A Bidirectional Generation Method of SmartC Models and Codes

Liping Zhang, Minde Zhao, Chao Wang, Ruyi Wu, Hong Li, Wang Dongdong
College of Computer Science
Zhejiang University, Hangzhou, Zhejiang, 310027, P.R.China
{zhang_liping, zmdd48, ariose, wuruyi, lihong, tk118}@zju.edu.cn

Renfa Li
School of Computer and Communication
Hunan University
Changsha, Hunan, 410082, P.R.China
lrf@hnu.cn

Abstract

This paper proposes a bidirectional generation method with a set of consistent bidirectional generation rules between SmartC models and codes. Based on these rules, the consistency of the bidirectional generation between SmartC models and codes is demonstrated by a case study. Extensive tests are conducted to show the performance of this bidirectional generation method. And the efforts of different industrial applications are recorded to illustrate its advantage of decreasing project efforts and accelerating project progress.

1. Introduction

Traditional development method of automotive electronic software follows the V-Model development mode, whose process consists of four steps: requirement analysis, system design, implementation and test [1] [2]. This method, however, has some disadvantages when the four key steps need to communicate with each other, such as the ambiguous description of the text-based requirement documents and the risk of hand-writing code.

In order to overcome these shortcomings, researchers proposed another software development method characterized by a model-driven architecture (MDA) [3]. Different levels of models are used as a unified description throughout all stages of software development. The new method based on MDA can enhances the consistency throughout the different design stages as well as the efficiency of software development by the auto-generation of codes and graphics.

Based on the essential idea of model-driven software design, a Model-based Driven Automotive Electronics Development Approach (ModaEDA) is proposed as an alternative method which introduces the MDA into automotive embedded software development [4]. ModaEDA needs domain specification and description languages to apply the modern driven description and development modes into automotive embedded software development. In our preliminary work, we proposed a novel specification and description language for automobile electronic control systems, named SmartC [5]. The SmartC language describes the design requirements of automotive control softwares as hierarchy models and supports two kinds of components of different grain-sizes. After fulfilling the requirement description and system design using SmartC, C-code files can be generated automatically. Automotive electronic engineers can adopt SmartC to develop control softwares and other applications, especially for our real-time operating system SmartOSEK OS [6] for automobiles. Cooperated with Chery Automobile Corporation and the Chinese FAW Car Corporation, we have developed an automotive automatic transmission (AMT), car lights control modules and auto mirror control modules using SmartC language as the specification and description language for ModaEDA. The AMT software module has been applied successfully in a Chery QQ EZDrive car.

The model and the code are two equivalent ways to describe SmartC. During the development of an automotive electronic control system, it is very difficult for the developers familiar with SmartC model design to communicate with other developers skilled in SmartC code design consistently. In order to solve this problem, we propose a bidirectional iterative development pattern with a bidirectional generation method for SmartC model and SmartC code design and the bidirectional auto-generation rules between SmartC models and their equivalent codes. Most importantly, the auto-generation rules can make sure the accurate and consistent generation between SmartC models and codes. This bidirectional generation method includes the SmartC code generator from model to code (SmartCG), and the SmartC model generator from code to model(SmartMG).

The remainder of this paper is organized as follows. Section 2 discusses state-of-the-art of relevant research. Section 3 presents the bidirectional auto-generation rules between SmartC models and SmartC codes. And in section 4, experiments and industrial applications are discussed. At last, section 5 concludes this paper.

2. Related work
The related work of this paper ranges from domain specific languages to language environments for embedded systems.

Domain specific languages (DSL) are advantageous to reusing models and validating models. There are many DSLs to meet the requirement of different research and industrial fields. Many well-known domain specification languages are available, such as galsC [7], which is used widely in various fields of industrial applications. Utilizing galsC in development can improve the performance of resource restricted and security-critical system for its high-quality and efficient code generation and software analysis. As an extending version of nesC [8] [9], galsC focuses on developing simple event-driven embedded systems such as sensor networks. Besides, Click [10] is designed for network routers while Lustre [11] is a development language for security-critical fields. The components of Bluespec [12] [13] models contain methods, active rules and constraints, which is more suitable for digital logic circuit design than other fields [14] [15]. Most of the synchronous languages cannot model the system hierarchically because they lack the ability of high-level description, and none of the above embedded domain-specific languages is developed specifically for the domain of automotive control systems. For the lack of a quantifiable notion for automobiles, they are not convenient to model automotive electronics real-time systems. Therefore, in our former work we proposed an automotive electronic control specific language, SmartC. It supports hierarchical modeling with two describing modes, the model and the code.

Embedded visual language development environment is another related area. Especially, Simulink [16] with real-time workshop, widely used in the automotive industry, is a powerful tool created for embedded control software. It uses sub-systems for structural modeling, supports hierarchical nested modeling and user-defined functions library loading, and introduces pre-defined S-Functions as the basic units of a system. However, Simulink does not consider the abstract of the models at different levels and is short of formalized semantics specifically for automotive electronics applications. As an application development environment for safety-critical embedded software, SCADE [17] is widely used in avionics, defense and energy industry, which can generate C code conforming to the specification of DO-178B, IEC61508 and TUV SUD. MoBIES [18] program orients to model-based approaches for embedded software composition and analysis, especially focusing on nonfunctional issues such as timing, synchronization, dependability and resource constraints. GME [19] is another toolset which supports the creation of domain-specific modeling and program synthesis environments. Rhapsody [20] is a graphic development environment through which developers could build and deploy real-time embedded software applications.

All these languages discussed above have been applied to certain areas due to their ability of generating C code automatically. Our smartC is designed as a specific language for the automobile electronics domain. And up to now, none of the above IDEs supports SmartC, not to mention the bidirectional auto-generation between the two description modes of SmartC.

3. Consistent bidirectional generation rules

A bidirectional generation method of the SmartC models and SmartC codes, as shown in Figure 1. In our bidirectional generation method, we implement these rules as two generating tools assistant. Most specifically, the main functions of two parts are summarized as follows.
- SmartCG: the tool auto-generating SmartC code description from SmartC model description;
- SmartMG: the tool auto-generating SmartC models from SmartC code description.

The bidirectional generation method supports the bidirectional iterative development pattern between SmartC model description and SmartC code description.

![Figure 1. A Bidirectional Generation Method](image)

The bidirectional auto-generation rules can make sure the accurate and consistent generation between SmartC models and their equivalent codes. In this section, we will first introduce the definitions of major elements in SmartC language, and then discuss the semantics of the bidirectional auto-generation rules. The consistency of the bidirectional auto-generation will be proved in section 4.1 with a case study.

3.1. Definitions of major elements in SmartC

Definition 1. **Component**: A component is a definite functional sub-unit in automotive electronics applications.
Definition 2. **Device**: A device is an external interface module interacting to components, including Actuator objects and Sensor objects.

Definition 3. **Network Equipment**: A network equipment is an abstract model of automotive electronic control networks.

Definition 4. **Connection object**: The connection objects express the transitive relations between data signals and control signals in models, named Data Link and Control Link respectively.

Definition 5. **Data object**: A data object is a type of data representation, and it is an important corresponding carrier between entities.

Definition 6. **Task**: A task is an executive module including a group of functions, providing the frame for function to execute.

Definition 7. **Task Supporting Item**: Task Supporting Items provide the scheduling tools for executing tasks, including Alarm CallBack functions, Counters, Alarms, Events, Interrupts and Discrete Data.

Definition 8. **Local Component**: A local component is the most basic modeling unit in a system, which is just effective locally in the domain of a component.

Definition 9. **Control Flow Item**: Control Flow Items specify the order in which the local component models are called. They include LoopSelect, LoopMerge, BranchSelect and BranchMerge.

Definition 10. **System Layer Model**: A system layer model is a five-tuple array model \( S=(C_p, Dev, Nwd, L_c, L_d) \), where: \( C_p \) is a system component; \( Dev \) is system equipment; \( Nwd \) is system network equipment; \( L_c \) is a system control connection; \( L_d \) is a system data connection.

Remark:
\[
L \subseteq \{a \times b \times b \times a | a \in \{Dev, Nwd\} \rightarrow b \in \{C_p, Dev, Nwd\} \rightarrow \{a\}\}
\]
\[
L \subseteq C \times C
\]
\[
\forall a \in \{Dev, Nwd\} \Rightarrow \exists C_p \wedge \exists L_c(a, C_p) \wedge \exists C \wedge \exists L_d(a, C_p)
\]

Definition 11. **Component Layer Model**: A component layer model is a five-tuple array model \( C=(T_s, D, TSI, L_c, L_d) \), where: \( T_s \) is a component task; \( L_c \) is a component control connection; \( L_d \) is a component data connection; \( D \) is a data object; \( TSI \) is a task supporter. If \( P_c \) and \( P_d \) represent the control port and data port of this component respectively, remark:
\[
L \subseteq \{TSI, T_s \times \{TSI, T_s\} \times \{TSI, T_s\} \times \{P_c \} \times \{P_c \} \times \{TSI, T_s\}\}
\]
\[
L \subseteq \{T_s, D, P_c \times \{TSI, T_s\} \times \{TSI, T_s\} \times \{P_c \} \times \{P_c \} \times \{TSI, T_s\}\}
\]
\[
\forall TSI \rightarrow \exists x = T_s \wedge \exists x = T_s \wedge \exists L_c(x, TSI) \wedge \exists L_d(x, TSI)
\]
\[
\forall D \rightarrow \exists x = T_s \wedge \exists L_c(x, D) \wedge \exists L_d(x, D)
\]
\[
\forall P_c \rightarrow \exists x = T_s \wedge \exists L_c(x, P_c) \wedge \exists L_d(x, P_c)
\]

Definition 12. **Task Layer Model**: A task layer model is a five-tuple array model \( T=(C_i, D, SC, L_c, L_d) \), where \( C_i \) is a local component; \( D \) is a data object used in task; \( L_c \) is a data connection; \( L_d \) is a control connection; \( SC \) is a control flow item. If the \( P_c \) and \( P_d \) represent the control port and data port of this component respectively, remark:
\[
L \subseteq \{SC, C_i \times \{SC, C_i\} \times \{SC, C_i\} \times \{P_c \} \times \{P_c \} \times \{SC, C_i\}\}
\]
\[
L \subseteq \{C_i, P_c \times \{D\} \times \{D\} \times \{SC, C_i\} \times \{SC, C_i\}\}
\]
\[
\forall D \rightarrow \exists l \rightarrow \exists x \in \{x, y\} \rightarrow \exists x \in \{x, y\}, x, y = SC, z = P_c
\]
\[
\forall SC \rightarrow \exists l \rightarrow \exists x \in \{SC\} \rightarrow \exists x \in \{SC, C_i\} \rightarrow \exists x \in \{SC, C_i\} \rightarrow \exists x \in \{SC, C_i\}
\]

3.2. The consistent bidirectional auto-generation rules

**Rule 1**: **Entity Generation Rule**

\[
\text{Entity Gen}(x): \forall x \in \{X\} \rightarrow \text{Entity Gen}(x)
\]
\[
\# \text{declare}\ X\_\text{Type}: x
\]
\[
\# \text{define}\ X\_\text{Type}: x
\]
\[
\{ X\_\text{properties} \}
\]

Where \( X \in \{C_p, Dev, Nwd, T_s, D, TSI, L_c, L_d\} \). The property of \( X\_\text{Type} \) are denoted as \( X\_\text{properties} \). The Entity Gen rule is used to analyze the model entities in model files and generate the corresponding SmartC code sections, while the anti_Entity Gen rule is to analyze the SmartC code sections and generate corresponding SmartC model entities.

**Rule 2**: **Control Flow Generation Rule**

\[
\text{ControlFlow Gen}(t): \forall t \in \{T\} \rightarrow \text{ControlFlow Gen}(t)
\]
\[
\# \text{define}\ _\text{Entry}: \}
\]
\[
\{ t\_\text{FlowDescription} \}
\]

Where the \( t\_\text{FlowDescription} \) is a set logically combined by \( SC \) and \( Cl \), which includes initializeBuffer(Cl), branchBuffer(SC, Cl), bodyBuffer(Cl), mergeBuffer(Cl,SC) and finishBuffer(Cl). The ControlFlow Gen rule is used to parse the control flow of SC and Cl in model files, and generate the corresponding SmartC code sections, while the anti_ControlFlow Gen rule completes the reverse operation.

**Rule 3**: **System Layer Horizontal Generation Rule**

\[
\text{Sys gen}(C_p, Dev, Nwd, L_c, L_d) \rightarrow \text{Sys gen}(C_p, Dev, Nwd, L_c, L_d)
\]
\[
\Leftrightarrow (\text{Entity gen}(C_p), \text{Entity gen}(Dev), \text{Entity gen}(Nwd))
\]
\[
| \text{Entity gen}(L_c), \text{Entity gen}(L_d) \}
\]

For the S layer model \( \text{Sys gen}(C_p, Dev, Nwd, L_c, L_d) \), we use the generation rule \( \text{Sys gen}_h \) at S layer, and generate the corresponding SmartC code. The \( \text{Sys gen}_h \) rule includes
two logical generation sub-processes which are connected with pipeline "|". Firstly, use the Entity_gen rule to generate SmartC codes from the model entities C_p, Dev, and Nwd. Then we use the entity information as the source and destination of connected entities. Finally the Entity_gen rule is used again to generate the connected entities. Similarly, the anti_ControlFlow_Gen Rule completes the reverse operation.

**Rule 4: Component Layer Horizontal Generation Rule.**

\[
\begin{align*}
\text{Comp}(T_s, D, TSI, L_c, L_d) & \leftarrow_{\text{comp}_{\text{h}}_{\text{gen}}} \text{comp}_{\text{h}}_{\text{gen}} \\
\text{Com}_c_{\text{ode}} & \leftarrow_{\text{anti comp}_{\text{h}}_{\text{gen}}} \\
\text{Entity}_\text{gen}(T), \text{Entity}_\text{gen}(D), \text{Entity}_\text{gen}(TSI) & \leftarrow_{\text{comp}_{\text{h}}_{\text{gen}}} \\
| \text{Entity}_\text{gen}(L_c), \text{Entity}_\text{gen}(L_d) & \leftarrow_{\text{anti comp}_{\text{h}}_{\text{gen}}}
\end{align*}
\]

For a C layer model \( \text{Comp}(T_s, D, TSI, L_c, L_d) \), we use the generation rule comp_h_gen for C layer, and generate the corresponding SmartC code. The comp_h_gen rule includes two logical generation sub-processes which connect with pipeline "|". The detail operations are similar to the Sys_h_gen rule. The anti_ControlFlow_Gen Rule completes the reverse operation in the similar way as the comp_h_gen rule.

**Rule 5: Task Layer Horizontal Generation rule.**

\[
\begin{align*}
\text{Tsk}(C_i, D, SC, L_c, L_d) & \leftarrow_{\text{tsk}_{\text{h}}_{\text{gen}}} \text{tsk}_{\text{h}}_{\text{gen}} \\
\text{Tsk}_c_{\text{ode}} & \leftarrow_{\text{anti tsk}_{\text{h}}_{\text{gen}}} \\
\text{Entity}_\text{gen}(C_i), \text{Entity}_\text{gen}(D), \text{Entity}_\text{gen}(SC) & \leftarrow_{\text{tsk}_{\text{h}}_{\text{gen}}} \\
| \text{Entity}_\text{gen}(L_c), \text{Entity}_\text{gen}(L_d) & \leftarrow_{\text{anti tsk}_{\text{h}}_{\text{gen}}}
\end{align*}
\]

For a task model \( \text{Tsk}(C_i, D, SC, L_c, L_d) \), the generation rule tsk_h_gen is used to generate the corresponding SmartC codes in component layer. The tsk_h_gen rule includes three logical generation sub-processes which are connected with the pipelines "|". First we use the Entity_gen rule to generate SmartC code from the model entities \( C_i, D \) and SC. Then we use the entity information as the source and destination of connected entities. Next, the Entity_gen rule is used again to generate the connected entities. Finally we use the ControlFlow_Gen Rule and generate the corresponding SmartC code. Anti tsk_h_gen Rule completes the reverse operation in the similar way.

**Rule 6: Use operator**

The Use operator is an operation which uses the reference keyword Use to present the hierarchical relations of extensible models (code).

**Rule 7: Name operator**

The Name operator is an operation which uses specific naming methods to present the relationships of hierarchical layers.

\[
\begin{align*}
\text{Sys}_\text{Name}(x) & \leftarrow_{\text{sys}_{\text{Name}_{\text{h}}_{\text{gen}}}} \text{sys}_{\text{Name}_{\text{h}}_{\text{gen}}} \\
x.Name & \leftarrow_{\text{anti sys}_{\text{Name}_{\text{h}}_{\text{gen}}}} \\
| \forall x \in \{SC_p\} & \leftarrow_{\text{sys}_{\text{Name}_{\text{h}}_{\text{gen}}}} \\
\text{Sys}_\text{Name}(x) & \leftarrow_{\text{anti sys}_{\text{Name}_{\text{h}}_{\text{gen}}}} \\
\text{Tsk}_\text{Name}(z) & \leftarrow_{\text{tsk}_{\text{Name}_{\text{h}}_{\text{gen}}}} \text{tsk}_{\text{Name}_{\text{h}}_{\text{gen}}} \\
x.Name & \leftarrow_{\text{anti tsk}_{\text{Name}_{\text{h}}_{\text{gen}}}} \\
| \forall z \in \{x, y, T\} & \leftarrow_{\text{tsk}_{\text{Name}_{\text{h}}_{\text{gen}}}} \\
x.Name & \leftarrow_{\text{anti tsk}_{\text{Name}_{\text{h}}_{\text{gen}}}}
\end{align*}
\]

**Rule 8: Vertical Hierarchical Generation rule (VHG)**

Vertical Hierarchical Generation (VHG) Rule uses Use operator and Name operator to implement the vertical hierarchical generation from models to SmartC code. The VHG rule also uses the Use operator to extract the extensible information among the layers from extensible entity and write this extensible information into SmartC code. The extensible entities include the \( C_p \) in S layer and the \( T_i \) in C layer. And it also uses the Name operator to conduct the hierarchical generation from hierarchical models to the names of SmartC code files. Anti vertical hierarchical generation rule performs the reverse operations.

**4. Experiments and Industrial Applications**

To evaluate the effect of the bidirectional generation method of SmartC models and SmartC codes, we conduct a series of experiments. It is illustrated that this method can ensure the consistency of the auto-generation between SmartC codes and SmartC models in developing a car light control system. Besides, we use a wide range of test cases to compare the generation time costs of the bidirectional generation method in different situations, getting the performance of the method. Finally, we compare three industrial applications to prove that the bidirectional generation method of SmartC models and SmartC codes could greatly accelerate the project progress.
4.1. Cases study

Figure 2 is a diagram illustrating the development vision of a case study. We use the bidirectional generation method of SmartC models and SmartC codes to develop a backlight control system (CLC) for a Chery car. From top to bottom, there are the SmartC model diagrams designed hierarchically in such turn: the S layer model (A), the C layer model (B) and T layer model (five diagrams C1-5). In the right part of Figure 2, from top to bottom there are the corresponding SmartC code diagrams to the model diagrams.

By the forward generation rules labeled with ①-⑦ (the label doesn’t mean the operating order), SmartCG completes the horizontal auto-generation of the SmartC code, and then generates the vertical SmartC code using the rules labeled with ⑧. Similarly, SmartMG uses the reverse rules labeled with ①-⑧ to automatically transform the right code to corresponding models.

The bidirectional automatically generation of SmartC models and codes conforms to the iterative development pattern and guarantees the consistency between the models and codes of all the three layers (S, C and T). For the generation in S layer, as is seen in label ①, four rules Rule1, Rule3, Rule6 and Rule7 are used to assure the consistent generation between models and codes in S layer. The specific process is as follows. First, the code file in S layer is generated; and then according to Rule7, it is named as backlight.stm. Second, Rule1 and Rule3 are used to transform all the models in S layer into corresponding SmartC codes. The horizontal consistent generation from models to codes in S layer is completed. Finally Rule6 is used to complete the consistent generation form the vertical hierarchical model information to code in S layer.

That is to say, forward rules denoted by ① are used to guarantee the consistent auto-generation from models to codes in S layer of SmartC and corresponding reverse rules with label ① are used to conduct the consistent auto-generation from codes to models in S layer in similar way.

4.2. Performance Evaluation

In this section, we use a computer which is configured with Pentium(R) Dual_Core 2.50G, 2047M memory and Windows Vista TM Ultimate OS to conduct following tests.

We scale the model numbers of S layer, C layer and T layer respectively and get the following four test types to carry on the performance evaluating experiments.

- **Rectangle shape:** the model numbers of three layers are equal, from 2 to 31. The increment is 1.
- **T shape:** the number of models in S layer increases from 2 to 31, and the step size is 1, while the numbers of models in C layer and T layer are both 1.
- **Diamond shape:** the number of models in C layer increases from 2 to 31, and the step size is 1, while the numbers of models in S layer and T layer are both 1.
- **T-down shape:** the number of models in T layer increases from 2 to 31, and the step size is 1, while the numbers of models in S layer and C layer are both 1.

For the testing auto-generation from models to codes, we repeated the experiment 100 times and took the average value as the generation time cost. Figure 4 illustrates the time costs of first ten code generation. It is clear that the first generation time cost is longer than the second one. And iterative generation time costs after the second one tends to be stable. The reason is that SmartCG uses file cache mechanism. If the code is generated for the first time, the code cache files are empty. SmartCG needs to read all the model information. But in the process of latter iterative generation, there already exists the code information generated by the first time in the code cache file. SmartCG only needs to generate code for the increased or revised model information. The time
cost for the second generation is much shorter than the first, reducing by 51.85%.

![Figure 4. Generation Time Cost: different generating times (model->code)](image)

Figure 4. Generation Time Cost: different generating times (model->code)

![Figure 5. Generation Time Cost: different generating times (code->model)](image)

Figure 5. Generation Time Cost: different generating times (code->model)

SmartMG provides the function of auto-generation from codes to models. Use the test types corresponding to those in Figure 4. Comparing the first 10 model generation time costs of these four types of tests in Figure 5, we can conclude that the first 10 generation time costs are almost same and the generation cost is not related to the generating order, dissimilar with the situations in Figure 4. SmartMG doesn’t use the file cache mechanism, and this will be complemented in our future work. In addition, comparing Figure 4 with Figure 5, it can be seen that the generation time cost from models to codes is far shorter than those from codes to models of the same test type. The output file of SmartCG (simple texts) is simpler and has less information than the output file of SmartMG (diagrams containing information such as pixels, color, coordinates and so on). At the same time, it can be seen from the test experiments that the model generation time cost of SmartMG also have the corresponding performance characteristics same as the code generation time cost of SmartCG.

4.3. Industrial Applications

Through the cooperation with Qirui Corporation and the Chinese FAW Car Corporation, we have developed the automobile automatic transmission (AMT), a light control system and a window control system for their car products. We also record the efforts of projects which use different development approaches: using the bidirectional generation method of SmartC models and SmartC codes or not. The effort is measured by Person*Months, which means the time of a person engaging in the software development in a month.

We define different development policies as follows:

- **A**: build SmartC model manually.
- **B**: write SmartC code manually.
- **C**: generate SmartC codes automatically from existing SmartC models by the SmartC development method.
- **D**: generate SmartC models automatically from existing SmartC codes by the SmartC development method.
- **A+B**: build models and write the SmartC codes manually without using the bidirectional generation method of SmartC models and SmartC codes.
- **A+C**: use the bidirectional generation method of SmartC models and SmartC codes to generate codes automatically after building SmartC models manually.
- **B+D**: use the bidirectional generation method of SmartC models and SmartC codes to generate models automatically after writing SmartC codes manually.

<table>
<thead>
<tr>
<th>Development Policies</th>
<th>AMT</th>
<th>CLC</th>
<th>CWC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+B</td>
<td>750.4473</td>
<td>29.2112</td>
<td>1.1805</td>
</tr>
<tr>
<td>A+C</td>
<td>154.87</td>
<td>6.2462</td>
<td>0.298</td>
</tr>
<tr>
<td>B+D</td>
<td>595.62</td>
<td>21.4826</td>
<td>0.9061</td>
</tr>
</tbody>
</table>

Table 1. Development Efforts (Person*Months)

In developing the three industrial applications, we use develop strategies A+B, A+C and B+D in three development groups and record the corresponding development efforts, as is listed in Table 1 and shown in

![Figure 6. Development Efforts](image)
The relationship between effort and development policies (AMT, CLC, CWC practical development cases)

For the project development requiring the project documents to use both code and model description, comparing the three different development methods, A+B, A+C and B+D in Figure 6, we can get the result: A+B>B+D>A+C. Compared to A+B, the ways of B+D and A+C reduces effort by 20.85% and 79.33% respectively.

The reason is that we should completely two sets of description, models and codes independently if we don’t use the bidirectional generation method of SmartC models and SmartC codes in development.

The policy A+C completes the auto-generation from models to codes by using the bidirectional generation method of SmartC models and SmartC codes, while policy B+D completes the auto-generation from codes to models by using the bidirectional generation method of SmartC models and SmartC codes. In contrast, the former can speed up the development process, and also reflects the superiority of the model-based design method.

5. Conclusion and future work

In this paper, we propose a set of bidirectional generation rules between SmartC codes and models, which can ensure the consistency in the auto-generations. Based on these consistent generation rules, a method, the bidirectional generation method of SmartC models and SmartC codes is implemented to provide automotive electronic engineers with auto-generation tools between SmartC models and SmartC codes. The method ensures the consistency and correctness between models and codes, which is illustrated in a case study. By comparing the development efforts of three industrial applications when use our the bidirectional generation method of SmartC models and SmartC codes or not, it is shown that SmartC-PLF has the advantage of accelerating developing progress for automotive electronic systems.

The future work will cover approaches of auto-generating SmartC models from UML’s platform-irrelevant models (PIM) and improving the description ability of specific algorithms in T layer. In the implementing aspects, user experience and the describing ability of SmartC language need to be improved.

References


