JAIST Repository
https://dspace.jaist.ac.jp/

Title	A case study: SOFL + model checking for OSEK/VDX application
Author(s)	Cheng, Zhuo; Zhang, Haitao; Tan, Yasuo; Lim, Yuto
Citation	Lecture Notes in Computer Science, 9559: 132-146
Issue Date	2016
Type	Journal Article
Text version	author
URL	http://hdl.handle.net/10119/14073
Rights	This is the author-created version of Springer, Zhuo CHENG, Haitao ZHANG, Yasuo TAN and Yuto LIM, Lecture Notes in Computer Science, 9559, 2016, 132-146. The original publication is available at www.springerlink.com, http://dx.doi.org/10.1007/978-3-319-31220-0_10
Description	5th International Workshop, SOFL+MSVL 2015, Paris, France, November 6, 2015. Revised Selected Papers

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

A Case Study: SOFL + Model Checking for OSEK/VDX Application

Zhuo Cheng, Haitao Zhang, Yasuo Tan, and Yuto Lim

School of Information Science, JAIST
Nomi, Ishikawa 923-1292, Japan
{chengzhuo, zhanghaitao, ytan, ylim}@jaist.ac.jp

Abstract. OSEK/VDX, a standard of automobile OS, is proposed to support the development of high-quality automotive applications. With its widely adopted, more and more automotive applications have been developed based on OSEK/VDX OS. As the continuously increasing complexity in the development of applications, how to efficiently develop an application is becoming a challenge. A primary problem is the requirement specification may not be accurately and easily understood by the developers carrying out different tasks. The major reason is the usage of informal languages or notations in the specification. To solve this problem, formal specification provides a feasible solution. However, some difficulties (e.g., high requirement of significant abstraction and mathematical skills) has hindered the widely usage of formal method. To address these difficulties, SOFL, a formal engineering methodology, has been proposed. In this paper, in order to investigate and study how SOFL can be used to help develop an OSEK/VDX application, we conduct a case study of cruise control system. Through the case study, we can see that SOFL specification can effectively help developer to develop an OSEK/VDX application throughout the development process.

1 Introduction

With consumers' insatiable appetite to pursue a better driving experience, more and more automotive applications have been developed. In order to support the development of high-quality automotive applications, and resolve the problem of increasing software content in automobiles, OSEK/VDX [1] [2], a standard of automobile OS, was proposed in 1994. It has been widely adopted by many automobile manufacturers to design and develop an automobile OS, such as BMW, Opel, and Volkswagen. With its widely adopted, a growing number of automotive applications have been developed based on OSEK/VDX OS. However, as the continuously increasing complexity in the development of applications, how to efficiently develop an application is becoming a challenge.

A primary problem is the requirement specification may not be accurately and easily understood by the developers carrying out different tasks. The major reason causing that problem is the notations and languages used in the specification lack a precise syntax and semantics. These notations and languages

inevitably associate ambiguity and may lead to misunderstanding. To solve this problem, formal specification gives a feasible solution. With precise constrain of semantics and syntax, formal specification can precisely define the behavior of the software, and provide a firm basis for next developers to design and verify the program.

However, there exist some difficulties in using formal method. For example, it requires significant abstraction and mathematical skills; it usually costs more in time and human effort for analysis and design [3]. These difficulties have hindered the widely usage of formal method. To address these difficulties, SOFL, a formal engineering methodology, has been proposed in [3] [4]. It proposes changes to software process, notation, methodology, and support environments for constructing systems, which makes formal methods more practical and acceptable.

In this paper, in order to investigate and study how SOFL can be used to help develop an OSEK/VDX application, we conduct a case study of cruise control system. Through the case study, we can see that SOFL specification can effectively help developer to develop an OSEK/VDX application throughout the development process.

The remainder of this paper is organized as follows. Section 2 gives an overview of our developing process. The formal requirement specification of the cruise control system is shown in section 3. In section 4, the design and implementation of the system is given. Simulation and verification of the developed application is illustrated in section 5. Section 6 concludes the paper and gives future work.

2 Overview

Our developing process is divided into three stages. The first stage is for requirement specification. Through a survey of various cruise control systems equipped in different kinds of automobiles, the primary functions of a cruise control system is given. According to the given functions, a formal requirement specification based on SOFL notations is documented.

According to the specification, the next stage is to design and implement the cruise control system. As the cruise control system is developed as an application running on an OSEK/VDX OS, the design and implementation needs to adhere the OSEK/VDX standard, which means the system should be implemented as a multi-threaded software. From the SOFL specification, we find the design and implementation is fairly intuitive. It means SOFL is suitable to construct a requirement specification for an OSEK/VDX application.

After implementation, the last state is verification. In order to completely check OSEK/VDX applications, model checking [9][10] as an exhaustive technique can be applied to verify the OSEK/VDX applications. There exist many model checking methods that have been applied to verify the general multi-threaded software [11] and sequential software [12]. However, these existing model checking methods cannot be directly employed to precisely verify the OSEK/VDX applications, since the execution characteristics of OSEK/VDX ap-

Fig. 1. Function buttons on the control lever of cruise control system. (The figure is from the home page of Audi)

plications are different from the sequential software and general multi-threaded software. In order to apply existing model checking methods to verify the system, we translate the developed cruise control system into a sequential software.

Moreover, based on SOFL specification, we can easily extract the checking properties which are translated into assertions inserted into the translated sequential software. After this, the existing model checkers (e.g., SPIN and CBMC) can be employed to verify the cruise control system.

3 Formal Requirement Specification

3.1 Cruise Control System

Cruise control system is a servomechanism that can maintain a constant vehicle speed as set by the driver. It accomplishes this function by measuring the vehicle speed, comparing it to the set speed, and automatically adjusting the throttle according to a control algorithm. It is usually used for long drives across highways. By using the cruise control system, drivers do not need to control the throttle pedal to maintain the speed of vehicles, which can alleviate the fatigue of drivers. Meanwhile, it can reduce the unnecessary change of speed, which usually results in better fuel efficiency. With these advantages, cruise control system has now been widely equipped in various brands of automobiles, such as BMW, Audi, and Volkswagen.

Cruise control systems developed by different automobile manufacturers usually have different auxiliary functions. Fig. 1 shows control lever of a cruise control system equipped in an Audi automobile. A driver can activate different functions by pressing the function buttons on the control lever. Button ON and OFF are to turn on and turn off the system, respectively. After system turns on, when the button SET is pressed, if the speed of vehicle is within a specific speed interval which is supported by the cruise control system, the system will start to maintain current vehicle speed until a driver presses button OFF, or CANCEL,

or steps on brake. SPEED+ and SPEED- is used to adjust the set speed when system keeps on maintain current vehicle speed. When SPEED+ is pressed, the set speed will be increased, and the system will increase current speed to the set speed and maintain the speed of vehicle at that level. The button CANCEL can temporarily turn off the system, meanwhile, button RESUME can resume the system to the moment at which the system is temporarily turned off.

3.2 Requirement Specification

As our objective is to investigate whether SOFL can be used in developing OSEK/VDX application, rather than develop a fully functional system, for simplicity, we only consider parts of the functions. Moreover, to the consideration of safety, a new CONFIRM button is provided.

After system turns on, the primarily functions required by a cruise control system are as follows (as function button SET in Fig. 1 is not considered in our design, in the following parts of the paper, system turns on means the system stars to maintain current vehicle speed):

1. Let drivers increase and decrease the value of set speed. The set speed is required within a speed interval supported by the cruise control system. To the consideration of safety, a confirm operation is needed to confirm the setting.
2. Keeps on maintaining the vehicle speed at the set value.

Although above specification is very simple, it still may cause misunderstanding. For example, the sentence "a confirm operation is needed to confirm the setting" does not clearly describe what will happen if the confirm operation is not performed. A designer may think that if an operation of increasing or decreasing is not followed by a confirm operation, it will be ignored. While another designer may think the confirm operation is needed only when the driver has finished the setting (maybe through multiple operations of increasing and decreasing). In order to avoid any potential misunderstanding, as we described above, formal notation can help greatly.

3.3 SOFL Specification

A SOFL specification is a hierarchical condition data flow diagram (CDFD) that is linked with a hierarchy of specification modules (**s-modules**) [4]. The CDFD comprises a set of condition processes and describes data flows between them, while the linked s-modules precisely defines the functionality of the components (condition process, data flow, data store) in the CDFD. Each condition process in the CDFD is linked with a **c-process** which is defined in the s-modules and describes functions in terms of **pre** and **post** conditions, within the specific specification context of the module [6]. For more details in SOFL specification, refer [3] [4].

The CDFD of cruise control system is shown in Fig. 2, and the linked s-model is shown in Fig. 3. Each box surrounded by narrow borders in Fig. 2

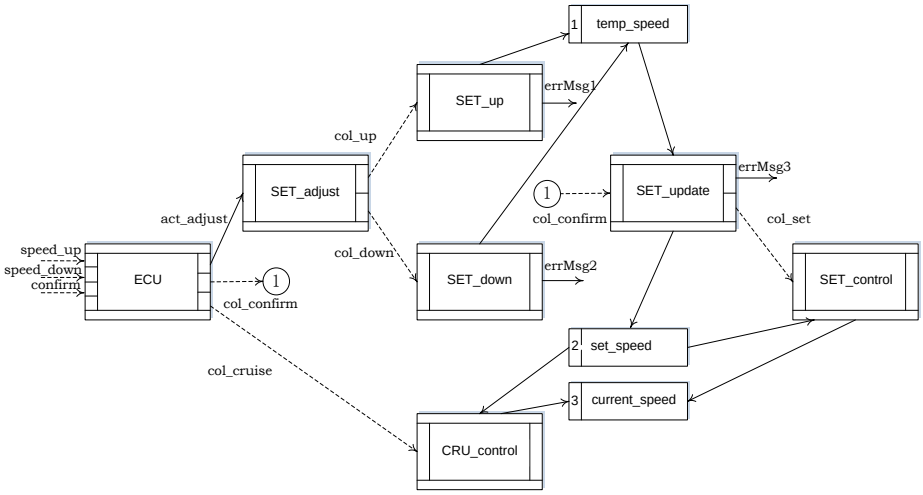

Fig. 2. Condition data flow diagram (CDFD) for cruise control system.

denotes a process, such as `SET_adjust()` and `CRU_control()`, which describes an operation. It tasks inputs and produces outputs. Each directed line with a labeled variable name denotes a data flow. The solid line denotes a active data flow, while, dotted line denotes a control data flow. The box with a number and an identifier (e.g., `temp_speed`) is a data store which can be accessed by processes. A directed line from a data store to a process represents the process can read the data from the store, while, a directed line from a process to a data store means the process can read, write, and update the data in the store. More details about the components used in the CDFD can refer [4] [7].

The cruise control system comprises two primarily functions: set the desired vehicle speed (through increasing or decreasing the set speed) and maintain the vehicle speed at the set value. These two functions are triggered by a `ECU()` (electronic control unit) process. Processes with names star with `SET_` are for the first functions, and processes `CRU_control()` is for the second function. When system is running, process `ECU()` keeps on monitoring the inputs of drivers. Different inputs will trigger different processes to achieve different functions. The selection of `speed_up, speed_down`, or `confirm` denotes the corresponding function button on the system control lever shown in Fig. 1 is pressed by the driver.

When button `speed_up` or `speed_down` is pressed, process `ECU()` then generates a data flow `act_adjust` to indicate which command has actually been selected, and passes this information to process `SET_adjust()`. Based on the value of `act_adjust`, process `SET_adjust()` will trigger either processes `SET_up()` (`speed_up` is selected) or `SET_down()` (`speed_down` is selected). Process `SET_up()` or `SET_down()` first reads the value of `temp_speed` from the data store, and try to update `temp_speed` by increasing or decreasing it with a constant value, re-

spectively. As cruise control system can only run within a speed interval, before updates the data, the process `SET_up()` and `SET_down()` will first check if the updated value of `temp_speed` is within the interval, if not, an error message will be issued. Data `temp_speed` is a temporary data used to react to the process `SET_up()` and `SET_down()`, and only after the process `SET_update()` performs, the data `temp_speed` can be assigned to `set_speed`. After process `SET_up()` or `SET_down()` completes the updates of `temp_speed`, it will sends the completion information to process `SET_update()`. Process `SET_update()` will assign data `temp_speed` to `set_speed` only after the `confirm` button is pressed. After process `SET_update()` assigns data `temp_speed` to `set_speed`, it will trigger process `SET_control()` to control current vehicle speed `current_speed` to the new set speed `set_speed`.

When no function button is pressed by driver, it means the driver does not want to adjust the set speed and wants to maintain current vehicle speed, process `CRU_control()` will be triggered by process `ECU()`. Process `CRU_control()` maintains current vehicle speed `current_speed` based on the value of `set_speed`.

s-module Compared with the specification written in natural language given in section 3.1, the functional abstraction expressed by the CDFD is obviously more comprehensible, especially, the dependency relations among processes can be clearly expressed. However, in order to completely define the CDFD, all the components (conditional process, data flows, data stores) in the CDFD must be precisely defined. To achieve this, the CDFD is linked with a s-model shown as in Fig. 3.

Part **const** shows the constant variables used in the module. All the data flow variables, and data stores in the CDFD are defined in the **var** part. Each of them is defined in a specific data type. Keyword **inv**, stands for invariant, indicates the properties that must be sustained throughout the entire specification. For example, `min_speed <= set_speed <= max_speed` in section **inv** means the setting value of the cruise control system must be larger than the maximum value that supported by the system and less than the minimum value. Function `Controller()` achieves the function of speed control based on a control algorithm. At this level of specification, the control algorithm is not defined.

Process `Init()` is the initial process which performs only one time when the system stars up. We can see that, pre condition defines in the process `Init()` requires that `current_speed` should be less than `max_sp` and larger than `min_sp`. This ensures that the system can star up only when vehicle is running within the speed interval that supported by the cruise control system.

Each processes in the CDFD is linked with a c-process. It describes functions of the processes in terms of pre and post conditions in which predicate logic is adopted. For example, the post condition in c-process `SET_adjust()` means: if the value of data flow variable `act_adjust` is true, process `SET_adjust()` will trigger `SET_up()`, otherwise, `act_adjust` with a false value will make process `SET_down()` be triggered.

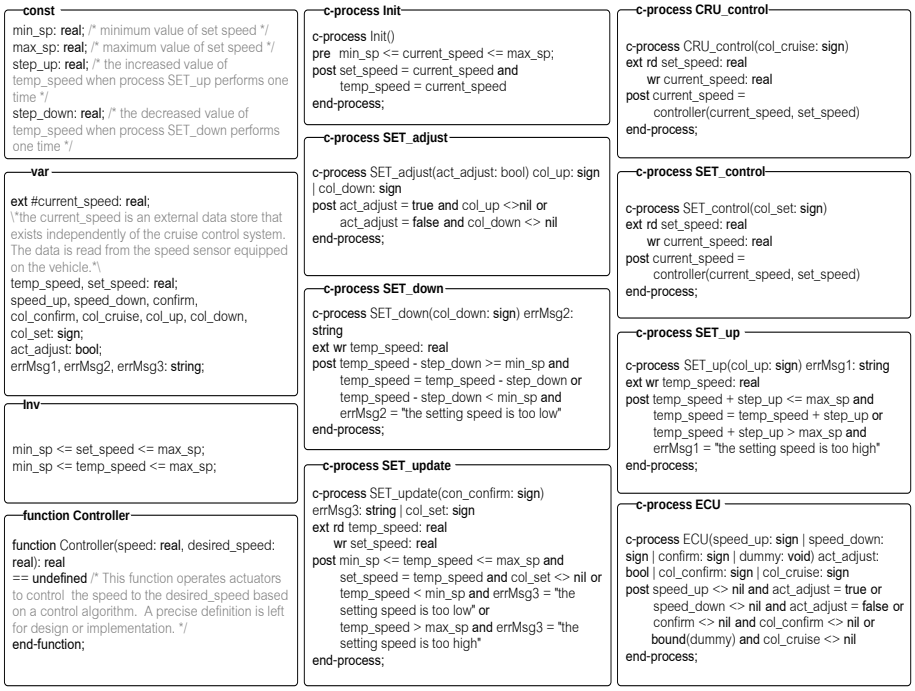

Fig. 3. s-module for cruise control system.

4 Design and Implementation

4.1 Running mechanism

Before start to design and implement the cruise control system as an OSEK/VDX application, we should first obtain a preliminary understanding of the running mechanism of OSEK/VDX OS and its applications.

An OSEK/VDX application is developed as a multi-threaded software running on an OSEK/VDX OS. Tasks (i.e., threads) within the application are concurrently executed and can invoke service APIs to interact with OSEK/VDX OS modules. According to the service APIs invoked by a running task, the corresponding OS modules can dynamically change states of tasks.

Module: A general OSEK/VDX OS is composed of scheduler module, event process module, resource process module, interruption process module, and alarm process module. For simplicity, the cruise control system only interacts with the scheduler module of OSEK/VDX OS. Scheduler module in OSEK/VDX OS adopts static priority scheduling policy. A ready queue, shown in Fig. 4, is maintained to store identifiers of ready tasks. The ready queue is a composition of queues with different priorities. Tasks in the ready queue with the highest priority will be first scheduled to execute. If tasks have the same priority, the scheduler module will schedule these tasks based on first in first out (FIFO).

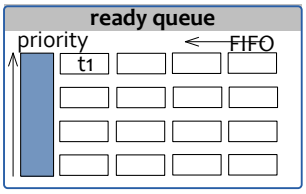

Fig. 4. Ready queue in OSEK/VDX OS.

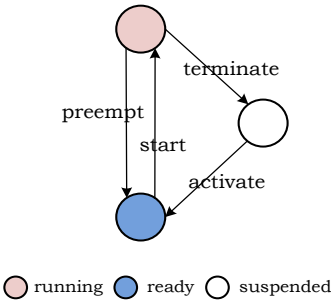

Fig. 5. States switch of basic tasks.

Task states switching: There are two kinds of tasks, *basic task* and *extended tasks*, that can be proceeded in OSEK/VDX OS. The states of a basic task consist of `running` state, `suspended` state, and `ready` state. Compared with the basic tasks, an extended task has an unique state called `waiting` state which are used to interact with event process module. As the cruise control system does not interact with the event process module, tasks in the cruise control system are all defined as basic tasks. Fig. 5 shows the states switch of basic tasks.

Service APIs: Scheduler module can respond to three kinds of API invocation.

- $TerminateTask()$: a running task is moved to `suspended` state.
- $ActivateTask(tid)$: task tid is moved from `suspended` state to `ready` state.
- $ChainTask(tid)$: equivalent to the execution sequence $ActivateTask(tid) + TerminateTask()$.

When one of these three service APIs is invoked by a running task, scheduler module will conduct corresponding operations to respond to the API invocation. This will change tasks' states according to the states switch rules shown in Fig. 5, and may lead to the context switch of tasks.

4.2 Design and Implementation as OSEK/VDX application

From the CDFD of requirement specification, shown in Fig. 2, each conditional process in the CDFD achieves a specific function which is precisely defined in

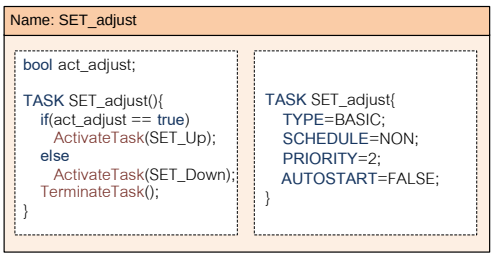

Fig. 6. Task *SET_adjust*().

the corresponding c-process in the linked s-module shown in Fig. 3. Relations between these processes are reflected via data flows (active data flow and control data flow) and data stores. For example, process `SET_adjust()` achieves a *selection* function, that is to active process `SET_up()` or process `SET_down()` based on the value of input active data flow variable `act_adjust`. The relations between these three processes are reflected by control data flow `col_up` and `col_down`.

As an OSEK/VDX application is developed as a multi-threaded software, the intuitive idea is to design each process in the CDFD as a task within the application. Every active data flow variables can be implemented as a variable that can be accessed by the related tasks, and the control data flow variable can be treated as an invocation of service API *ActivateTask*(*tid*), where *tid* is the identifier of the activated task. Based on this idea, the example of task *SET_adjust*() (i.e., the implementation of process `SET_adjust()`) is shown in Fig. 6. It consists of two files: *source code* and *configuration* file. The source code file, show in the left side of the figure, is used to present the concrete behaviors of the application. The configuration file, shown in the right side of the figure, is used to indicate the configuration data of tasks (e.g., priority).

Let's first focus on the source file of task *SET_adjust*(). As shown in Fig. 6, the source file is written in `C++` programming language. The active data flow variable `act_adjust` is defined as a bool variable and that can be accessed by task *SET_adjust*(). The control data flow variables `col_up` and `col_down` are treated as invocations of service APIs *ActivateTask*(*SET_up*) and *ActivateTask*(*SET_down*) respectively, where task *SET_up*() and *SET_down*() are the implementation of process `SET_up()` and `SET_down()` respectively. According to the post condition of process `SET_adjust()` in the s-module, when the value of `act_adjust` is true, process `SET_up()` will be triggered. Reflected in the task *SET_adjust*(), when the variable `act_adjust` is true, the service API *ActivateTask*(*SET_up*) will be invoked, which is to activate task *SET_up*(). The *TerminateTask*() in the last line is to move *SET_adjust*() task itself to the `suspended` state. This service API is invoked when a task has completed his operation.

For the configuration file of a task, there are four items:

- `TYPE`: type of tasks (**BASIC** or **EXTENDED**).

- `AUTOSTART`: to indicate the initial states of tasks when system turns on (**TRUE**: `ready` or **FALSE**: `suspended`).
- `SCHEDULE`: to indicate if the task can be preempted by another ready task with higher priority (**FULL**: can or **NON**: cannot).
- `PRIORITY`: task priority (e.g., 1, 2 ...).

For the type of task, as mention in the last subsection, all the tasks in the cruise control system are defined as basic tasks.

As the setting of `AUTOSTART`, it depends on if the task needs to be activated by another task. Only task that do not need to be activated by another task is set to has `AUTOSTART` as TRUE. Thus, according to CDFD in specification, only task *ECU*() has `AUTOSTART` as TRUE.

For the setting of property `SCHEDULE`, it depends on task's specific running characteristic. For task *ECU*(), when system is running, it keeps on monitoring the input of driver, and activates another tasks. For example, when no function button is pressed, task *ECU*() will activate task *CRU_control*(). It hopes *CRU_control*() can run immediately to achieve the corresponding function. When task *CRU_control*() completes its operation, task *ECU*() goes on activating another task according to the input of driver. If the setting of `SCHEDULE` is NON, the activated task *CRU_control*() can only run after task *ECU*() is terminated. As task *ECU*() keeps on running after system turns on, it means task *CRU_control*() will never run, which is an obviously wrong setting. Thus, the setting of `SCHEDULE` for task *ECU*() should be FULL.

For other tasks, e.g., *SET_adjust*(), it activates task *SET_up*() or *SET_down*(). Then, it will be terminated by invoking service API *TerminateTask*(). To implement this running characteristic, both setting of property `SCHEDULE` can achieve the requirement. As a FULL setting of `SCHEDULE` may lead to more frequent context switch of tasks, we set `SCHEDULE` of all the tasks except task *ECU*() as NON.

For the setting of `PRIORITY`, from the running characteristic of task *ECU*() described above, we can see when a task is activated, it should preempt task *ECU*(). Thus, the priority of task *ECU*() should be the lowest among all the tasks. For other tasks, as the setting of `SCHEDULE` is NON, any settings of `PRIORITY` that higher than the setting of *ECU*() are reasonable.

Based on the ideas described above, based on the SOFL specification, we can implement the cruise control system as an OSEK/VDX application. The complete implementation is shown in Fig. 7.

5 Simulation and Verification

To completely check OSEK/VDX applications, we employ model checking to verify the developed application. As mentioned in section 2, the existing model checking methods cannot be directly employed to precisely verify the OSEK/VDX applications, since the execution characteristics of OSEK/VDX applications are different from the sequential software and general multi-threaded software.

As described in section 4, an OSEK/VDX application is developed as a multi-threaded software. When it runs on OSEK/VDX OS, the executions of tasks

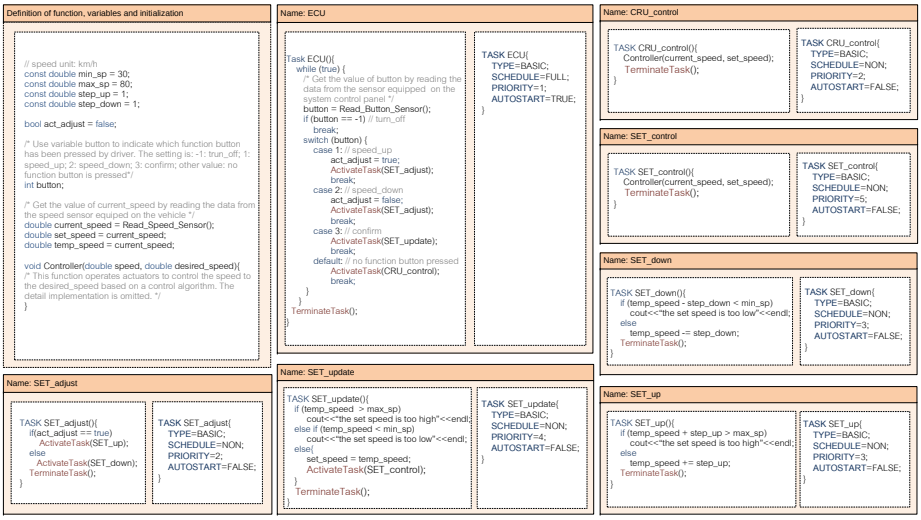

Fig. 7. Implementation of cruise control system as an OSEK/VDX application.

within the application are dispatched by scheduler module, and the running task is explicitly determined by the scheduler according to the task priorities and configuration data. Moreover, tasks can invoke service APIs supported by OSEK/VDX OS to dynamically change the states of tasks defined in the application, and the changed states will affect the scheduling of tasks.

On the one hand, if we directly apply the existing model checking methods for general multi-threaded software to verify OSEK/VDX applications, it is too imprecise because a lot of unnecessary interleavings of tasks will be checked by existing methods, and these unnecessary interleavings may result in a spurious bug in the verification. This is because, in the existing works for the general multi-threaded software (e.g., systemC programs), since the running thread cannot be explicitly determined, all of the possible interleavings of runnable threads are taken into account in the verification in order to completely check the target software. However, in OSEK/VDX applications, the running task is explicitly determined by OSEK/VDX scheduler. The difference of scheduling policy between the general multi-threaded software and OSEK/VDX applications makes it is not unsuitable to employ existing model checking methods for the general multi-threaded software to check OSEK/VDX applications.

On the other hand, if we want to employ the model checking methods for sequential software to verify OSEK/VDX applications, the developed target application has to be translated into sequential software in advance. To achieve this, a sequentialization approach proposed in our previous work [5] can be applied.

5.1 Sequentialization

The key idea of the sequentialization approach is to use a directed graph to represent the sequential program of an OSEK/VDX application. To construct the directed graph, a directed graph constructor as a simulator has been developed. A simplified OSEK/VDX OS model is included in the constructor to respond to the invoked service APIs. A tool named `autoC` has also been developed based on this approach. More details about the approach can be found in [5].

Through the sequentialization approach, based on the implementation shown in Fig. 7, the directed graph for cruise control system is shown in Fig. 8. The definition of directed graph is as follows.

DEFINITION: The directed graph is a tuple $\mathcal{G} = (V, v_0, v_e, E, L)$. V is the set of nodes, and a node $v \in V$ is a tuple $v = (pcs, osd)$, $pcs = [n^1, \ldots, n^m]$ is a array used to record the current locations of tasks $t_1, \ldots, t_m$ (m is the number of tasks), *osd* is the set of values used to store the data within D of OS model, where $D = \{runTask, readyQueue, suspendList\}$ is the set of data structures used to store the states of tasks. In the data structure set D, $runTask$ which is a variable is used to store the *tid* of running task (*tid* is task identifier). The *readyQueue* is composed of queues with different priorities and used to store the *tids* of tasks in the ready state. The data structures *suspendList* are used to store the *tids* of tasks in the suspended state. $v_0 \in V$ is the start node, $v_e \in V$ is the end node. $E \subseteq V \times V$ is the set of directed edges. $L : E \to \bigcup \Sigma^{tid}$ is the labeling function from an edge $(v, v') \in E$ to a task statement $\alpha \in \bigcup \Sigma^{tid}$, where *tid* is the identifier of tasks, Σ^{tid} is the set of statements of tasks *tid*, the expression of a statement $\alpha \in \Sigma$ is as follow:

$\alpha ::=$ *condition* | *assignment* | *goto* | *assertion* | `API`

Note that, the invocation of service APIs are replaced as *goto* statement in Fig. 8.

5.2 Verification

Before verify the program, we should first extract checking properties from the requirement specification. We can easily extract checking properties from two parts of the SOFL specification. The first part is the pre and post condition of a c-process in s-module. A pre (post) condition in a c-process can be translated as assertions, and inserted into the directed graph at the locations before (after) the execution of the corresponding task.

For example, from the post condition of process `SET_adjust()`, we can know that task *SET_adjust*() will activate task *SET_up* if the value of `act_adjust` is true, otherwise, it will activate task *SET_down*. Because each step of constructing the directed graph, the data within D which stores the states of tasks has been translated into the directed graph. This makes we can know which task is running (in `running` state), and can also check whether a task is activated (in `ready` state) or not (in `suspended` state) at any node of the directed graph. Through searching the directed graph, we can know task *SET_adjust*() runs following nodes $\langle v_{27}, v_{28}, v_{29}, v_{30} \rangle$ (`act_adjust` is true) or $\langle v_{27}, v_{35}, v_{36}, v_{37} \rangle$ (`act_adjust`

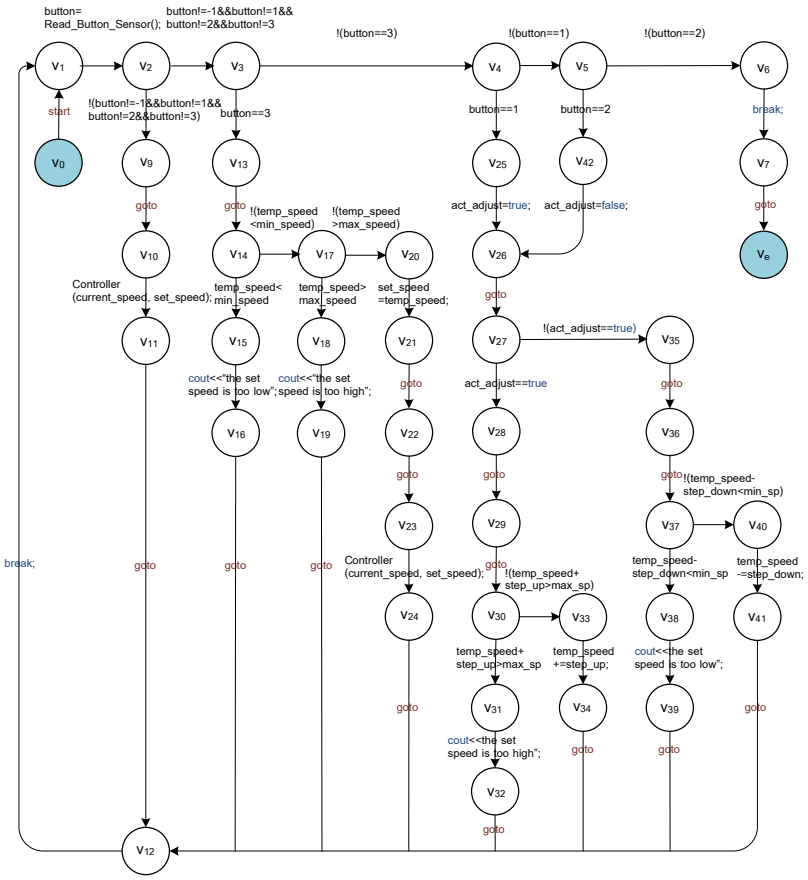

Fig. 8. Directed graph for cruise control system.

is false). Thus, we can insert $assert$(SET_up@$readyQueue$) after node v_{30} and $assert$(SET_down@$readyQueue$) after node v_{37}, where tid@$readyQueue$ means task tid is in the ready queue.

The second part is the part inv in s-module. Part inv indicates the properties that must be sustained throughout the entire specification. Thus the translated assertion can be inserted into any locations from node v_0 to node v_e of the directed graph. Specific to the s-module shown in Fig. 3, from the inv part, we can get four assertions: $assert$(`set_speed <= max_speed`), $assert$(`set_speed >= min_speed`), $assert$(`temp_speed <= max_speed`), and $assert$(`temp_speed >= min_speed`). These assertions can be inserted into any locations from node v_0 to v_e. Note, assign a value in interval [`min_speed`, `max_speed`] to `current_speed` is needed to check these assertions.

After inserts the assertions, as cruise control system has been translated into a sequential program, now we can employ existing model checkers (e.g., SPIN,

CBMC) to verify the program. The checking results indicate the application satisfies the checking properties discussed above.

6 Concluding Remarks

Through developing the cruise control system, we can see that SOFL specification can effectively help developer to develop an OSEK/VDX application throughout the development process. SOFL requirement specification can precisely defines the behavior of the software. From the specification, the design and implementation of cruise control system is fairly intuitive. A developer can easily develop and implement an OSEK/VDX application based on SOFL specification. Moreover, checking properties can be easily extracted from SOFL specification, which helps a lot for employing model checking technique to verify the developed applications.

For the future work, an important direction is to develop a tool to automatically generate assertions from SOFL specification and insert them into the translated directed graph.

References

1. OSEK/VDX Group, *OSEK/VDX Operating System Specification 2.2.3*. http://portal.osek-vdx.org/
2. J. Lemieux, *Programming in the OSEK/VDX Environment*. CRC Press, 2001.
3. S. Liu, *Formal Engineering for Industrial Software Development using the SOFL method*. Springer-Verlag, 2004.
4. S. Liu, A.J. Offutt, C. H.-Stuart, Y. Sun, and M. Ohba, "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Trans. Softw. Eng.*, vol. 24, no. 1, pp. 24–45, Jan. 1998.
5. H. Zhang, T. Aoki, and Y. Chiba, "Yes! You Can Use Your Model Checker to Verify OSEK/VDX Applications," *Proc. 8th IEEE International Conference on Software Testing, Verification and Validation*, pp. 1–10, Apr. 2015.
6. X. Wang and S. Liu, " An Approach to Declaring Data Types for Formal Specifications," *Proc. 3rd International Workshop on SOFL+MSVL*, LNCS, vol. 8332, pp. 135-153, Springer-Verlag, 2014,
7. C. H.-Stuart and S. Liu, "An Operational Semantics for SOFL," *Proc. Asia-Pacific Software Engineering Conference*, pp. 52–61, Dec. 1997.
8. H. Zhang, T. Aoki, H.-H. Lin, M. Zhang, Y. Chiba, and K. Yatake, "SMT-Based Bounded Model Checking for OSEK/VDX Applications," *Proc. Asia-Pacific Software Engineering Conference*, pp. 307–314, Dec. 2013.
9. E.M. Clarke, O. Grumberg, and D.E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.
10. E.M. Clarke, E.A. Emerson, and J. Sifakis, "Model Checking: Algorithmic Verification and Debugging," *Commun. ACM*, vol. 152, no. 11, pp. 74–84, Nov. 2009.
11. S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," *TACAS 2005*, LNCS, vol. 3340, pp. 93–107, 2005.
12. Z. Yang, C. Wang, and A. Gupta, "Model Checking Sequential Software Programs Via Mixed Symbolic Analysis," *CM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–26, Jan. 2009.