Hindawi Publishing Corporation
Discrete Dynamics in Nature and Society
Volume 2015, Article ID 580328, 11 pages
http://dx.doi.org/10.1155/2015/580328

Research Article

Research on the Development Approach for Reusable Model in Parallel Discrete Event Simulation

Jianbo Li,[1] Yiping Yao,[1,2] Wenjie Tang,[2] and Feng Zhu[2]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, China*
[2]*College of Information System and Management, National University of Defense Technology, Changsha 410073, China*

Correspondence should be addressed to Jianbo Li; lijianbo0404@163.com

Received 16 November 2014; Accepted 1 January 2015

Academic Editor: Qingang Xiong

Model reuse is an essential means to meet the demand of model development in complex simulation. An effective approach to realize the model reusability is to establish standard model specification including interface specification and representation specification. By standardizing model's external interfaces, Reusable Component Model Framework (RCMF) achieves the model reusability acting as an interface specification. However, the RCMF model is presently developed just through manual programing. Besides implementing model's business logic, modeler should also ensure the model strictly following the reusable framework, which is very distracting. And there lacks model description information for instructing model reuse or integration. To address these issues, we first explored an XML-based model description file which completed RCMF as the model representation and then proposed a RCMF model development tool—SuKit. Model description file describes a RCMF model and can be used for regenerating a model and instructing model integration. SuKit can generate a skeleton RCMF model together with a model-customized description file with the configured information. Modeler then just needs to concentrate on the model processing logic. The case study indicates that SuKit has good capability of developing RCMF models and the well-formed description file can be used for model reuse and integration.

1. Introduction

To respond quickly to the model development requirement in developing complex simulation applications, model reuse is considered as a promising approach. It improves the development efficiency, guarantees models' reliability, and minimizes engineering efforts as well as resource costs [1–3]. Model reuse has been widely studied in the field of modeling and simulation (M&S). The existing reusable model development approaches can mainly be classified into four categories: (a) reusing based on unified M&S language; (b) reusing based on standard model specification (RBSMS); (c) reusing based on generic model; and (d) reusing based on unified M&S environment [4]. Through establishing standard model specification including interface specification and representation specification, RBSMS greatly realizes model reusability and portability. The interface specification standardizes model interfaces and defines uniform model structure, which makes the model easy to be reused and integrated across different simulation applications. Model representation describes the model in a platform-independent language and can be used to regenerate a platform-specific model according to the specific simulation application.

In our previous work [1], a Reusable Component Model Framework (RCMF) guiding modeler developing reusable model as an interface specification was proposed. RCMF sets six standard interfaces to encapsulate a simulation computational module. RCMF model interacts with simulation application only through the unified external interfaces. RCMF has been successfully applied in several complex analytic simulation applications.

However, the present development approach for developing RCMF model is just writing model code manually. With this development mode, modeler not only needs to implement model processing logic but also needs to ensure the model strictly following the standard reusable framework, which makes modeler cannot focus on the implementation of model function. A similar framework needs to be coded

manually for each model, which is repetitive and boring. And there lacks enough model description information, so that the model user needs a face-to-face assistance from the model developer when integrating models, let alone reusing them.

Motivated by this, we first explored a platform-independent model description file for each RCMF model and then developed a semiautomatic RCMF model development tool—SuKit for modelers.

The model description file further supplements RCMF as a kind of model representation. It is written in XML and describes sufficient information of the RCMF model, such as model's basic information, model members, and model external interfaces. The information described in the description file can be used not only to guide model user integrating the model but also to regenerate a corresponding RCMF model. SuKit provides a graphical interface for modeler to configure a model and then generates a standardized RCMF skeleton model together with a model-customized description file with the configuration information. SuKit makes the reusable model development much easier. Based on the framework code, modeler then just needs to concentrate on the implementation of model processing logic. SuKit supports modeler implementing model processing logic through either importing computational model or implementing it anew. When modeler further edits the model, SuKit implements a six-way synchronization mechanism to synchronize the data among configuration information, model code, and description file. Thanks to the synchronization mechanism, both model header file and description file can be parsed by SuKit and start the model generation. The compiler integrated in SuKit will make the completed model and package the model into a library file. The final products SuKit is providing are a RCMF model in the form of source model or binary model and a corresponding description file. Then the model can be integrated in a simulation application with the instruction from the description file.

The case study of developing a simplified radar model with SuKit indicates that SuKit has good capability of developing the RCMF model and the well-formed description file can be used in model integration and reuse.

Section 2 summarizes the related reusable model development approaches and discusses the reusability realized in RCMF. Section 3 presents the designed model description file. In Section 4, we elaborate the architecture of SuKit and its synchronization mechanism. Section 5 describes the case of developing a radar model with SuKit. Finally, our conclusion and future work will be made in Section 6.

2. Background and Related Work

2.1. Related Reusable Model Development Approaches. RBSMS is an effective means to realize model reusability and there are several model reuse approaches belonging to RBSMS. SISO proposed the Base Object Model (BOM) [5, 6] to reuse the simulation object model in HLA [7]. SMP/SMP2 [8, 9] came from the European Space Agency (ESA), and it was established to reuse model across different applications within any platform supporting SMP/SMI. SMP/SMP2 standardizes the interfaces between simulation models and simulation infrastructure, and the Simulation Model Definition Language (SMDL) is proposed to represent the descriptive SMP model. SEM/MB based on DEVS [10, 11] is also a successful specification for model reuse. It represents the model in DEVS formalization and the DEVS-Bus simulation protocol acts as the interface specification.

Although these approaches can reuse model in certain simulation platform, there still exist some limitations. As the interface specification of BOM is based on HLA-OMT, BOM models run relying on the services from RTI. Models under SEM/MB should also be run on the simulation platform supporting DEVS formalization. Both BOM and SEM/MB lack the technique of transforming the conceptual model into simulation model [4]. As for SMP2, SMP2_Cpp_Mapping [12] implemented the mapping from SMDL metamodel to model code for SMP2, but models under SMP2 still rely on the services from SMI. All these model reuse approaches cannot reuse model across different simulation platforms, as the model relays on services from certain simulation engine.

2.2. Reusable Component Model Framework and Its Reusability. RCMF is proposed by Zhu et al. [1] and has been successfully applied in several complex simulation applications. Considering that models with unified interfaces are easy to be reused and be integrated in applications, RCMF sets six standard service interfaces to encapsulate a simulation computational module. Actually the computational module developed by domain experts can also be used in applications if it is being invoked correctly. But the diversiform interface of the computational module fetters the development of simulation applications, as an application needs to implement special invocation logic for certain computational module. RCMF addressed this issue by encapsulating the computational module with standard interfaces, including state restoring interface, input interface, dynamic data driven interface, business process interface, state getting interface, and output interface.

Different from the above approaches, interfaces specified in RCMF are abstracted from models' execution law in simulation applications. RCMF is independent from specific simulation platform. RCMF model is a platform-independent black box and it interacts with the outside world only through the standard interfaces. Within each model, modeler just needs to concentrate on the implementation of model function. RCMF is applicable for simulation models of different simulation platforms, such as SUPE [13], HLA, POSE [14], and Charm++ [15].

The reusability realized in RCMF mainly reflects in the three aspects. (1) RCMF model is a reusable computational model. It need not care about the underlying works, like Time Keeping, Event Managing, and Scheduling, so that the model can be reused across different simulation platforms. (2) RCMF model can be reused across simulation applications. As a result of the uniform interfaces, RCMF model can be used to assemble applications like a component. As long as we keep the standard interfaces unchanged, models can be reused no matter whether it is upgraded or modified. (3) After being encapsulated with the standard interfaces, the existing computational model can also be reused in

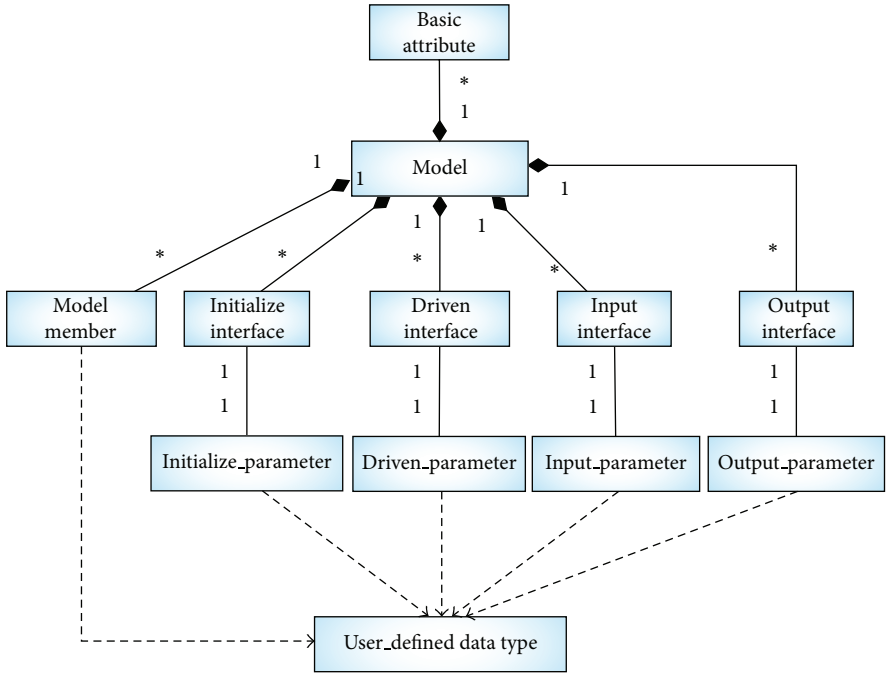

Figure 1: Structure of the description file.

new applications. However, RCMF only specifies model interfaces. A self-contained complete model representation mechanism is needed.

3. The Platform-Independent Model Description File

XML is a widely used markup language. It has lots of good characteristics such as self-description and being data-structured, extensible, platform-independent, and machine-readable. In simulation field, XML also has a wide application. For example, the military simulation scenario is written in XML. SRML [16] and SML [17, 18] are all M&S language based on XML. And SMP2 also designed the XML-based SMDL to describe Catalogue, Assembly, and Schedule. In view of these good properties of XML and its description power, we first customized an XML-based description file for each RCMF model.

For each RCMF model, the external interfaces are unified and the form of each interface is fixed. And we restrict that each interface owns only one parameter, so that a parameter information item in the description file can represent an interface. If modeler designs any model member, the information of each member should also be recorded in the description file. To support the rollback mechanism of optimistic synchronization algorithm, each member needs to set an attribute field which indicates whether the member could be a state variable. The state variable is handled by state restoring interface and state getting interface. The parameter information of these two interfaces does not need to be recorded, as the parameter is always a string in which all the state variables are packed. The string is encoded in state getting interface and decoded in state restoring interface. The business processing interface is fixed for all the models and it also need not be recorded. The type of model members or the interface parameters could be a user-defined data type. To make model user clear about the user-defined data type, it should also be described in the description file. In addition to the above information, the model basic attributes like model name and model function also need to be configured in the description file. Figure 1 is the metamodel of the description file. Different from the interfaces specified in RCMF, we create an *Initialize* interface to fulfill the initialization task of the first invocation of the state restoring interface.

Algorithm 1 gives an instance of the model description file. The description file has a three-layer structure (excepting the description of *User_defined_DataTypes*, it also has a three-layer structure). The first layer is the root node *Model*, and the attribute field of the *Model* node describes the following basic information: *ModelName*, *Namespace*, *Function*, *Invocation_time*, and additional *Instruction*. Children of the *Model* node are organized in the second layer, which is set to group the model information. The first child is the node *ModelMembers*, of which each child node *Member* describes a model member. And in the attribute field of *Member*, one records the member information: *Name*, *DataType*, *VariableType*, *IsStateVariable*, and *Instruction*. The second child of *Model* node is the node *Initialize_parameters* which collects a set of *Initialize* interface parameters. Each *Parameter* node has five attributes: *Name*, *DataType*, *VariableType*, *Corresponding_model_member*, and *Instruction*. The following third, fourth, and fifth child node of *Model* describe the dynamic data driven interface parameters, input parameters, and output parameters, respectively. Parameters of the four interfaces are described with the same rule.

```
<?xml version="1.0" encoding="UTF-8"?>
<Model ModelName="testModel" Namespace="YHSim" Function="" InvocationTime="" Instruction="">
    <ModelMembers>
        <Member Name="m_point" DataType="Point" VariableType="POINTER" IsStateVariable="YES" Instruction=""/>
        <Member Name="m_speed" DataType="Speed" VariableType="POINTER" IsStateVariable="No" Instruction=""/>
        <Member Name="m_result" DataType="Result" VariableType="VECTOR_POINTER_PTR" IsStateVariable="YES"
        Instruction=""/>
    </ModelMembers>
    <Initializing_parameters />
    <Driven_parameters />
    <Input_parameters>
        <Parameter Name="speed" DataType="Speed" VariableType="POINTER" Corresponding_model_member="m_speed"
        Instruction=""/>
        <Parameter Name="point" DataType="Point" VariableType="POINTER" Corresponding_model_member="m_point"
        Instruction=""/>
    </Input_parameters>
    <Output_parameters>
        <Parameter Name="result" DataType="Result" VariableType="VECTOR_POINTER_PTR"
        Corresponding_model_member="m_result" Instruction=""/>
    </Output_parameters>
    <Macros>
        <Macro Name="Pai" Value="3.14"/>
    </Macros>
    <HeaderFiles Files="iostream.h;"/>
    <LibFiles Files="caputationalModule.so;"/>
    <User_defined_dataTypes>
        <User_defined_dataType TypeName="Point" Namespace="YHSim" Type="STRUCT" Instruction="">
            <member Name="x" DataType="double" VariableType="VARIABLE" Instruction=""/>
            <member Name="y" DataType="double" VariableType="VARIABLE" Instruction=""/>
        </User_defined_dataType>
        <User_defined_dataType TypeName="Speed" Namespace="YHSim" Type="STRUCT" Instruction=""/>
        <User_defined_dataType TypeName="Result" Namespace="YHSim" Type="STRUCT" Instruction="nested structure"/>
        <User_defined_dataType TypeName="testEnum" Namespace="YHSim" Type="ENUM" Instruction="">
            <member Name="A" Value="1"/>
            <member Name="B" Value="2"/>
        </User_defined_dataType>
        <User_defined_dataType TypeName="testUnion" Namespace="YHSim" Type="UNION" Instruction="">
            <member Name="" DataType=""/>
        </User_defined_dataType>
    </User_defined_dataTypes>
</Model>
```

Algorithm 1: An instance of the model description file.

We set the two fields, *DataType* and *VariableType*, to describe the type of a parameter. The two fields will assist modeler configuring the type. Take a variable *a*, for example; it may be defined as *int a*, *int* $^*a$, *int[10] a*, or *int[10]* $^*a$. And the type *int* could also be *double* or *string*. As a result, it needs to preinstall all the possible type options for modeler. Let *DataType* be the primitive type such as *int*, *double*, and *string*, and let *VariableType* be the combining-mode of complex type, such as Pointer, Array, Pointer Array or vector. Through combining the two fields, it is much easier for modeler to define the variable *a* as *int* $^*a$ or *vector<string>* $^*a$. The *DataType* could also be a user-defined complex data type which is described in the *User_defined_dataTypes* block in the description file. Modeler can define a *struct*, a *union*, an *enum*, or a *class*. Each *User_defined_dataType* node describes a complex user-defined data type and the child node *member* describes the member information of the data type.

The attribute of each model member, *IsStateVariable*, indicates whether the member could be a state variable of the model. And the attribute of each interface parameter, *Corresponding_model_member*, points out which model member is interrelated with the interface parameter.

Actually, the model description file not only has the descriptive function, but also can be utilized to regenerate a model. A model description file corresponds to a model class, that is, the model header file, so that the description file also records header files, library files, and macrodefinitions used in model codes.

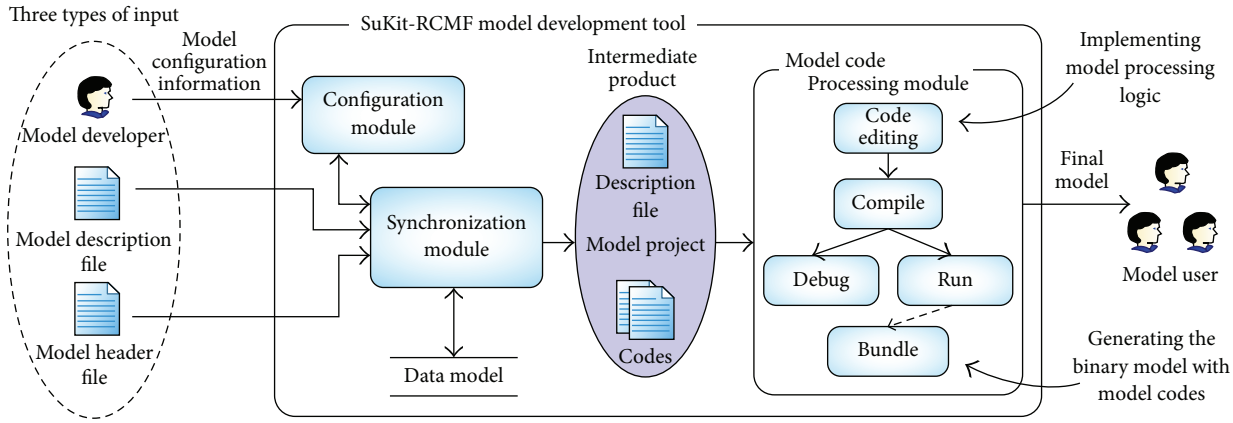

Figure 2: Architecture of SuKit.

Model description file records sufficient information which can be used to instruct simulation application developer integrating the model. And the platform-independent XML-based description file can also be parsed to generate a platform-specific (C++ or Java) RCMF model, which realized the model's reusability. Compared with the SMP Catalogue file written in SMDL, our description file with hierarchical structure is much more readable. SMDL constructed a new data type system to declare a variable, which complicated the Catalogue. We need not construct the system here. Model user just gets model information from the literal meaning of each field. And the mapping rule of generating model codes from the description file is also easy to be established.

4. The Reusable Model Development Tool SuKit

To make modeler developing RCMF model conveniently and efficiently, a useful developing tool is demanded. There are several simulation tools supporting the reusable model development, such as SimSat [19], UMF [20, 21], and MAX [22]. But they are mainly designed for SMP2 models development and cannot be applied for the development of RCMF model. In this section, we will elaborate on the RCMF model development tool, SuKit.

4.1. The Architecture of SuKit. SuKit is developed based on the Eclipse RCP framework. It is mainly composed of the following function modules: configuration module, synchronization module, and model code processing module. The architecture of SuKit is illustrated in Figure 2.

In the left dashed oval, SuKit has three possible inputs, model configuration information, description file, and header file. Any of the three inputs will initiate SuKit for model development. In the first input pattern, Modeler configures model information in the graphical configuration interface, and then the configuration information is recorded in the data class *DataModel* maintained in the synchronization module. With the information, synchronization module generates model header file and skeleton source file together with a model description file. A model project

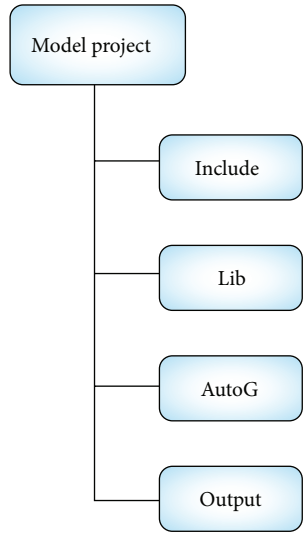

Figure 3: The structure of model project.

named as modeler configured is also created to organize these generated files at the same time. The synchronization module can also generate the skeleton model project with the input of model description file or header file. Figure 3 is the structure of the model project. Within each model project, there are four folders: *include*, *lib*, *autoG*, and *output*. The *include* folder collects header files included in the model code, while under the *lib* folder are library files, for example, a Dynamic Link Library of a computational module. The generated files model description file, header file, source file, and the makefile which will be used to make model codes are placed under *autoG*. And the *output* folder is set to gather the final products of a model.

The model project generated in synchronization module is an intermediate product. Modeler needs to further implement model's processing logic. SuKit supports two implementation patterns: (1) Import the library file of the computational module into the *lib* folder and then implement

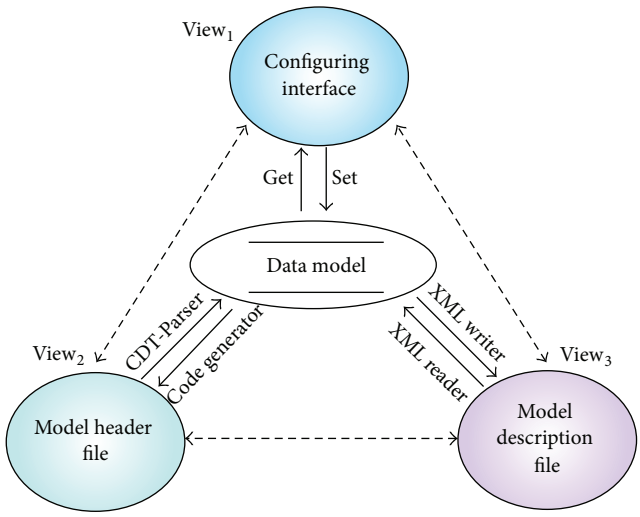

Figure 4: The synchronization mechanism in SuKit.

the processing logic by invoking functions in the computational module. (2) Implement the processing logic based on the generated model framework directly. Modeler can also create private functions in the model class if necessary. After completing the model code, modeler can compile and debug the model with SuKit and then package the final model code into a library file.

Model description file is bundled with each model and it is the needed file for model user. It provides detailed description information for application developer when integrating the model. And the simulation model is delivered in the form of source code or binary model, according to the specific simulation application.

4.2. The Synchronization Mechanism in SuKit. The model information in SuKit has three different views: the data in the configuration interface, model header file, and model description file. SuKit implemented a synchronization module to synchronize the three views. The synchronization module is designed in the MVC pattern. We set a data class *DataModel* to maintain the model information. The synchronization between the three views is shown in Figure 4.

There are two types of connection in the synchronization graph. The dashed one between different views indicates that one view of the data can be transformed to another one, but it is not a direct conversion. The synchronization between views is achieved via the second connection, the conversion between *DataModel* and views. By invoking the Seter provided by *DataModel*, the data in the configuration interface is recorded into the data class. And the interface can also obtain and show the data in *DataModel* through the Geter. With SAXReader, the XML reader provided in DOM, the information in the description file can be easily parsed and then recorded into *DataModel*, and the data in *DataModel* can also be written into the description file through XMLWriter.

The conversion between *DataModel* and model header file is much difficult. We established a mapping rule and developed a code generator to generate the standard header file with the data in *DataModel*. As for the model header file coded in C++ programming language, it can be parsed with the CDT Parser. But the parsed result is an Abstract Syntax Tree (AST), which needs a further analysis before we get the wanted information. What is more, in AST all the comment statements are organized together in a linked list, resulting in the fact that it is hard to figure out the corresponding relationship between the comment statement and the functional statement. We add a unique label in each comment statement. And the label can be constructed according to the corresponding functional statement. With the label, it is easy to find the right comment among all the comment statements for a model member or an interface parameter when parsing the header file. Then the data in model header file can be synchronized into *DataModel* with nothing left out.

We override the *Save* function of the Text Editor. When doing the save operation, we invoke the corresponding synchronization process to synchronize the edited header file into *DataModel* and the description file, the same to the edited description file. The six directions conversion not only realized the synchronization between the data class and each view but also made SuKit have good flexibility, as any of the three (configuration information, description file, or header file) could be the input of SuKit.

5. Case Study: Developing an Illustrative Radar Model with SuKit

In this section, we will present the development process of a simplified radar model with SuKit. In complex military simulation, the simulation model of radar is a frequently used component model, which can be integrated in a warship or an aircraft. The main function of radar is detection, location, searching, and tracking.

5.1. Design the Radar Model. The radar model developed here is a generic model. We can get different radar instances after initializing the radar with specific performance parameter and deployment information. The initialization information comes from the simulation scenario and is configured to the radar by invoking the *Initialize* interface. In simulation, the process of radar sending and then receiving the echoed signal is simulated by the target sending signal to radar directly. The input information of radar is the echoed signal from the target. Then the radar model will compute with the signal and provide the information of the detected target to the control center, including target basic information, target position, and its flight velocity. The simplified radar model does not need any dynamic data. The parameter information of each interface is designed as shown in Table 1. Given space limitations, members designed for the radar model are not listed in the table.

5.2. Develop the Radar Model with SuKit. Run SuKit and click on the New Model button. Then modeler configures the above information of the radar model in the pop-up WizardDialog. To make the configuration process clearer, we illustrate it in Figures 5 and 6. The marked number on each page shows

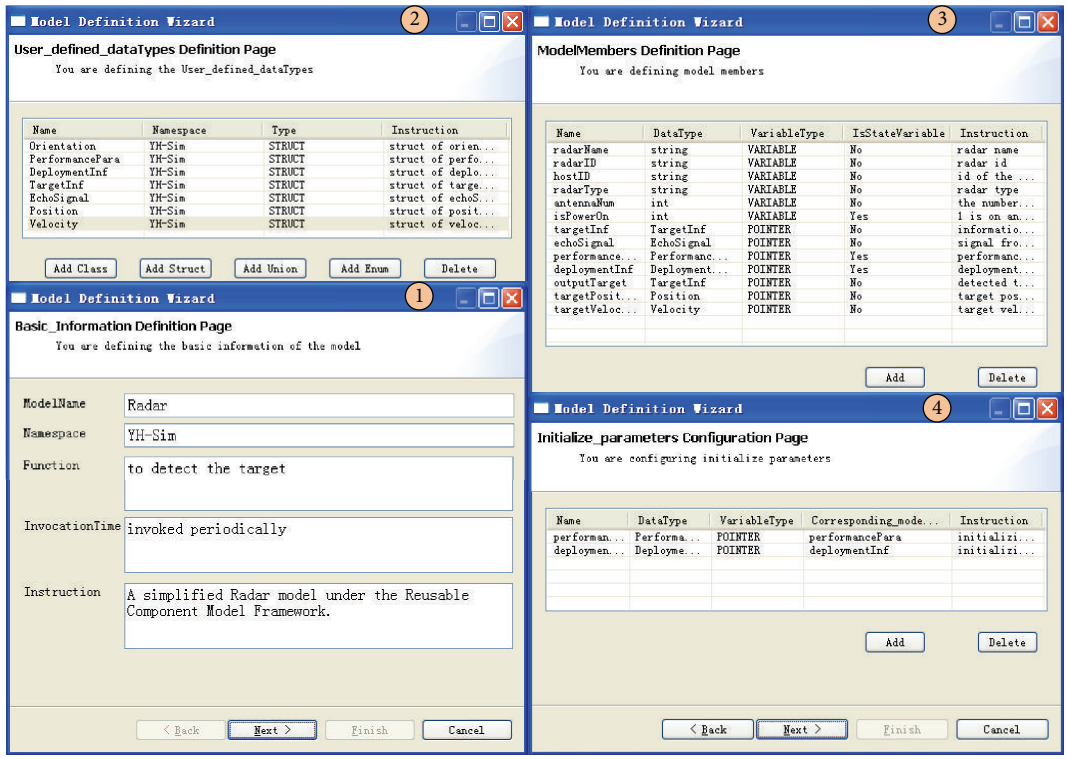

Figure 5: The configuration of radar model (1/2).

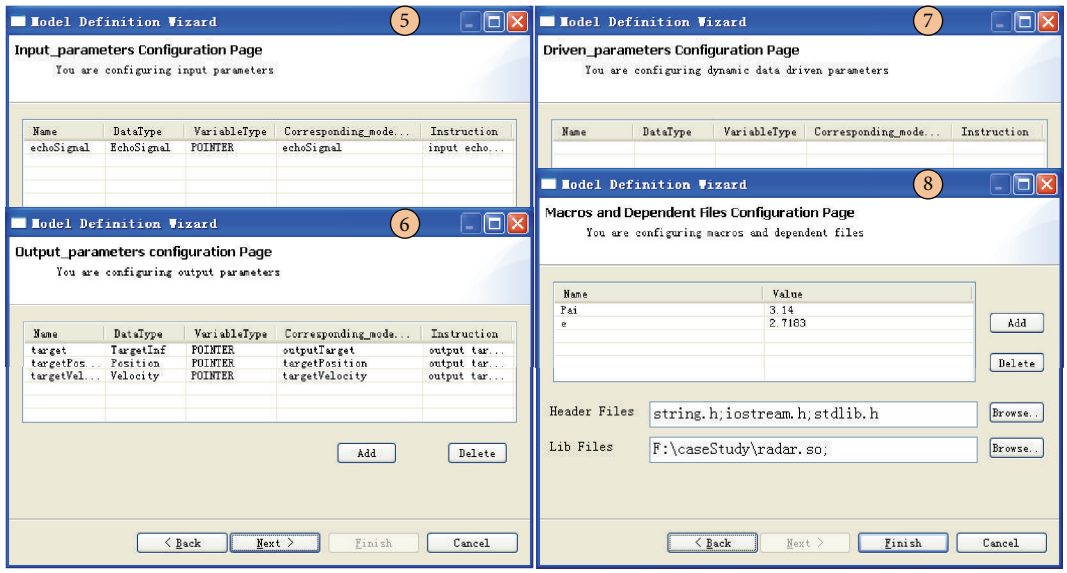

Figure 6: The configuration of radar model (2/2).

the sequence of the wizard page. The configuration process in the dialog is consistent with the structure of the model description file. The first page configures the basic information of the radar model, including model name, namespace, function, invocation time, and instruction. The second page provides the interface of defining user-defined data types and all the user-defined data types used in Table 1 are created in this page. With the *Add* function on the third page, we can define the radar model's members. The fourth page configures the initialize interface parameters of the radar model. And then configure the input interface parameters, output interface parameters, and dynamic data driven interface parameters designed in Table 1 on the following pages. As the radar model does not need any dynamic data, the seventh

Table 1: Parameter information of each interface.

	Parameter name	Parameter type	Members	Member DataType	Instruction
Initialize	performancePara	PerformancePara	radarType	string	The type of radar
			antennaNum	int	Antenna number
			orientation	Orientation	Orientation of the radar
			isPowerOn	int	1 is on and 0 is off
	deploymentInf	DeploymentInf	radarName	string	Radar name
			radarID	string	ID of the radar
			hostID	string	ID of the radar's host
Input	echoSignal	EchoSignal	targetInf	TargetInf	Target information
			signalIntensity	double	The intensity of the echo signal
			delay	double	The delay of the echo Signal
Driven	None				
Output	target	TargetInf	targetID	int	ID of the target
			targetType	string	Type of the target
			isEnemy	int	1 is enemy and 0 is not
	targetPosition	Position	longitude	double	Longitude
			latitude	double	Latitude
			altitude	double	Altitude
	targetVelocity	Velocity	V_x	double	X-velocity component
			V_y	double	Y-velocity component
			V_z	double	Z-velocity component

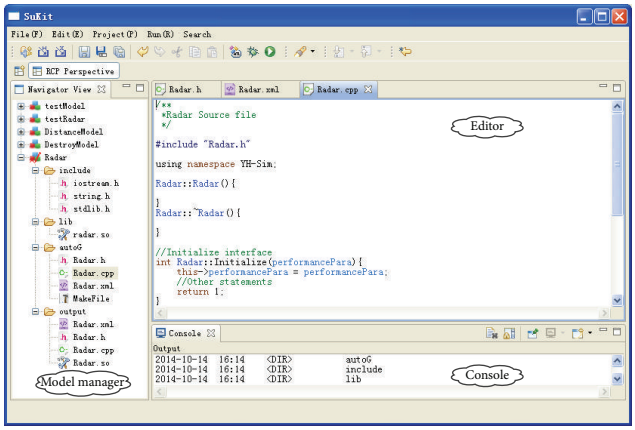

Figure 7: Snapshot of the radar model in SuKit.

page, driven-parameters configuration page, has no item. The last page configures macros, header files, and library files used in the radar model codes.

After the configuration, SuKit creates a model project radar in the current workspace and adds it to the model tree managed in the navigator view. Figure 7 gives the snapshot of SuKit with the created radar model. In the navigator view, the tick on the icon of radar means that the radar model is the active project in the workspace. Under the radar model, there are four folders, *include*, *lib*, *autoG*, and *output*. In *autoG* are the generated files, including the header file Radar.h, source file Radar.cpp, description file Radar.xml, and makefile. In the right text editor opened three files and the active page shows a fragment of the source file Radar.cpp. The console in the bottom prints the real-time information when modeler operates SuKit. The information shown currently is the directory information of radar as it is selected as the active project.

The generated description file of radar records the entire configuration information and is composed strictly following the format designed in Section 3. The information in the first page is recorded as attributes of the root node. And the user-defined data types are written in the *User_defined_DataTypes* block. Then the remaining information is organized in the corresponding child node.

The generated header file is illustrated in Algorithm 2. It is a code structure of the header file with much similar information omitted. The configured information is completely mapped into the header file. Besides the information mapped to the functional code, the instruction information is generated as a comment statement. At the beginning of each comment, the constructed string prefixed with the symbol @ is the label set for synchronization.

The processing logic of radar is completed by invoking functions in the imported library file of the computational radar model, which is placed under the *lib* folder. SuKit puts the final product of the model in the *output* folder, in which the Radar.so is the compiled dynamic library file of the radar model.

SuKit provides a rich toolbar. The first three actions respond to the three input patterns, respectively. The first one is the New Model button, which pops up the configuration

```
/**
*Radar Header file
*@Function: to detect the target
*@InvocationTime: invoked periodically
*@Instruction: A simplified Radar model under the Reusable Component Model Framework.
*/
#include <iostream>
   ...
#define Pai 3.14
#define e 2.7183
using namespace std;
namespace YHSim {
   /**
    *User_defined_dataTypes
    */
   //@struct_TargetInf Namespace:YHSim;Instruction:struct of orientation
   struct TargetInf {
         ... //struct members
   };
         ... //other user_defined dataTypes
   class Radar {
   private:
         //@model_member_radarName IsStateVariable:No;Instruction:radar name
         string radarName;
            ...
         Velocity * targetVelocity;
   public:
         Radar();
         ~Radar();
         //Default initialize function without any parameter
         int Initialize();
         //@initialize_PerformancePara_Pointer Corresponding:performancePara;Instruction:init
         int Initialize(PerformancePara * performancePara);
         //@initialize_DdeploymentInf_Pointer Corresponding:xxx;Instruction:xxx
         int Initialize(DdeploymentInf * deploymentInf);
         //@input_EchoSignal_Pointer Corresponding:xxx;Instruction:xxx
         int SetInputData(EchoSignal * echoSignal);
         //@output_TargetInf_Pointer Corresponding:xxx;Instruction:xxx
         int GetOutputData(TargetInf * target);
            ...
         int SetSimuStatus(string & SimuStatus);
         int GetSimuStatus(string & SimuStatus);
         int ModelProcess(double dSimuTime);
   };
}
```

Algorithm 2: Header file structure of the radar model.

wizardDialog. SuKit then will develop the RCMF model from configuring. The second button is an import button. It pops up a fileDialog for modeler to select a model header file to generate a description file for the model. The third button is also an import button. But modeler selects a description file here, and the description file is utilized to generate the model codes.

5.3. Discussion. With the configured model information, SuKit generates a model with a header file, a source file, and a description file. For a simulation model, the generated header file and description file are complete. Modeler just needs to implement the model processing logic in the source file. If there gets some modification or supplement, modeler can edit at one place and SuKit will synchronize the information to all the files. However, in manually development pattern, modeler needs to code the same model information into both model header file and description file. And modeler should not forget to do the corresponding modification in the description file if he revised any information in the header file, and vice versa. Figure 8 gives the developing time of the same model developed with SuKit and manually. Comparing with developing manually, SuKit saves much time as modeler just needs to configure the main information of

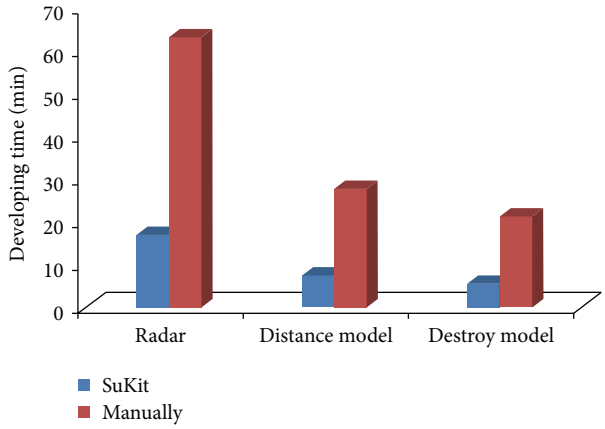

Figure 8: Developing time of models developed with SuKit and manually.

the model and then SuKit will generate the standard model codes together with the description file. Moreover, by parsing the header file, SuKit can also generate the corresponding description file for the existing RCMF model developed manually.

6. Conclusion and Future Work

At present, modelers mainly develop RCMF model through writing programming code manually. Modelers need to write a similar framework for each model, which is repetitive and boring. And modelers also need to write an extra file to describe the model. Modelers cannot focus on the implementation of model processing logic. In this paper, we first designed a platform-independent model description file for model representation, which is useful for model integration and reuse. And then we developed the RCMF model development tool—SuKit—to assist modelers developing RCMF model conveniently and efficiently. SuKit has good capability of developing RCMF models. The model description file not only can be generated along with model but also can be parsed by SuKit and used to regenerate a specific RCMF model. With SuKit, modelers just need to concentrate on model function. Work for model reuse will be done by SuKit. As for our future work, we plan to further enrich the model development capability of SuKit, so that SuKit can generate RCMF model for different platforms with the description file.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors appreciate the support from the National Natural Science Foundation of China (no. 61170048), the Research Project of State Key Laboratory of High Performance Computing of National University of Defense Technology (no. 201303-05), and the Research Fund for the Doctoral Program of Higher Education of China (no. 20124307110017).

References

[1] F. Zhu, Y.-P. Yao, H. Chen, and F. Yao, "Reusable component model development approach for parallel and distributed simulation," *The Scientific World Journal*, vol. 2014, Article ID 696904, 12 pages, 2014.

[2] H. Lee, J.-S. Yang, K. C. Kang, and J.-J. Pyun, "Domain-oriented variability modeling for reuse of simulation models," *Simulation*, vol. 90, no. 4, pp. 438–459, 2014.

[3] K. L. Edwards, "Exploring the reusability of discrete-event simulation models: a case study of project challenges and issues of feasibility," in *Proceedings of the IEEE Systems and Information Engineering Design Symposium (SIEDS '13)*, pp. 7–12, Charlottesville, Va, USA, April 2013.

[4] Y.-L. Lei, *Research about the Theory and Approaches of Simulation Model Reuse and Heterogeneous Integration Techniques*, National University of Defense Technology, Changsha, China, 2006.

[5] P. Gustavson, J. Hancock, and B. Lutz, "BOM study group final report," in *Proceedings of the Simulation Interoperability Workshop*, March 2001.

[6] SISO BOM PDG, "Base Object Model (BOM) Template Specification," SISO-STD-003-2006, 2006.

[7] SISC, *IEEE Standard for Modeling and Simulation High Level Architecture (HLA)-Framework and Rules*, 2000.

[8] European Space Agency, *Simulation Model Portability Handbook*, EWP-2080, no. 1.1, 2000.

[9] European Space Agency, *SMP 2.0 Handbook*, EGOS-SIM-GEN-TN-0099, Issue 1.0, Revision 2, 2005.

[10] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling And Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press, 2000.

[11] B. P. Zeigler, "DEVS today: recent advances in discrete event-based information technology," in *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS '03)*, pp. 148–161, October 2003.

[12] European Space Agency, *SMP2.0 C++ Mapping*, EGOS-SIM-GEN-TN-0102, no. 1.0, Revision 2, 2005.

[13] Y.-P. Yao and Y.-X. Zhang, "Solution for analytic simulation based on parallel processing," *Journal of System Simulation*, vol. 20, no. 24, pp. 6617–6621, 2008.

[14] T. L. Wilmarth and L. V. Kalé, "POSE: getting over grainsize in parallel discrete event simulation," in *Proceedings of the International Conference on Parallel Processing (ICPP '04)*, pp. 12–19, August 2004.

[15] K. R. Bisset, A. M. Aji, E. Bohm et al., "Simulating the spread of infectious disease over large realistic social networks using Charm++," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW '12)*, pp. 507–518, IEEE, Shanghai, China, May 2012.

[16] S. W. Reichenthal, "SRML case study: simple self-describing process modeling and simulation," in *Proceedings of the Winter Simulation Conference*, pp. 1461–1466, December 2004.

[17] R. A. Kilgore, "Open source Simulation Modeling Language (SML)," in *Proceedings of the Winter Simulation Conference*, vol. 1, pp. 607–613, December 2001.

[18] T. Wiedemann, "Next generation simulation environments founded on open source software and XML-based standard interfaces," in *Proceedings of the Winter Simulation Conference*, pp. 623–628, December 2002.

[19] A. Walsh and N. Lindman, "SIMSAT 3.0—a platform independent version of ESOC's simulation infrastructure," in *Proceedings of the 8th International Workshop on Simulation for European Space Programmes (SESP '04)*, 2004.

[20] A. Walsh, M. Pecchioli, V. Reggestad, and P. Ellsiepen, "ESA's model based approach for the development of operational spacecraft simulators," in *Proceedings of the 13th International Conference on Space Operations*, May 2014.

[21] P. Ellsiepen, P. Fritzen, V. Reggestad, and A. Walsh, "UMF—a productive SMP2 modelling and development tool chain," in *Proceedings of the Simulation and EGSE Facilities for Space Programes (SESP '12)*, September 2012.

[22] C. Lannes, "MAX-A simulator of the ground station equipment monitoring and control based on SIMSAT and SMP2 compliant models," in *Proceedings of the Data Systems In Aerospace (DASIA '08)*, May 2008.