where the input ports serve as the input parameters while the output ports serve as the output parameters.

For evaluation, we first construct a complex model to verify the effectiveness of the implemented code generator, i.e., whether we can support more advanced modeling features correctly. Then, we use the benchmark examples of the original code generators and a real industrial example from Huawei for further comparison. Compared to the existing C and Java code generators of Dataflow [9], the lines of code synthesized by *DFSynth* are decreased by an average of 99.7% and 81.4%, and the execution time of the code synthesized by *DFSynth* is decreased by an average of 76.2% and 56.8%. We also build the corresponding Stateflow model of the Dataflow model benchmark, synthesize the C code with Simulink [10] for further comparison, and *DFSynth* reduces the lines of code and the execution time with an average of 61.9% and 22.7%, respectively.

II. Background

A. Model-Driven Development

Model-driven development is widely used for the system design and mainly consists of model construction, model validation, and code synthesis. There have been many studies related to model-driven development [11]–[16], and there are many related tools, such as Tsmart, Simulink, SCADE, and Polychrony [10], [17]–[23]. Among them, Simulink is a widely used commercial design environment and supports the code synthesis of the Stateflow model with many advanced features, such as structure and expression calculation. SCADE is also a widely used commercial design environment with automatic code synthesis for a safe-state machine model. Ptolemy-II is an opensource environment that supports Dataflow and widely used for the modeling and simulation of modern embedded systems. For example, Kim *et al.* [24] combined the gem5 simulator with Ptolemy-II to create power and thermal model for a DRAM. Bagheri established an adaptive rail-based control system, with the hierarchical modeling capability of Ptolemy-II [25], [26].

B. Models of Computation

The Dataflow model consists of actors and data connections between the ports of the actor. Actors are used to receive data, process data, and send data. The connections between ports are used to transfer data. The execution order of the actors in the model depends on the topological ordering of the model. The essence of model simulation is to execute the actor from the beginning of model topology sorting. The actor sends data

[1]The code of *DFSynth* is open-source and can be downloaded at: https://github.com/CodeGen123/DataflowCodeGeneration.

to the output ports according to the input data and then passes the data to the subsequent actor for calculation. The port of the branch actor sometimes does not output data, even if it has a data connection to the subsequent actor, it will not trigger the subsequent actor. There can be a composite actor in the Dataflow model. The inside of the composite actor can be another Dataflow model or a state machine model. The external and internal parts of the composite actor transmit data through ports. When the outer model is executed to this composite actor, the internal model of the composite actor will be executed. After the internal model is executed, it will return to the outer model to continue execution.

Stateflow is a computational model used in Simulink. It mainly consists of two models: 1) data flow and 2) finite-state machine. The semantics of the data flow model in Stateflow is consistent with the aforementioned Dataflow model. The state machine model in Stateflow has a more powerful expressive ability than the ordinary state machine models. The state of the state machine allows the definition of substates, and various events can be defined on the state, such as entry, during, and exit events. In addition, more complex operations in the transition process, such as conditional judgments, loops, etc., can be implemented through junction nodes and transition connections between junction nodes.

The simulation process of the Dataflow model and the Stateflow model is iterative. The simulator will execute the entire model once on each timestamp, which is the same as the step function of our embedded program. The process of each iteration is considered to be completed instantaneously in the simulator and real embedded system, and the actors in the model are also considered to be executed instantaneously, but their execution order is restricted by the topological sort.

C. Code Synthesis of Dataflow

Two code generators were implemented for Ptolemy Dataflow models, one is the C code generator and another one is the Java code generator [9]. Both synthesize the executable code with a flattened event queue and have been integrated as the standard code generator in Ptolemy-II. The C code generator generates the code with complete discrete event calculations and the code mainly consists of three parts: 1) the fundamental data structure; 2) model scheduling; and 3) actor execution. The fundamental data structure mainly contains all the basis for simulating event transmissions, such as actor elements, events, queues, and so on. The model scheduling is primarily responsible for scheduling actors in a topological sort order. Based on the data in the input queue, the actor execution file is synthesized for each actor to calculate the result and then output it to the input queue of the next actor. The Java code generator consists of two parts: one is the basic data type conversion and code comparison and another one is the model scheduling and code computation. It also uses the method of the event queue for data transmission, but it uses the queue implemented by the fixed-length array. The data load of the fixed-length array is limited, and a lot of extra space will be occupied. Furthermore, it only supports the sequential computation of actors and does not support dynamic operations, such as branch selection or state machine.

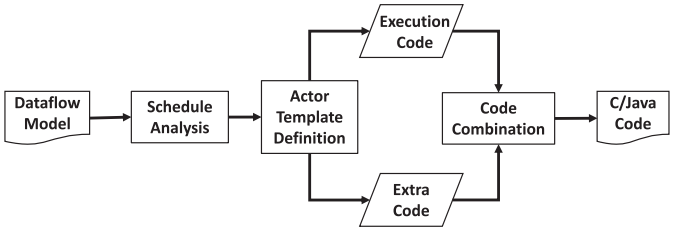


Fig. 1. *DFSynth* workflow, the code synthesis of Dataflow.

Simulink coder can generate the code for discrete Simulink models. This code generator is relatively robust, it can generate the code for almost any valid Simulink model. However, there is no data branch actor in the Simulink model. If you want to implement data branching, you must model in a very complicated way, which greatly limits the expressive ability of the model. Of course, this does reduce the difficulty of code generation for the Simulink coder. There are many works to generate the code for Dataflow models, but they tend to generate codes for a specific feature, rather than considering the entire model. For example, Stavros generates the code for the composite actors in the synchronous Dataflow model [27]. Miyazaki and Lee proposed a code generation method for the integer-controlled dataflow (IDF) model [28].

Although the above code generators work well in some cases, it cannot catch up with the real system design requirements. First, the code synthesized by these code generators not only contains a large size of redundant code but also does not retain the original hierarchical model structure. They expand the composite actors of all levels, which will lead to the code review quite troublesome. Second, both of these code generators only support few actors and it is difficult to extend them for actors with dynamic types.

III. DFSynth Design

We propose *DFSynth*, an efficient code synthesis approach for the Dataflow model. Rather than synthesizing the Dataflow-oriented code with the flattened event queue, our goal is to generate short, structured, and control flow-oriented sequential code for better efficiency. *DFSynth* follows three steps: 1) analyze schedule order; 2) define template for each actor; and 3) combine code of entire model, and the overview is presented in Fig. 1.

First, we need to analyze the schedule order, which includes the execution order of actors and the logical relationships caused by branch-sensitive actors. We just mark the branch information on each actor of the Dataflow model. The conditions for execution of each actor will be calculated by this step, and it can indicate where the code of each actor should lie, within an if statement or an else statement. Second, the code synthesis template should be defined and applied to each actor. All possible actors in the model need to be defined actor templates. The actor template will generate the corresponding code for each actor according to the type of actor and the data connection relationship in the model. On the one hand, the execution code contains the main logic of each actor, and on the other hand, an extra code must be synthesized to ensure

Algorithm 1 Overview of Schedule Analysis

Input: *model*: the hierarchical dataflow model
Output: execution order and branch labels

```
1:  Function executionOrderAnalysis(model)
2:      // sm means sorted model
3:      var sm = topologicalSort(model);
4:      // lom means layers of model
5:      var lom = markLayers(sm);
6:      // lomwta means layers of model with TempActor
7:      var lomwta = addTemporaryActor(lom);
8:      return lomwta
9:  Function logicRelationshipAnalysis(lomwta)
10:     // lwbf means layers with branch flags
11:     var lwbf = markBranchFlag(lomwta);
12:     // lwfbf means layers with final branch flags
13:     var lwfbf = mergeResultOfMark(lwbf);
14:     return lwfbf
```

integrity, such as the header files, variables used to store the state, etc. Finally, after we use the actor template to generate the code for each actor in the model, we can assemble them according to the calculated model scheduling relationship. Besides, we need to wrap the composite actors and state machines as functions and integrate header files and global variables to generate complete executable code.

A. Schedule Analysis

The schedule analysis mainly infers the execution order of actors and the logic relationships caused by branch-sensitive actors. The execution order is determined by the topological order of the actors, which is consistent with the semantics of the Dataflow. There may be many scheduling orders in the execution process of the model. Here, we only sort the possible execution relationship of the actors in the model and mark the calling order of the actors with branch information, just as we use an if-else statement to control the scheduling order when we write C programs. In the subsequent code generation process, that is, Algorithm 4, the actors in the model will be generated into the corresponding program statement block according to their branch mark information to achieve the scheduling logic consistent with the model. The overview is presented in Algorithm 1.

The method of topologicalSort() in Algorithm 1 works in a recursive manner. It will find the actors without input data, delete their corresponding connected successor actors, and then continue to find the actors without input data. For the loop situation, it could be broken by the two corresponding actors that can pass variables.

The method of markLayers() is used for the actors in the same topological sort order, which should be classified into a layer. The first layer is labeled with layer 1, then increased by 1 for each subsequent layer accordingly. All actors of the first layer are connected to the *Root* node, and all actors of the last layer are connected to the *LastRoot* node. Both *Root* node and *LastRoot* node are empty without any information. Some

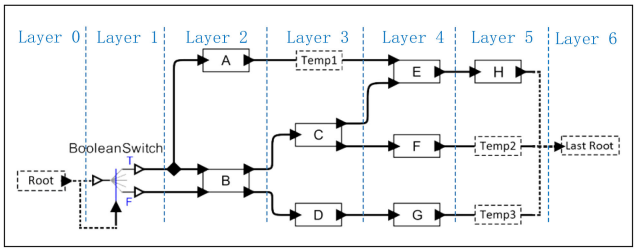


Fig. 2. Example of the execution order with five layers.

Algorithm 2 Mark the Branch Flag for Actors

Input: *Layers*: Layers of actors
Output: Preliminary branch information label

```
1:  var predecessors
2:  var successors
3:  var root = Layers[0]
4:  for all successor of the the root do
5:      if isBranchActor(root) then
6:          addFlagByBranch(successor);
7:      else
8:          addFlagZero(successor);
9:      successors.insert(successor);
10: while true do
11:     predecessors = successors
12:     successors.clear();
13:     for all actor in the predecessors do
14:         if isBranchActor(actor) then
15:             markBranchFlag(actor); //Call recursively
16:         for all successor of actor do
17:             inheritFlag(successor, act);
18:             successors.insert(successor);
19:     if isAllSatisfyC(successors) then
20:         // X means the layer of successors
21:         layerLabel(X);
22:         break
23: return Layers
```

temporary actors should be added by the method of addTemporaryActor() between two adjacent actors with different layer numbers. In this way, we could ensure that the actors in each layer represent all the Dataflow in the model. A simple example is shown in Fig. 2, which consists of five calculation layers and contains three temporary actors.

Then, we deal with the logic relationships caused by the branch-sensitive actors, such as the BooleanSwitch actor and switch actor. These actors will be synthesized to the if-else or switch-case statement. We need to mark all actors with branch information flags and identify where these branches should merge.

The method of markBranchFlag() is refined in Algorithm 2. It marks the sorted actors with branch flags layer by layer. If a branch-sensitive actor is encountered, one more flag set will be added to the subsequent actors, as presented in lines 14 and 15. For the *Root* node, it is marked with flag 0 (representing the first branch), and then all subsequent actors will inherit

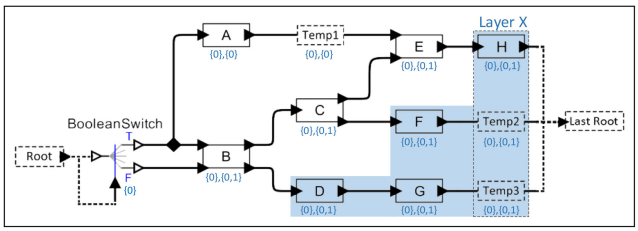


Fig. 3. Example of the branch mark and merge.

flag 0, as presented in line 8. If a BooleanSwitch actor is encountered, a new flag set will be added. The actor after the TrueOutput port should be marked with 0 while the actor after the FalseOutput port should be marked with 1, as presented in line 6. Eventually, all actors will be marked and there will be some actors that are simultaneously marked with 0 and 1 in the current flag set. Then, we define a composite condition C that the actor is marked with 0 and 1, and all its precursor actors are also marked with 0 and 1. If all actors of layer X satisfy the composite condition C, they must not be controlled by the branch-sensitive actor, and the marking algorithm will finish as presented in lines 19–22. When the code is synthesized, layer X can be synthesized outside of the if-else statement, because the data source of these actors is the same, regardless of which branch they belong to. In the worst case of this algorithm, *LastRoot* will become the layer X.

Fig. 3 shows the flags on each actor, and we can see that all actors after the BooleanSwitch actor have a new flag set that contains the branch information caused by the BooleanSwitch actor. The branch information indicates whether the actor's code should be on which branch or both of the if-else statement. In this example, actors B, C, D, E, F, G, and H can be synthesized on both branches of the if-else statement, and actor A can be synthesized on the TRUE branch.

The method of mergeResultOfMark() is refined in Algorithm 3 and is responsible for the final branch merge. As presented in Fig. 3, actors F, D, and G should be merged outside of the synthesized if-else statement. Actor C cannot be merged because actor E, which is behind actor C, cannot be merged. This means that when the actor cannot be merged, all its predecessors cannot be merged. The branch merge algorithm accomplishes the task with the following four recursive steps.

1) In lines 1–3, it traverses all the actors of layer X. If there is a predecessor of an actor that does not satisfy the composite condition C, then all predecessor actors of this actor are marked with flag *Break*.
2) In line 4, it initializes the current layer with the predecessors of layer X.
3) In lines 6–8, it traverses the actors of the current layer, if an actor has a flag of *Break*, then mark all its predecessor actors with the flag of *Break*.
4) In lines 9–15, if all actors of the current layer are marked with *Break*, the algorithm ends. If not, it deletes their branch flags. If there is a predecessor of an actor that does not satisfy condition C, all predecessor actors of this actor will be marked with *Break*. In line 16, it will initialize the current layer with the previous layer and jump to the third step for iteration.

Algorithm 3 Merge the Final Branches for Actors

Input: *Layers*: Layers of actors with branch flags
X: Index of layer X calculated in Algorithm 2
Output: Branch set of actors

```
1:  for all actor of the Layers[X] do
2:      if not isAllPredecessorSatisfyC(actor) then
3:          setAllPredecessorBreak(actor)
4:  currentLayer = Layers[X– –]
5:  while true do
6:      for all actor of the currentLayer do
7:          if isMarkWithBreak(actor) then
8:              setAllPredecessorBreak(actor)
9:      if isAllMarkedWithBreak(currentLayer) then
10:         break
11:     for all act of the currentLayer do
12:         if not isMarkWithBreak(actor) then
13:             removeBreakFlags(actor)
14:             if not isAllPredecessorSatisfyC(actor) then
15:                 setAllPredecessorBreak(actor)
16:     currentLayer = Layers[X– –]
```

Finally, we can get the actor sets for different branches. Similar to the BooleanSwitch actor, the select actor is also branch sensitive and can be translated to the switch-case statement. We can mark the branch of the select actor using the above method. Moreover, the branch flag of the select actor is from 0 to n–1, where n is the number of input ports connected to the select actor. Based on the scheduled branches, we can generate the control-flow-oriented sequential code.

B. Actor Template

To synthesize codes for different actors, especially for those with dynamic types, we design a general template, which could be instanced by actors dynamically and contains four blocks, including the header, function, variable, and fire.

The header block includes the declarations of the library files that the actors will use. Different actors may need to use different function libraries. For example, the AbsoluteValue actor needs to use the fabs function in the "math.h" header file, and the StringConst actor may need to use the string function defined in the "string.h" header file. The function block includes the declarations of the utility functions that the actors will call. For example, the StringToInt actor requires a stringToInt function support. The variable block includes the declarations of variables that the actors will use. For example, all output ports of a composite actor must correspond to an output variable, and the data of the input port that would be used in the expression actor needs to be saved with intermediate variables. The fire block includes the declarations of fire functions that the actors will use. Each actor needs a piece of corresponding calculation code. For example, the AddSubtract actor that adds the data of the addition port and subtracts the data of the subtraction port needs the expression statement.

TABLE I
DESCRIPTION OF FIVE REPLACEMENT MARKS

Mark	Content	Value
$type\_$	Input/output port name	Type of the port
$out\_$	Output port name	Actor name + Name of the port
$in\_$	Input port name	Actor name + Name of the port
$para\_$	Parameter name of actor	Value of the parameter
$defaultValue\_$	Input/output port name	Default value of the port type

TABLE II
DESCRIPTION OF TWO REPLACEMENT SYMBOLS

symbol	function	description
${ }$	To specify the replacement identifier	It can only contain one replacement content, such as in_input.
$\\ //$	Repeat the internal text content n times (n represents the number of the data sources for the inport)	There can be other text inside. It can support user-defined ports.

TABLE III
PARSED SYMBOL LIST

type	symbol
bracket	( )
comma	,
unary operators	+ - ~ !
binary operators	. + - * / % ≪ ≫ > ≥ < ≤ == ≠ & — ^ && ‖
ternary operators	?:

Within the template, there may be some contents in the variable declaration and fire function that need to be replaced with symbols. The description of the five replacement marks in the variable declaration and fire function is shown in Table I. In Table I, combining the first and second columns together is the complete mark that needs to be replaced and the third column shows what the mark will be replaced with. For example, *type_output* for a StringToInt actor will be replaced with *int*. The description of the two replacement symbols in the fire function is shown in Table II and these two symbols identify the marks to be replaced.

For some more complex actors, complex expressions should be supported in the template. For example, the temporary variable contained in the parameter of the Expression actor needs to be declared in advance and replaced in the expression. The expression actor also supports complex operations and structures. Therefore, we implement a parser, which could parse the expression into a computation tree. The following symbols are parsed in Table III, where the bracket, comma, unary operators, binary operators, and ternary operators are supported. Then, we will infer the type of the expression result and convert the type of operations in case of inconsistency. For example, the type of *string* + *int* should be converted to *stringAdd*(*string*,*intToString*(*int*)) so that the output data are with the type of *string*. For the structure operations, it is necessary to make the structure support the addition operation. Thus, we recursively solve the computation tree from the bottom to up. For a subtree, the type of the subtree should be got in the light of its operator, and the type conversion function should be added if necessary. If encounter an operation of a structure, an operator overload function should be added to the code. Moreover, considering that the structure may be nested, we also need to declare the operator overloaded function for the internal structure. Finally, the expression could be regenerated based on the computation tree.

TABLE IV
SAMPLE OF TEMPLATE CODE

template of AddSubtract	
header	Null
function	Null
variable	${$ $type\_output$ $}$ ${$ out_output $}$ $= 0;$
fire	${$ out_output $}$ $=$ ${$ out_output $}$ $\\ +$ ${$ in_plus $}$ $//$ $\\ -$ ${$ in_minus $}$ $//$;
template of StringToInt	
header	#include <stdlib.h>

function	int stringToInt(char* str){
	return atoi(str);
	}

variable	${$ $type\_output$ $}$ ${$ out_output $}$ $= 0;$
fire	${$ out_output $}$ $=$ $stringToInt$(${$ in_input $}$);

Then, we can initialize the template for each actor. For example, Table IV shows the synthesized code blocks of the AddSubtract actor and StringToInt actor. We can see that the AddSubtract actor needs a variable block to save the result of calculation and fire block to perform calculation. The StringToInt actor needs the header block and utility function block to support the conversion from string to int. It is precisely because our template supports custom utility functions, so the ability of the template is relatively strong. Users only need to understand the functions of the newly added actors, they can easily create new templates.

C. Sequential Code Assembly

In order to synthesize executable and well-structured code, we need to organize the code of each actor based on the result of schedule analysis. Algorithm 4 shows the workflow of code assembly. The input of Algorithm 4 is a complete model of all composite actor models processed by Algorithm 3. The output is the entire code generated from the complete model. And the final synthesized code is mainly composed of four parts: 1) head files; 2) utility functions; 3) global variables; and 4) functions. These four parts are all obtained when traversing the model recursively and would be deduplicated.

Because the Dataflow model supports hierarchical modeling, we encapsulate each composite actor into a function. The input ports are used as input parameters while the output ports are used as output parameters. For example, an actor named Com1 with an input port (int type) and an output port (double type) will be encapsulated with the following

Algorithm 4 Sequential Code Assembly

Input: *model*: the hierarchical Dataflow model
Output: Sequential code of entire model

```
1: var headFiles = {}
2: var utilityFunctions = {}
3: var globalVariables = {}
4: var functionCodes = {}
5: Function getComActFunCode(subModel)
6:     var globalVar= ""
7:     var function= ""
8:     if isNotFSMModel(subModel) then
9:         order = executionOrderAnalysis(subModel)
10:        schedule = logicRelationshipAnalysis(order)
11:        for all actor in schedule do
12:            if isCompositeActor(actor) then
13:                g,f = getComActFunCode(actor)
14:                globalVariables += g
15:                functionCodes += f
16:            h,u,g,f = genCodeByTemplate(actor)
17:            headFiles += h
18:            utilityFunctions += u
19:            globalVar += g
20:            function += f
21:    else
22:        g,f = genFSMModel(subModel)
23:        globalVariables += g
24:        functionCodes += f
25:    return globalVar, function
26: g,f = getComActFunCode(model)
27: globalVariables += g
28: functionCodes += f
29: var code = ""
30: code += headFileDeduplicate(headFiles)
31: code += utilityFuncDeduplicate(utilityFunctions)
32: for all globalVariable in the globalVariables do
33:     code += globalVariable
34: for all functionCode in the functionCodes do
35:     code += functionCode
36: code += genMainLoopFunc()
37: return code
```

function header: *void FuncCom1*(*int input, double* output*). The getComActFunCode function takes a hierarchical model as input (a composite actor is also regarded as a hierarchical model) and generates the entire function of the hierarchical model and the required global variables as output. If this hierarchical model is not a finite-state machine, we use the previously calculated scheduling sequence to generate the header files, function functions, global variables and fire function of each actor through the template, as presented in lines 9–20. If a composite actor is traversed in this process, we have to recursively generate the function corresponding to the composite actor, as presented in lines 12–15. In addition, we also add the support of the refined finite-state machine. The composite actor of a state machine is also wrapped as a function, as presented in lines 22–24. In the function, we use a static variable to save the current state, switch-case statements to determine the current state, and if-else statements to express condition judgments on the transition between states.

After traversing the entire model, we need to deduplicate the collected header files and utility functions in lines 30–1, because the same header file or utility function only needs to be declared once in the code.

Then, we translate the global parameters and structures of the model into variables and structure types, respectively, in lines 32–35. Finally, we declare the main function and call the mainLoop function to trigger the executable code in line 36.

IV. Implementation

The *DFSynth* is implemented in the C++ language, with 14 218 lines of code. It reads model files as input and generates the executable C and Java code. It is implemented as an external plugin.

Before the schedule analysis, we need to parse the model file. As the model of the Ptolemy-II project is saved as an XML file, and we use the TinyXML library to parse the model file and load the global parameters, actors, and connection relationships among ports. A class, named Actor, is implemented to store the information of each actor, which defines the port information of the actor. This information is used in the code synthesis to name variables. In order to analyze the scheduling order, we implement the topological sorting algorithm to calculate the scheduling order of the actors and store the results in a "map" data structure. Then, we implement Algorithm 2 to mark flags to the actors and merge the positions of the branch actors by Algorithm 3.

For the template definition, different template files are initialized for different actors. Moreover, different actors can be implemented into different C++ classes based on the Actor class. The head files set and the utility functions set are collected, and the result variable and fire code are synthesized according to the scheduling order. If there is no variable and fire code in the template, we need to call the corresponding script to generate a template code dynamically.

Finally, we generate functions for each composite actor and combine the header files set, utility functions set, variables, and functions corresponding to each composite actor into a complete executable code as described in Algorithm 4. It is worth mentioning that because Java does not support the function parameter passing by reference, in the Java code synthesis, the return value of a composite actor is passed through an array. At the end of the code is the main function, which determines the number of times the model is executed based on the duration of the model and the sampling time of the trigger clock.

Furthermore, if we want to add code synthesis support for a new actor, we just need to implement the template file corresponding to the actor. If the template cannot support the features, then implement a targeted script file to generate the template dynamically.

TABLE V
COMPARISON OF FUNCTION AND SUPPORT ABILITY

Modeling Feature	Ptolemy-II C Generator	Ptolemy-II Java Generator	*DFSynth*
Basic actors	△ (23)	△ (19)	◯ (40)
Logic control actor	△ (1)	× (0)	◯ (3)
Structure	×	×	◯
Type conversion	×	×	◯
Global variable	×	×	◯

◯ represents fully supports, △ represents does not fully support, × represents does not support.

V. EVALUATIONS

For evaluation, we compare *DFSynth* with the existing C code generator and Java code generator of Ptolemy-II [9]. So far, we haven't seen the latest work about Ptolemy-II code generator after its C and Java code generator, so we have done a comparative experiment with Simulink's code generator. We construct the experiments to answer the following three research questions.

1) *RQ1 (Capability of DFSynth):* Can it support more actors than the existing code generators of the Dataflow model?
2) *RQ2 (Effectiveness of DFSynth):* Can it generate more efficient code compared to the code generators of the existing design platforms, such as Ptolemy and Simulink?
3) *RQ3 (Expressiveness of DFSynth):* Can it generate more structural code than the existing code generators of the Dataflow model?

A. Can DFSynth Support More Actors?

We selected several frequently and commonly used modeling features. They are important in system modeling, and the support for each code generator is shown in Table V. From the last column of Table V, we can see that *DFSynth* supports the code synthesis of more modeling actors and features.

From the second and third columns of Table V, we can see that the existing code generators in Ptolemy-II cannot completely support those basic actors, such as SinWave actor and StringToInt actor. According to our statistics, Ptolemy-II's C and Java code generators support 23 and 19 basic actors, respectively, and we can currently support 40. Realizing the code generation of new actors in Ptolemy-II is a complicated matter, and it requires adding a lot of code to the Ptolemy-II project. DFSynth can quickly implement actor code generation support by simply making actor templates. The logic control actors, such as BooleanSwitch actor and Switch actor are the basis for the modeling of software systems, it is impossible to model a real system without those logic control actors. The code generator of Ptolemy-II does not have the ability of model scheduling analysis, so its C code generator only supports the two-branch actor BooleanSwitch. Its Java code generator does not support any branch actors, while DFSynth can support multibranch actors, switch, and data selection actor Select.

Some advanced features, such as structure, are used to describe structural data. Code synthesis for the structure is also very complex. First, structures can be nested and all types of elements in the entire structure must be inferred. Second, when it comes to the operation of the structure, such as addition, comparison, etc., it is necessary to generate operator overload function for the structure.

Type conversion is used to facilitate the calculation. For example, the Dataflow model supports the addition of a string and a number, and the number will be converted to a string type automatically. The data type would be calculated bottom-up according to the computation tree of the expression. Once the operation of different data types is found, we must add a conversion function, such as the intToString function.

The global variable is used to facilitate the modeling, which can be used directly in expressions. From Table V, we could see that *DFSynth* could support those modeling actors and features well.

We read the source code of the Ptolemy-II code generator and analyzed the reason why they currently do not support structure, type conversion, and global variables. That is, their main method is actor-based code generation. There is a lack of collation of model context information in the generation process. So, it is difficult to generate the code outside of the actor but closely related to the actor, such as the global variable and struct.

For more details, we use a thermostat model [29] for further illustration, as presented in Fig. 4. This model determines heating or cooling based on the input temperature, and it contains many advanced modeling features that can not be dealt with the existing code generators, such as the state machine, structures ("Thr: {High = 5, Low = 0}"), global variables ("heatingRate:10" "coolingRate:-1"$\cdots$), expressions ("in>Thr.Low" "str+val"), and branch actor ("BooleanSwitch").

The code synthesized by *DFSynth* is presented in Listing 1. Lines 7–12 implement the conversion of data types and string processing in Expression2 actor, and the structure is implemented in lines 13–16. Lines 17–21 show the declaration of global variables, and lines 22–39 implement the code synthesis of the refined state machine. Lines 41–46 show the declaration of the actors' result variables and intermediate variables, where the data type of the expression actor's result is resolved. Line 48 indicates the call of the state machine. Lines 51–57 implement the branch-sensitive actor BooleanSwitch with the if-else statement block.

B. Can DFSynth Generate More Efficient Code?

Because the existing code generators of Ptolemy-II do not support the synthesis of advanced features, we cannot use the above complex example in Fig. 4. We adopt the benchmark models provided by the existing C code generator of Ptolemy-II and three more complex Dataflow models built by ourselves for comparison. These models include state machine,

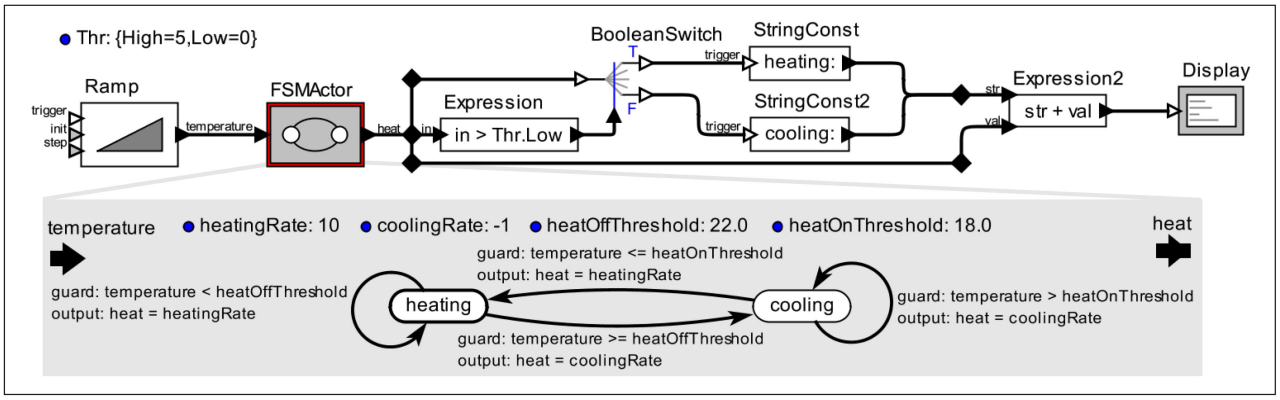


Fig. 4. Dataflow model of thermostat control, which includes many advanced features, such as composite actor, state machine, and BooleanSwitch.

complex mathematical calculation, complex branch expression, etc., and are sufficiently representative. Furthermore, because Simulink cannot directly read Ptolemy II's model, we need to manually build the Stateflow model of the corresponding Dataflow model. The benchmark Dataflow models and the manually transferred Stateflow models can be downloaded from the footnoted Website and a brief introduction is as follows.

1) ClockRamp is a model that uses a Clock actor to trigger an arithmetic progression actor named Ramp, which outputs the last output value plus a step value each time when triggered.
2) HelloWorld is a model that just output a "HelloWorld."
3) HeteroMK is a state machine model, which determines what value the output will be used, the original value of the input, or the opposite value of the input.
4) Math is a model that computes the sum of the first n natural numbers using the formula: $Sum(1 \ldots n) = n * (n + 1)/2$.
5) PiSquare is a model that calculates the square of π by Riemann's Zeta function.
6) ScaleCFlat is a model that scales the output of a Ramp actor using the Scale actor.
7) LeakyRelu is a model that can show the LeakyRelu function, an activation function commonly used in the field of deep learning.
8) Piecewise function is a model that can show a piecewise function with three function segments controlled by two BooleanSwitch actors.
9) Complex branches is a more complex model with branches nested and crossed.

For the comparison of the code generators of Ptolemy-II, Simulink, and *DFSynth*, we use five metrics: 1) the lines of code; 2) the number of files; 3) the execution time of the synthesized code; 4) the consistency between the code execution and model simulation; and 5) the time (ms) of the code generation process. We compiled and ran the code in the same environment (window10 ×64, Cygwin64 Terminal, gcc and javac compiler). To avoid the randomness, we executed the code for 10 000 times, and got the average execution time. The unit was milliseconds. The Ptoelemy-II Java code generator cannot synthesize the code for the model with state machine and branch actor, such as HeteroMK, LeakyRelu, piecewise function, and complex branches. The detailed results are presented in Table VI.

As for the lines of the synthesized code, *DFSynth* outperforms Ptolemy and Simulink, with an average size reduction of 61.9%, 99.7%, and 81.4% for Simulink-C, Ptolemy-C , and Ptolemy-Java, respectively. This is because we use the compact variable passing approach to generate the code. While the original Ptolemy C code generator uses the event passing method to generate code. It generates a .h file and a .c file for each actor and generates a lot of custom classes, such as HashMap, PriorityQueue, etc., so the code size is huge (around 10 000 LoCs) for a small model. Although the code synthesized by the original Ptolemy Java code generator is relatively short, it synthesizes all the type conversion functions which results in plenty of redundancy. Furthermore, we only count the main control logic code synthesized by Simulink Coder and do not include the synthesized libraries or utilities, which are more complex with an extra $\sim$ 2000 LoCs. We found that it had a lot of redundant runtime-related codes.

In terms of the execution time, the code synthesized by *DFSynth* is the shortest, and the running time is decreased by 22.7%, 76.2%, and 56.8% for Simulink-C, Ptolemy-C, and Ptolemy-Java, respectively. This is because the synthesized code is the most compact and we do not need to execute some redundant code. As we can see in Table VI, the number of lines of C code and Java code we generated are almost the same, but the running time of Java code is relatively long, which is caused by the extra overhead of the javavm. Although the code generated by *DFSynth* has advantages in the number of lines of code and structure, because the hierarchical composite actor is expressed as a function with input and output parameters, frequent function calls during the code execution process will bring some additional expenses.

Furthermore, for the correctness of the results, we also compare the results of the Dataflow model and Stateflow model simulation with the results of the synthesized code execution, with 10 000 random runs. We found that the results of the code synthesized by *DFSynth* are always the same with the original model simulation result, but there is an inconsistency between the result of the simulation and the result of the code synthesized by the existing code generator. Even there is a

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define bool int
#define false 0
#define true 1
char* intToString(int n){
    char* ret = (char*)malloc(12);
    sprintf(ret, "%d", n);
    return ret;
}
...
struct Struct_Thr{
    int High;
    int Low;
};
struct Struct_Thr Thr = {5,0};
int coolingRate = -1;
float heatOffThreshold = 22.0;
float heatOnThreshold = 18.0;
int heatingRate = 10;
void FuncFSMActor(int temperature,int* heat){
    static int FSMActor_nextstate;
    enum FSMActor_state{heating,cooling};
    switch(FSMActor_nextstate){
    case heating:
        if(temperature < heatOffThreshold){
            *heat = heatingRate;
            FSMActor_nextstate = heating;
        }else if(temperature >= heatOffThreshold){
            *heat = coolingRate;
            FSMActor_nextstate = cooling;
        }
        break;
    case cooling:
        ...
        break;
    }
}
void mainLoop(){
    static int Ramp_output = 15 - 1;
    int FSMActor_heat = 0;
    bool Expression_output = false;
    char* StringConst_output = "heating:";
    char* StringConst2_output = "cooling:";
    char* Expression2_output;
    Ramp_output += 1;
    FuncFSMActor(Ramp_output,&FSMActor_heat);
    int Expression_in = FSMActor_heat;
    Expression_output = Expression_in > Thr.Low;
    if(Expression_output){
        char* Expression2_str=stringCopy(
     StringConst_output);
        int Expression2_val = FSMActor_heat;
        Expression2_output = stringAdd(Expression2_str,
     intToString(Expression2_val));
    }else{
        ...
    }
    printf("%s\n", Expression2_output);
}
#define LOOP_COUNT 10
int main(){
    int i;
    for(i = 0;i<LOOP_COUNT;i++){mainLoop();}
    return 0;
}
```

Listing 1. Code synthesized by *DFSynth*.

runtime error in the code of complex branches generated by Ptolemy-C. The results of *DFSynth* are also optimal in terms of the time cost of the code generation process. Base on these various models, combining with the statistics above, it is reasonable to conclude that we can accomplish the same tasks with less code size and less execution time, thus generating a more efficient code.

C. Can DFSynth Include More Structure Information?

We apply *DFSynth* on a real case from Huawei to demonstrate the effectiveness in information preservation. The model is built for the CPU resource allocation scheduling. Due to space limitations and confidentiality agreements, only a part of the model about CPU priority adjustment is shown in Fig. 5. The complete synthesized code for Fig. 5 can be downloaded from the Website of *DFSynth* presented in footnote 1. Because the Java code and C code synthesized by *DFSynth* are similar, only the C code is shown in Listing 4. We select the same number of lines of code snippets synthesized by the three code generators for display, as shown in Listings 2–4. In addition,

TABLE VI
COMPARISON OF PTOLEMY, SIMULINK, AND *DFSynth*

Model	Generator	Lines	Files	Run Time	Consistency	Gen Time
ClockRamp	Simulink-C	67	2	46	yes	692
	Ptolemy-C	11063	46	781	yes	112
	Ptolemy-Java	116	2	453	yes	46
	DFSynth-C	**21**	**1**	**30**	yes	3
	DFSynth-Java	**16**	**1**	**390**	yes	3
HelloWorld	Simulink-C	45	2	9	yes	709
	Ptolemy-C	11000	46	172	yes	104
	Ptolemy-Java	109	2	62	yes	39
	DFSynth-C	**20**	**1**	**5**	yes	4
	DFSynth-Java	**15**	**1**	**60**	yes	3
HeteroMK	Simulink-C	135	2	52	yes	1364
	Ptolemy- C	12423	69	1923	no	359
	Ptolemy-Java	-	-	-	-	-
	DFSynth-C	**55**	**1**	**32**	yes	7
	DFSynth-Java	**58**	**1**	**405**	yes	7
Math	Simulink-C	72	2	58	yes	697
	Ptolemy- C	11797	56	1890	yes	151
	Ptolemy-Java	180	2	468	yes	91
	DFSynth-C	**30**	**1**	**46**	yes	4
	DFSynth-Java	**25**	**1**	**405**	yes	4
PiSquare	Simulink-C	70	2	60422	yes	808
	Ptolemy-C	11520	52	168921	yes	141
	Ptolemy-Java	153	2	60814	yes	76
	DFSynth-C	**27**	**1**	**46703**	yes	5
	DFSynth-Java	**22**	**1**	**51151**	yes	4
ScaleCFlat	Simulink-C	68	2	16	yes	698
	Ptolemy-C	9873	43	328	yes	124
	Ptolemy-Java	119	2	218	yes	61
	DFSynth-C	**23**	**1**	**12**	yes	4
	DFSynth-Java	**18**	**1**	**188**	yes	5
LeakyRelu	Simulink-C	46	2	24	yes	1163
	Ptolemy-C	11556	52	343	yes	196
	Ptolemy-Java	-	-	-	-	-
	DFSynth-C	**18**	**1**	**20**	yes	3
	DFSynth-Java	**13**	**1**	**244**	yes	3
Piecewise function	Simulink-C	53	2	26	yes	1046
	Ptolemy-C	12175	60	362	yes	290
	Ptolemy-Java	-	-	-	-	-
	DFSynth-C	**20**	**1**	**21**	yes	4
	DFSynth-Java	**15**	**1**	**296**	yes	5
Complex branches	Simulink-C	136	2	32	yes	1786
	Ptolemy-C	13940	87	Error	-	340
	Ptolemy-Java	-	-	-	-	-
	DFSynth-C	**49**	**1**	**26**	yes	6
	DFSynth-Java	**44**	**1**	**379**	yes	7
Average	Simulink-C	76	2	7578	yes	995
	Ptolemy-C	11705	56	24690	no	201
	Ptolemy-Java	135	2	15386	yes	62
	DFSynth-C	**29**	**1**	**5856**	yes	4
	DFSynth-Java	**25**	**1**	**6639**	yes	4

```
if((*(multiply->hasToken))((structIOPort*)multiply,0)){
    Sample_MultiplyDivide__result=convert_Int_Int((*(multiply->get))((structIOPort*)multiply,0)->payload.Int);}
if((*(multiply->hasToken))((structIOPort*)multiply,1)){
    Sample_MultiplyDivide__result=multiply_Int_Int(Sample_MultiplyDivide__result,(*(multiply->get))((structIOPort*)
     multiply,1)->payload.Int);
}
if((*(divide->hasToken))((structIOPort*)divide,0)){
Sample_MultiplyDivide__result=divide_Int_Int(Sample_MultiplyDivide__result,(*(divide->get))((structIOPort*)divide,0)->
     payload.Int);
}
(*(output->send))((structIOPort*)output,0,Int_new(Sample_MultiplyDivide__result));
```

Listing 2. Code synthesized by Ptolemy-II-C and covers on actor partially.

```
if (jsf_simulink_M->Timing.t[0] < jsf_simulink_P.Ramp_start)
    jsf_simulink_B.Step = jsf_simulink_P.Step_Y0;
else
    jsf_simulink_B.Step = jsf_simulink_P.Ramp_slope;
jsf_simulink_B.Clock = jsf_simulink_M->Timing.t[0];
jsf_simulink_B.Sum = jsf_simulink_B.Clock - jsf_simulink_P.Ramp_start;
jsf_simulink_B.Product = jsf_simulink_B.Step * jsf_simulink_B.Sum;
jsf_simulink_B.Output = jsf_simulink_B.Product + jsf_simulink_P.Ramp_InitialOutput;
jsf_simulink_B.Divide = jsf_simulink_B.Output * jsf_simulink_B.Output / jsf_simulink_P.Constant_Value;
```

Listing 3. Code synthesized by the Simulink coder for the corresponding Stateflow model and covers two actors partially.

```
int Const_output = 2;
static int Ramp_output = 0 - 5;
int MultiplyDivide_output = 1;
int PrioritySC_out = 0;
int BooleanMultiplexor2_output = 0;
Ramp_output += 5;
MultiplyDivide_output = MultiplyDivide_output * Ramp_output * Ramp_output / Const_output;
FuncPrioritySC(MultiplyDivide_output,Const_output,&PrioritySC_out);
BooleanMultiplexor2_output = PrioritySC_out;
```

Listing 4. Code synthesized by *DFSynth* for the Dataflow model and covers five actors.

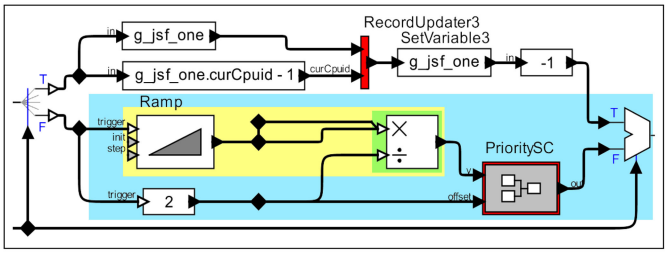


Fig. 5. Dataflow model for CPU priority adjustment in Huawei. With the same lines of the synthesized code in Listings 2–4, the code synthesized by Ptolemy-II-C only covers one actor partially labeled with green, the code synthesized by Simulink Coder covers two actors (originated from the corresponding manually transferred Stateflow model) partially labeled with yellow, and the code synthesized by *DFSynth* covers five complex actors labeled with blue.

for the original C code generator of Ptolemy-II and Simulink, we removed the useless symbols, comments, and libraries.

The C code shown in Listing 2 is part of the execution code of the MultiplyDivide actor synthesized by the Ptolemy-II-C, as highlighted in green in Fig. 5. When a result is calculated, it sends the result to the input port of the subsequent actor through the output port of the MultiplyDivide actor. The synthesized code contains Token handlers, event handlers, data dispatch functions, and many other functions that handle event dispatching. The C code shown in Listing 3 is the execution code for the two actors (mapped from the corresponding stateflow model) synthesized by the Simulink coder, as highlighted in yellow in Fig. 5. In contrast, the C code shown in Listing 4

TABLE VII
EVALUATION ON THE INDUSTRIAL MODEL

Generator	Lines	Files	Runtime	Consistency	GenTime
Simulink-C	524	2	564	yes	3482
Ptolemy-C	14422+	100+	-	-	-
Ptolemy-Java	714+	2	-	-	-
DFSynth-C	**223**	**1**	**436**	yes	23
DFSynth-Java	**246**	**1**	**930**	yes	22

synthesized by *DFSynth* contains five actors of this model, as highlighted in blue in Fig. 5. It abstracts the composite actor completely into a function and execute the composite actor through parameter passing and function call.

Table VII shows the evaluation of the whole synthesized code. Because the two code generators in Ptolemy-II do not support many features, such as global variables, structures, etc., they cannot synthesize complete code and their code lines and file numbers are only partial results that can be counted, and these incomplete code cannot be compiled and run at all. Because the Simulink coder only supports the Stateflow model, we build it based on the above Dataflow model and collect LoCs of the main logic.

When the Dataflow model is synthesized to the control-flow code, the synthesized code could express more actors with limited lines, while more logical and structure information would be preserved. If a branch control actor exists in a Dataflow-based model, the execution logic of other actors will be affected. For a branch control actor with two branches, there

could only be two types of execution logic, that is, some actors should be executed either after the first branch of or after the second branch. Although the complex schedule analysis from Dataflow to control-flow can be avoided while generating the code for event transfer execution, plenty of code related to event handling will be synthesized and the logic of the code will not be clear. Furthermore, we could get the synthesized code with clearer structure when the composite actors and refined state machine are wrapped as functions. If we generate code by event passing, as for every composite actor, we should also generate code for processing event queue. In contrast, only one line of code is needed to call the function based on the control-flow approach of *DFSynth*.

VI. Lesson Learned

In the practice of code generation of Dataflow model, we learned some valuable experiences as follows.

In real model design practice, in addition to modeling and simulation, engineers have a great need for code generation. Modeling and simulation are important functions of many modeling tools. They can find performance problems and functional problems in some predesigns. Furthermore, code generation is also important. On the one hand, it is hard to read the code written by others due to the different styles of code, and the gap between the model and the implemented code brings a challenge to the development lifecycle, such as software maintenance and model-based testing. On the other hand, based on the comparison between the code written by engineers and the code generated by us, we find that engineers like to use a complex expression to directly calculate the results that can be processed by combining multiple actors. The optimization of data flow based on the model may be our further research content.

We can get the generated code with more logical information when the model is converted from Dataflow to control-flow. If a branch control actor exists in a Dataflow-based model, the execution logic of the actor will be affected in the model. Obviously, as for an actor with two branches, there could only be two types of execution logics, that is, some actors should be executed either after the first branch of the branch actor or after the second branch. In this case, it is a more reasonable choice to generate blocks of if-else statements from these actors. Although the complex conversion from Dataflow to control-flow can be avoided while generating the code for event transfer execution, plenty of code related to event handling will be generated and the logic of the code will not be clear. From our experiments, the code generated in this way is clearly structured, and the number of lines is relatively short. However, the model with logic control actors may have some branches that will not be executed at all, and these branches will also generate the code that will not be executed. Therefore, it is our further research work to remove such codes through data flow analysis.

VII. Conclusion

Although the Dataflow model and its corresponding platforms are widely used for system modeling and simulation, the code synthesis ability is quite limited. In this article, we tried to bridge the gap between the simulation and synthesis of the Dataflow model via a strengthened code generator *DFSynth*. We adopted the schedule analysis to accomplish the transformation of Dataflow actors execution order to control flow code branches, encapsulated hieratical composite actors, and state machines into functions, and defined code templates for basic actors, which makes the synthesized code short and structured. Compared to the existing code generators of Ptolemy and Simulink, *DFSynth* could include more logic and structural information with less size of code and execution time. Our future work includes bridging the code synthesis gap between the Dataflow model and Stateflow model.

References

[1] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558607026icle/pii/B978155860702650048X

[2] E. A. Lee and Y. Xiong, "A behavioral type system and its application in Ptolemy II," *Formal Aspects Comput.*, vol. 16, no. 3, pp. 210–237, 2004.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II)," Dept. Electr. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Rep. UCB/EECS-2008-28, 2008.

[4] Y. Jiang *et al.*, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, Feb. 2015.

[5] P. Baldwin, S. Kohli, X. Liu, Y. Zhao, and E. A. Lee, "Modeling of sensor nets in Ptolemy II," in *Proc. 3rd Int. Symp. Inf. Process. Sens. Netw.*, 2004, pp. 359–368.

[6] C. Brooks, E. A. Lee, and S. Tripakis, "Exploring models of computation with Ptolemy II," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardw. Softw. Codesign Syst. Synth. (CODES + ISSS)*, 2010, pp. 331–332.

[7] J. Eker *et al.*, "Taming heterogeneity—The Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[8] H. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in Ptolemy II," in *Proc. Amer. Control Conf.*, vol. 6, 2001, pp. 4984–4985.

[9] G. Zhou, M.-K. Leung, and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," in *Proc. Int. Conf. Embedded Softw. Syst.*, 2007, pp. 193–206.

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

[11] 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 *Proc. 1st Int. Workshop Softw. Eng. Res. Pract. Internet Things*, 2019, pp. 25–32.

[12] K. Jahed and J. Dingel, "Enabling model-driven software development tools for the Internet of Things," in *Proc. 11th Int. Workshop Model. Softw. Eng.*, 2019, pp. 93–99.

[13] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf, "A model-driven workflow for distributed microservice development," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, pp. 1260–1262.

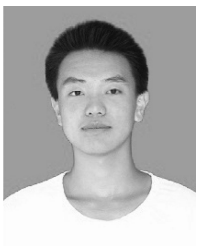
[14] F. Pasic, "Model-driven development of condition monitoring software," in *Proc. 21st ACM/IEEE Int. Conf. Model Driven Eng. Lang. Syst. Companion*, 2018, pp. 162–167.

[15] Y. Jiang *et al.*, "Safety-assured model-driven design of the multifunction vehicle bus controller," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 10, pp. 3320–3333, Oct. 2018.

[16] Y. Jiang *et al.*, "Dependable model-driven development of CPS: From stateflow simulation to verified implementation," *ACM Trans. Cyber Phys. Syst.*, vol. 3, no. 1, p. 12, 2018.

[17] Y. Jiang *et al.*, "Tsmart-GalsBlock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 711–714.

[18] Y. Jiang *et al.*, "Design of mixed synchronous/asynchronous systems with multiple clocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2220–2232, Aug. 2015.
[19] H. Zhang, Y. Jiang, H. Liu, H. Zhang, M. Gu, and J. Sun, "Model driven design of heterogeneous synchronous embedded systems," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2016, pp. 774–779.
[20] G. Berry, "SCADE: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Dordrecht, The Netherlands: Springer, 2007, pp. 19–33.
[21] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *J. Circuits Syst. Comput.*, vol. 12, no. 3, pp. 261–303, 2003.
[22] F. Balarin *et al.*, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. New York, NY, USA: Springer, 1997.
[23] G. Berry, "Circuit design and verication with Esterel V7 and Esterel studio," in *Proc. IEEE Int. High Level Design Validation Test Workshop*, 2007, pp. 133–136.
[24] H. Kim, A. Wasicek, and E. A. Lee, "An integrated simulation tool for computer architecture and cyber-physical systems," in *Proc. Int. Workshop Design Model. Eval. Cyber Phys. Syst.*, 2017, pp. 83–93.
[25] M. Bagheri *et al.*, "Coordinated actor model of self-adaptive track-based traffic control systems," *J. Syst. Softw.*, vol. 143, pp. 116–139, Sep. 2018.
[26] M. Látková, M. Baherník, P. Braciník, and M. Höger, "Modelling of a dynamic cooperation between a PV array and DC boost converter," in *Proc. 5th Int. Youth Conf. Energy (IYCE)*, 2015, pp. 1–7.
[27] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 3, pp. 1–26, 2013.
[28] T. Miyazaki and E. A. Lee, "Code generation by using integer-controlled dataflow graph," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, vol. 1, 1997, pp. 703–706.
[29] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II*, vol. 1. Berkeley, CA, USA: Ptolemy. Org., 2014,



Zhuo Su received the B.S. degree in software engineering from Northeastern University, Shenyang, China, in 2018. He is currently pursuing the Ph.D. degree in software engineering with Tsinghua University, Beijing, China.

His research interests are in the areas of model-driven development and embedded software engineering.

Dongyan Wang received the B.S. degree in computer science and technology from Shandong University, Weihai, China, in 2018. She is currently pursuing the M.S. degree in computer science with Peking University, Beijing, China.

Her research interests are in the areas of computer architecture and digital image processing.

Yixiao Yang received the B.S. degree in software engineering from Nanjing University, Nanjing, China, in 2014, and the Ph.D. degree in software engineering from Tsinghua University, Beijing, China, where he is currently pursuing the Postdoctoral degree with the School of Software.

His research interests include code completion, test case generation, model-driven design, and their applications to industry.

Yu Jiang received the B.S. degree in software engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2010, and the Ph.D. degree in computer science from Tsinghua University, Beijing, in 2015.

He was a Postdoctoral Researcher with the Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2016. He is currently an Associate Professor with Tsinghua University. His research interests include domain-specific modeling, formal computation model, formal verification, and their applications in embedded systems.

Wanli Chang (Member, IEEE) received the bachelorŠs degree (First Class Hons.) from Nanyang Technological University, Singapore, and the Ph.D. degree in electrical and computer engineering from the Technical University of Munich (TUM), Germany, in 2017, and won the Departmental Best Dissertation Award.

He is an Assistant Professor with the Department of Computer Science, University of York, York, U.K. His research spans across all aspects of real-time and embedded systems.

Dr. Chang is serving in the TPC of premium conferences on embedded and real-time systems, as well as design automation, including DAC, ICCAD, DATE, RTSS, EMSOFT, CODES+ISSS, and LCTES.

Liming Fang (Member, IEEE) received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2012.

He has been a Postdoctoral Researcher of Information Security with the City University of Hong Kong, Hong Kong. He is an Associate Professor with the School of Computer Science, Nanjing University of Aeronautics and Astronautics. He is currently a Visiting Scholar with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, USA. His current research interests include cryptography and information security. His recent work has focused on the topics of public-key encryption with keyword search, proxy re-encryption, identity-based encryption, and techniques for resistance to CCA attacks.

Wen Li received the B.S. degree in computer science and technology from Xidian University, Xi'an, China, in 2006, and the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2011.

He is currently working with Huawei, Hangzhou. His research focuses on modeling, simulation, and automatic code generation in the intelligent driving field.

Jiaguang Sun received the B.S. degree in automation science from Tsinghua University, Beijing, China, in 1970.

He is currently a Professor with Tsinghua University. He is currently the Director of the School of Information Science and Technology and the School of Software, Tsinghua University. He is dedicated in teaching and research and development activities in computer graphics, computer-aided design, formal verification of software, and system architecture.