Formal Construction and Verification of Home Service Robots: A Case Study

Moonzoo Kim and Kyo Chul Kang

CSE Dept. Pohang University of Science and Technology,
Pohang, South Korea
{moonzoo,kck}@postech.ac.kr

Abstract. Home service robots have attracted much attentions to anticipate improved quality of human life. Considering that malfunctions of home service robots can directly threat the safety of human users, the assurance of robot's safe operation is a crucial prerequisite for the wide deployment of home service robots. Current practice of robot development, however, often fails to satisfy this requirement. Robot developers tend to concentrate on technical components only and fail to consider how these components will integrate to create the service. This practice frequently causes *feature interaction problems*. Furthermore, reactive nature of the robot applications adds to further complexity. Traditional testing is unsuccessful with this setting due to the difficulty of testing embedded systems and uncertainty caused by sensor devices. These situations make *formal construction and verification* essential to ensure safe operation of home service robots.

In this paper, we present our experience of formally constructing and verifying the core of Samsung Home Robot (SHR) with the use of Esterel. First, we reverse-engineered SHR to identify and analyze the core of SHR. Then, we re-implemented the core part in Esterel and verified SHR to satisfy safety properties regarding stopping behaviors through model checking. Through the verification, we detected and solved a feature interaction problem which caused the robot to ignore a stop command.

1 Introduction

With the advances of robotics, computer science, and other related areas, home service robots have received a strong academic and industrial attention. It is because home service robots can increase a quality of human life in a wide range of application areas. Thus, those leading companies such as Sony [3], Honda [2], and Samsung have invested a great deal of efforts in developing home service robots. Home service robots utilize various technology-intensive components such as vision recognizer, speech processors, and actuators to offer services. Thus, robot applications should coordinate these components in harmony. Robot developers, however, tend to focus on technical components at an early stage of product development without any consideration of how they will integrate these components to provide services. In addition, these components are developed by

D.A. Peled and Y.-K. Tsay (Eds.): ATVA 2005, LNCS 3707, pp. 429–443, 2005.

separated teams, which makes integration of these components more difficult. As a result, robot products often suffer from feature interaction problems [9,17]. Furthermore, reactive nature of home service robots adds further complexity to robot applications. Therefore, it is a highly challenging task to develop robot applications satisfying stringent temporal and safety requirements.

Due to high complexity of robot applications, testing and debugging often takes more than a half of total development time but still fails to provide satisfying result. Thus, the necessity of *formal validation and verification (V&V)* has been recognized in robotics areas [11,18]. Also, robot domain specific V&V frameworks such as ORCAAD [7] and MAESTRO [8] have been developed. In robot industry, however, a practice of applying formal methods is not very popular because robot industry does not have enough field experiences with formal methods yet. In addition, the gap between a formal model and a real implementation discourages developers from adopting formal methods as well. Therefore, we need to apply a *unified formal framework* supporting both *construction* and *verification* of robots. Furthermore, for practicality, we should aim to apply formal methods to the *core* of relatively a small size with an acceptable development overhead, rather than to the whole applications [12].

In this paper, we describe our experience of formally constructing and verifying Samsung Home Robot (SHR) with Samsung Advanced Institute of Technology (SAIT). First, we reverse-engineered SHR application that SAIT had developed. Based on the extracted architectural information, we re-engineered SHR application while identifying the core of the application. Then, we re-implemented the core in Esterel [6] and verified that SHR satisfied safety properties regarding stopping behaviors through model checking. Through the verification, we detected and solved a feature interaction problem which caused the robot not to stop when a user commanded the robot to stop. [1]

Section 2 describes background of SHR. Section 3 overviews the Esterel framework. Section 4 illustrates the previous SHR application and re-engineered one. Section 5 shows the verification results about stopping behaviors of SHR. Finally, Section 6 concludes with the summary of this paper.

2 Background of SHR100

Sect. 2.1 gives an overview of the SHR project and Sect. 2.2 explains the services which SHR provides. Sect. 2.3 describes statistics on the SHR application code.

2.1 SHR Project

We have developed three versions of SHR - SHR00, SHR50, and SHR100. The development of SHR00 started in 2002 by four separate teams of SAIT consisting

[1] We used the Esterel framework for this project mainly because we can verify an Esterel program by model checking and generate a C code from the verified Esterel program. This unified framework is suitable for industrial projects, which require a reliable working code as a final result with minimal overhead.

of thirteen people working on speech recognition, vision recognition, simultaneous localization and mapping (SLAM), and actuator control. SHR50 as well as SHR00, however, often experienced unstable behaviors such as missing user's commands and showed stuttered movement even though each part worked successfully before the integration (this kind of failure is not uncommon in robotics field [9]). After ten months into the new development of SHR100, for higher reliability, SAIT requested POSTECH to re-engineer SHR100 supporting "call and come" and "user following" services (see Sect. 2.2). With this request, POSTECH re-engineered an existing implementation for six months. The overview of the SHR100 components is illustrated in Fig. 1.

Fig. 1. HW Components of SHR100

2.2 Services of SHR100

Some of the primary services of SHR100 are described as follows.

- Call and Come (CC)
  There are two commands: namely "come" and "stop". Once a "come" command is recognized, the robot tries to detect the direction of sound source by comparing the strength of sound captured via microphones. Then, the robot rotates to the direction of sound source and tries to recognize user's face by analyzing images captured through the front camera. If the caller's face is detected, the robot moves forward until it reaches within one meter from the user. A "Stop" command makes the robot stop. CC is *preemptible*, i.e., while CC is executed, newly recognized command makes the robot ignore the previous command and allow to proceed the new one.
- User Following (UF)
  This service is triggered right after CC is completed. Once UF is triggered, the robot constantly checks vision data and data from the structured light sensor which is used to locate the user. The robot tracks down the user within the distance of one meter range. If the robot misses a user, the robot

notifies the user by speaking "I lost you" and UF ends. Similar to CC, UF is a preemptible service.
- Tele-presence (TP)
  A remote user can control a robot by using a PDA. In addition, the robot can send images obtained from the front camera to the PDA for surveillance purpose through a home server.
- Security Monitoring (SM)
  The robot patrols around a house using the map generated by SLAM component for surveillance. When accidents are detected, the robot reports to the user.

2.3 SHR100 Application Statistics

The rough statistic summary of the SHR100 application is described in Table 1. Some parts of the application (mostly controller parts) were given to POSTECH as source code in C/C++ while other parts (mostly recognition algorithms and device drivers) were given as binary libraries.

Table 1. Statistics on the SHR100 application

Components	# of files	Size
Call and come	29	4000 lines
User following	43	9000 lines
Others	43	3600 lines
Libraries	39	38 MB

3 The Esterel Framework

In this section, we briefly describe the Esterel language and its toolset.

3.1 The Esterel Language

Esterel [6] is a language for programming reactive systems that wait for a set of inputs, and react to these inputs by computing and producing outputs. Since Esterel is based on the "synchrony hypothesis", every reaction to a set of inputs should be instantaneous. In practice, this means that a system should react to input signals before input signals of the next cycle arrive. Synchrony hypothesis considerably simplifies the specifications of reactive systems. Furthermore, many application areas satisfy this hypothesis.

A program written in Esterel specifies the components (called modules) running in parallel. Modules communicate with each other and the outside world through input/output signals, which are broadcasted and may carry values of arbitrary types. Thus, the interaction between components can be clearly described. Furthermore, Esterel provides reactive/preemptive operators which are

useful for developing robot applications. An Esterel program has its semantics as a finite state Mealy machine whose transitions are labeled with pairs of input and output signals (see Fig 6).

3.2 The Esterel Toolset

Esterel toolset consists of the following three components. [2]

- Esterel compiler `esterel`
- graphical simulator `xes`
- model checker `xeve`

`esterel` compiles an Esterel program into various formats including the C language. Using `esterel`, once a developer has proved the correctness of an Esterel program through model checking, one can generate correct C code without a manual conversion. This WYPIWYE (What You Prove Is What You Execute) principle is a strong advantage of Esterel over other formal modeling languages. In addition, an Esterel program can be seamlessly integrated with existing C/C++ codes through well-defined APIs. Furthermore, generated C code is platform neutral so that a developer can port an Esterel program into different OS/HW platforms (e.g. Linux or VxWorks) without a difficulty.

`xes` supports interactive simulation as well as guided simulation. With a given Esterel program, a user can execute the program by symbolically selecting input signals to emit and advancing its ticks (time instants). `xes` is also used to examine the execution trace of a counter example generated from `xeve`.

`xeve` minimizes and analyzes a finite state machine generated from an Esterel program. Basic verification process of `xeve` is to check the presence of output signals with given configuration of input signals by model checking. A simple property such as "if a user does not give a command to a robot, the robot must not move" (see Sect. 5.2) can be checked in this way. More complex property can be checked by building an observer module which emits a violation signal when the property is violated (see Sect. 5.3). [15] proved that safety properties described in temporal logic could be translated into observer modules in Esterel.

4 Re-engineering of SHR100

In this section, we describe both previous SHR100 implementation (Sect. 4.1) and re-engineered SHR100 implementation (Sect. 4.2).

4.1 Previous SHR100 Implementation

SHR100 was implemented in a service-oriented way because each service feature such as CC and UF, had been developed separately. Consequently, operations

[2] A commercial Esterel studio [1] provides an integrated development environment including a visual language editor.

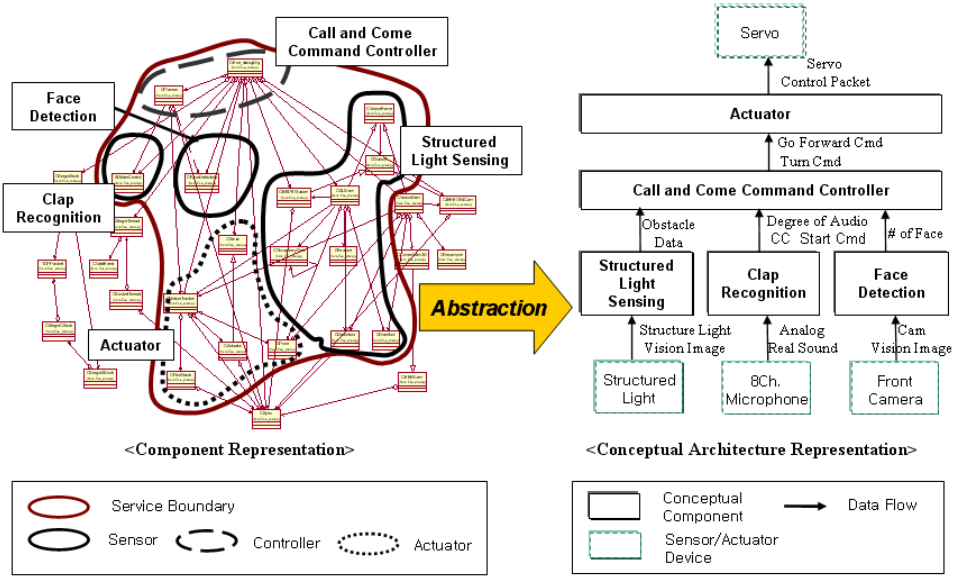

Fig. 2. Previous architecture of SHR100 regarding the CC service

of a service were dispersed among components, which does not clarify component architecture design. In addition, there existed redundant computational components because some computational components (e.g. vision) were used by different services. Therefore, developers experienced difficulty in identifying the interactions among the components of its original implementation, which often caused feature interaction problems. In order to improve reliability, we needed to clarify the previous SHR100 architecture first. Otherwise, it would be difficult to identify and analyze the core. Also, re-writing the core of SHR100 in Esterel would be messy because the new core should cooperate with existing C/C++ components.

Fig. 2 illustrates the recovery of a conceptual architecture from the object relationship diagram of CC through abstraction. The left part of Fig. 2 describes CC service unit and its constituent operational units. Using functional cohesion as a criterion, we classified operational units into three categories - *sensor (input), controller (coordination)*, and *actuator (output)*. Then, we identified five operational units -"Face Detection", "Clap Recognition", "SL Sensing", "CC Command Controller", and "Actuator". After a data flow analysis, these units are configured into the conceptual architecture depicted in the right part of Fig. 2. We found out that "CC Command Controller" unit, which consists of `CCallComeDlg` and `CPlanner` classes, serves as the core of CC service by receiving data from the sensor units and making decisions to the actuator unit. Also, we found that the core was executed fast enough to satisfy the synchrony hypothesis required by the Esterel framework (see Sect. 3.1).

Through the re-engineering process, several bugs in the original implementation were found. For example, a main control function of the CC service is `void CCallComeDlg ::processState()` as in Fig. 3. `processState()` is called periodically once in every 100 milliseconds. Given a command, CC executes the command through multiple sequential steps. Each step is represented by a corresponding case statement block and is identified by the value of `m_order` declared at line 2. At the end of each case statement block, `m_order` is updated to determine the next step. After one step is executed, `processState()` is terminated and is called again after 100 milliseconds. If a new command is given between these two adjacent invocations, a previous command is ignored and the new command is processed.

```
01:class CCallComeDlg {
02:    int m_order;
03:    ...
04:    void processState() {
05:        ...
06:        switch(m_order) {
07:            case 0: STOP();
08:                    m_order++;
09:                    break;
10:            case 1: ROTATE();
11:                    m_order++;
12:                    break;
13:            case 2: static int nCount = 0;
14:                    if (abs(m_bef0-cur0)==0) nCount++;
15:                        else nCount = 0;
16:                    if (nCount > 2) m_order++;
17:                    break;
18:            ...
19:            case 9: CC_DONE();
20:                    m_order = -1;
21:                    break;
22:}   }   }
```

Fig. 3. A main control procedure for the CC service in C++

This pattern of reactive programming is a straight-forward way to allow preemption in C++, and is found frequently in robot applications. This pattern of reactive programming is, however, error-prone. For example, at line 16, `nCount` is used to test *twice* whether SHR100 stops its rotation or not. However, testing may happen only once because `nCount` is declared as a *static* local variable at line 13 and can be greater than two all the time without re-initialization. This error decreases the accuracy of user recognition due to blurred images captured while the robot does not stop its rotation completely. As a number of possible cases increase by adding more features, the complexity of C/C++ code increases

rapidly so that developers can hardly manage and debug the program. Note that Esterel prevents such errors by handling a preemptive event e with preemption operator `every` e `do` *statements* `end every` (see line 11 to line 24 in Fig. 5).

4.2 New SHR100 Implementation

The architecture model in Fig. 2 is not adequate for multiple services; it does not provide the coordination of multiple service controllers (e.g. CC controller and UF controller) to handle interaction among services. Furthermore, the complexity of interactions among services grows exponentially within the previous architecture due to spaghetti-like communications among the components. Therefore, based on the extracted conceptual architectures, we re-designed the architecture of SHR100 concerning with handling issues such as priorities among services or global system modes (for more details on the re-engineering process of SHR100, see [14]). We separated *control plane* containing control components from *data plane* containing computational components. Firstly, we could easily identify four separate control components (CC, UF, TP, and SM) which specify their own behaviors for corresponding services. In addition, we defined Mode Manager to control global behaviors (e.g. initialization and interaction policy) of the robot by receiving all up-stream events from the computational components and managing control components. Each of these control components was implemented as a separate Esterel module. [3] Secondly, after data flow analysis, we could come up with five computational components - SLAM, Navigation, User Interface, Vision Manager, and Audio Manager. Fig. 4 describes the new software architecture.

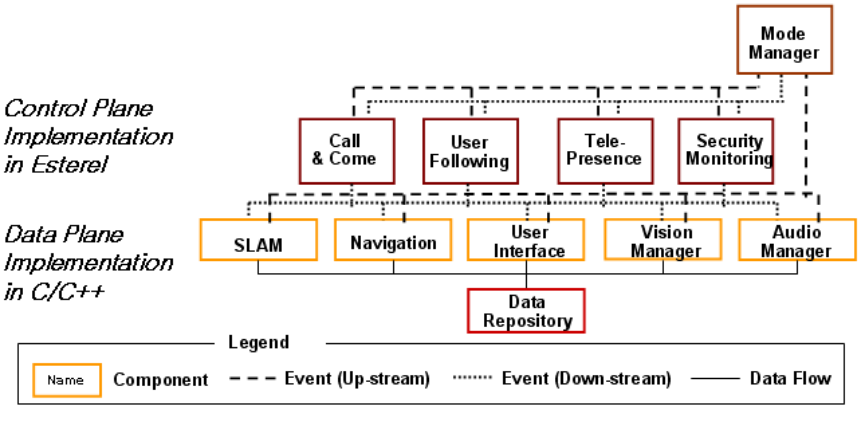

Fig. 4. New SHR100 architecture

[3] The size of the Esterel program is around 200 lines. A generated C code from the Esterel program is around 1700 lines. Memory usage and execution speed of the new implementation does not show observable difference from the previous one.

Fig. 5 is a skeleton of the re-implemented CC service in Esterel. A module `control_plane` (line 1 to line 5) represents a whole control application including the Mode Manager `mm`, the CC service `cc`, the UF service `uf`, and so on (see line 4). Communication among those modules is implemented by using input/output signals declared at line 2 and line 3 (note that the output signals in Fig. 5 invoke the C functions of the same names shown in Fig. 3). `COME_CMD` and `STOP_CMD` are input signals corresponding to the "come" and "stop" commands. A "come" command is handled from line 14 to line 18 and a "stop" command is handled from line 19 to line 21. A task of rotating SHR100 toward the user and detecting the user is implemented as a submodule `rot_det` and is executed at line 16.

```
01:module control_plane:  % Control software
02:    input COME_CMD, STOP_CMD, ...
03:    output STOP, ROTATE, GO, CC_DONE, UF_DONE,...
04:    run mm || run cc || run uf || run tp || run sm ...
05:end module
06:
07:module cc:       % Call and Come service
08:    input COME_CMD, STOP_CMD;
09:    output STOP,ROTATE,GO,CC_DONE,...
10:    signal Reset in
11:        every immediate [COME_CMD or STOP_CMD] do
12:            weak abort
13:                present
14:                case COME_CMD do   % come command
15:                    emit STOP; pause;
16:                    run rot_det;
17:                    ...
18:                    emit CC_DONE;pause;
19:                case STOP_CMD do   % stop command
20:                    emit STOP;
21:                    emit CC_DONE;pause;
22:                end present;
23:            when Reset;
24:        end every
25:    end signal
26:end module
27:...
```

Fig. 5. Skeleton Esterel code for the CC service

As we have seen, the new implementation defines components concretely using modules/submodules. In addition to this, the new implementation makes interaction visible among the components by using explicit communication mechanisms such as input/output signals. These features assign responsibility for the behaviors to these components clearly and it helps to analyze feature interaction problems.

5 Formal Verification of SHR100 Movement

There are various safety properties to assure SHR100's correct operation. In this project, we concentrate on the most critical safety properties, which are about movements that may cause crash with any obstacles (e.g. furniture or users). There can be many causes for collision such as obstacle recognition failure, HW failure to signal actuator, etc. We focused on, however, discrete SW controller which we re-wrote in Esterel. *Stopping behavior* of SHR100 is the first target to verify. Considering that SHR100 can move upto 2 m/s (=7.2 km/h), these properties should be checked carefully for user's safety.

In this section, we describe safety properties P_1, P_2, and P_3 regarding stopping behavior of SHR100. We describe the most primitive safety property P_1 first, then incrementally refine P_1 into P_2 and P_3. Sect. 5.1 describes verification preliminary for SHR100. Sect. 5.2 and Sect. 5.3 illustrate verification of SHR100 running the CC service only. Sect. 5.4 describes verification of SHR100 running CC and UF concurrently. Sect. 5.5 summarizes the verification results. [4]

5.1 Verification Preliminary for SHR100

We used the `xeve` model checker [4] to verify safety properties. First, `xeve` performs bisimulation minimization on FSM which is generated from an Esterel program. Then, a user selects input signals as "always present", "always absent", or "having any value". In addition, a user can specify exclusion relation among input signals (e.g. `COME_CMD` and `STOP_CMD` cannot present at the same time). Finally, the user selects an output signal to check if it can be emitted with given configuration of input signals. Simple properties (e.g. P_1 in Sect. 5.2) can be checked easily in this way. More complex properties (e.g. P_2 in Sect. 5.3 and P_3 in Sect. 5.4) can be checked by building an observer which emits a violation signal when given properties are violated.

Basically, safety properties on stopping behavior can be described using bounded-response formula [5] in temporal logic [16]

$$\Box(C_{stop} \rightarrow \Diamond_d stop)$$

where C_{stop} and *stop* stand for `STOP_CMD` and `STOP` signals in the Esterel implementation. The actual safety properties for robot application, however, are more complex. First, when `STOP` is emitted, `GO` or `ROTATE` must *not* be emitted without any new command. In other words, we also need to check signals *nullifying* `STOP` such as `GO` and `ROTATE`. In addition, we have to check whether output signals are emitted arbitrarily regardless of input signals. For example, if a user does *not* give a command to the robot, the robot must *not* move at all. We could describe a safety property in temporal logic and then translate the property into an Esterel observer following the guideline of [15]. This generates, however, a unnecessarily complex observer. Thus, we developed an observer directly in Esterel without describing and translating a temporal logic property.

[4] Preliminary verification results are from [13].

5.2 Verification of the CC Service Without an Observer

First, we checked the CC service without other services. Consider the following property P_1.

> P_1 : *If a user does not give a command to the robot, the robot must not move.*

Although P_1 looks obvious, this requirement is important to ensure a safe operation of the robot. Violation of P_1 may lead the robot to move autonomously without a user's command and as a result, it can cause damage to house appliances or accidently hurt a man. Furthermore, guaranteeing satisfaction of P_1 is a difficult task without model checking because a developer has to find out all the possible test cases [10].

There are only two output signals to make the robot move - `GO` and `ROTATE`. We checked if the CC service satisfied P_1 by setting `COME_CMD` and `STOP_CMD` as "always absent" and selecting `GO` as an output signal to check. Then, `xeve` showed that `GO` is *never* emitted by the robot. In the same way, `xeve` showed that `ROTATE` is never emitted, either. Thus, we concluded that the CC service satisfied P_1.

Slightly refined property P_1' can be described as follows.

> P_1' : *If a user does not give a "come" command, but may give a "stop" command to the robot, the robot does not move.*

We can verify that the CC service satisfies P_1' too.

Using a FSM visualization tool `atg`, we could explore the FSM of the new SHR100 implementation written in Esterel. This FSM exploration helps understand global behavior of SHR100. For example, Fig. 6 depicts the behavior of the CC service. Each transition is labeled with a pair of input/output signals. A present input signal has a prefix `?` and an input signal which is not present has a prefix `#`. A present output signal has a prefix `!`. An initial state (a doubly circled state at the top left corner) of the CC service has only two outgoing transitions.

- a self transition α :`#CM.#ST + #CM.?ST.!STOP.!CC_DONE` [5]
- a transition β (going to a lower state) :`?CM.!STOP`

The first half of α transition (`#CM.#ST`) indicates that SHR100 does not move without any command, which corresponds to the verification result on P_1. The second half of α transition means that "stop" commands alone do not make the robot move, which corresponds to the verification result on P_1'. Once a "come" command is given, SHR100 takes β transition and traverses the FSM.

[5] To increase readability of Fig. 6, we use shorthand notations `CM` and `ST` for `COME_CMD` and `STOP_CMD` respectively.

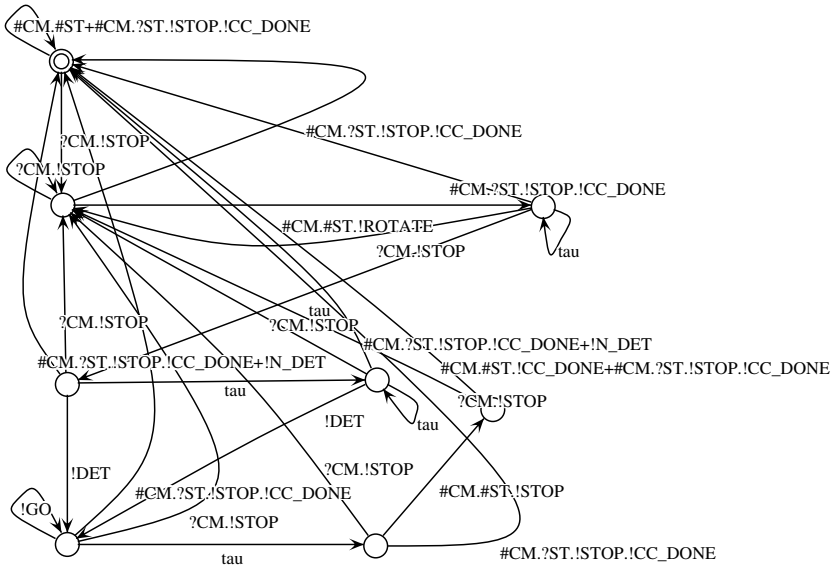

Fig. 6. A FSM for the CC service

5.3 Verification of the CC Service Using an Observer

Consider the property P_2 as below.

> P_2 : *If a user gives a "stop" command, the robot stops and does not move without any new command.*

To verify P_2, we built an observer as in Fig. 7. We incorporated `observer` with the `cc` module in parallel. `observer` emits `STOP_VIOLATION` at line 9 and line 13 if P_2 is violated. If a "stop" command is given (line 5) and the robot stops immediately (line 6), then `observer` keeps its watch if the robot rotates or moves forward (line 7 to line 11) unless any new command is given by the user. We verified that `STOP_VIOLATION` is never emitted.

5.4 Verification of the Concurrent CC and UF Services

We checked if the control software which consists of the CC and UF services satisfied P_1 and P_1'. We showed that the control software satisfied P_1, but surprisingly not P_1'. The verification result on P_1' claimed that `ROTATE` and `GO` could be possibly emitted when `COME_CMD` was absent and `STOP_CMD` might be given. In general, verification result from `xeve` is sound but *not* complete because a FSM is generated from an Esterel program without evaluating expressions. [6] Therefore, a user has to check whether a violation is a real one or a false alarm.

[6] `xeve` ignores external C functions as well because the expressions containing return values of external C functions are ignored anyway.

```
01:module observer: % Observer for detecting safety violation
02:    input STOP_CMD, COME_CMD, ROTATE, STOP, GO;
03:    output STOP_VIOLATION;
04:    weak abort
05:        every immediate STOP_CMD do
06:            present STOP then
07:            loop
08:                present [ROTATE or GO]
09:                then emit STOP_VIOLATION;
10:                end present;pause;
11:            end loop;
12:            end present
13:            emit STOP_VIOLATION;
14:        end every
15:    when COME_CMD;
16:end module
```

Fig. 7. An observer for detecting violation of P_2

Through simulations displaying interactions between the CC and UF components, we could figure out that UF made the robot rotate and move forward when a "stop" command was given; the violation was a real one. This violation occurred because UF was triggered by `CC_DONE` which was emitted by CC when a "come" command or a "stop" command was successfully processed (see Sect. 2.2 and line 18/line 21 of Fig. 5). UF should have been triggered only after a "come" command was processed, not after a "stop" command was processed. Thus, we refined `CC_DONE` into `CC_COME_DONE` and `CC_STOP_DONE`. Then, we modified the UF implementation so that only `CC_COME_DONE` could invoke UF. After this modification, the concurrent CC and UF services satisfied P_1'.

SAIT had not find this feature interaction problem previously. During UF service, SHR100 does not move unless it succeeds to detect the user. While the user was testing SHR100, he did not intend to be detected by the robot when he gave a "stop" command because he expected the robot to stop, not to start UF service. Thus, the user was usually outside the vision area of SHR100 when he gave a "stop" command. In addition, due to uncertainty of the vision recognizer (e.g. low accuracy in a dark room or with strong light), SHR100 often misses the user. When the robot fails to detect the user, it should report to the user by synthetic voice. It happened, however, that voice synthesis sometimes did not work when running with other components. Therefore, without thorough testing, SAIT simply thought that SHR100 stopped accordingly to a given "stop" command and missed the problem.

We checked if P_2 was satisfied by the revised control software running CC and UF concurrently. We used `observer` in Fig. 7 without modification and verified that the control software satisfied P_2.

Furthermore, we refined P_2 into P_3 by adding a real-time constraint.

P_3 : *If a user gives a "stop" command, the robot stops within one second and does not move without any new command.*

P_3 is more general than P_2 because the robot may not stop immediately with a given "stop" command but within one second due to an urgent situation such as collision avoidance. Temporal property can be encoded in an observer by using the fact that the CC service is invoked every 100 milliseconds (see Sect. 4.2). We verified that the CC and UF services satisfied P_3 after modifying the observer to check this temporal property.

5.5 Experimental Results of the Verification

We used a WindowsXP machine with Pentium IV 2.8C and 1GB memory for the verification. Verification of each property of P_1, P_1', P_2, and P_3 generated around 100 states and took less than ten seconds and 128 MB memory, which was not burdensome to developers. Notice that what we had verified here was a *real implementation*, not an abstract model. We replaced the legacy C/C++ implementation of control software loaded on SHR100 with the Esterel program. After replacing the control software, SHR100 operated successfully with high reliability obtained by formal V&V.

As we have seen through Sect. 5.2 to Sect. 5.4, defining safety properties takes considerable effort. We believe, however, that such effort can reduce an overall development and its field operation costs by increasing the reliability of applications.

6 Conclusion

We have reported our experience of formally developing and verifying a home service robot SHR100. Our task to develop a robot with high reliability was a challenging target due to its own reactive nature and its complexity caused by coordinating diverse components together. The gist of our approach is to re-engineer a robot application to formally develop and verify *the core* for increased reliability. Through the re-engineering process, we found and fixed several subtle bugs which would be uncovered otherwise. In addition, we could demonstate that formal V&V was useful to identify feature interaction problems which were hard to detect through traditional testing. Furthermore, the new SHR100 implementation re-written in Esterel became compact and easy to analyze due to its clear component definitions and explicit communication mechanisms.

References

1. Esterel technology. http://www.esterel-technologies.com.
2. Honda asimo home page. http://asimo.honda.com/.

3. Sony qrio home page. http://www.sony.net/SonyInfo/QRIO/top_nf.html.
4. A.Bouali. Xeve: an esterel verification environment. Technical report, INRIA, 2000.
5. R. Alur and T. Henzinger. Time for logic. *ACM SIGACT News*, 22(3), 1991.
6. G. Berry. The foundations of esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.
7. J. Borrelly, E. Coste-Maniére, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The orccad architecture. *International Journal of Robotics Research*, 17(4):338–359, 1998.
8. E. Coste-Maniére and N. Turro. The maestro language and its environment : Specification, validation and control of robotic missions. *Proceedings of the 10th IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1997.
9. A. C. Domínguez-Brito, D. Hernández-Sosa, J. Isern-González, and J. Cabrera-Gámez. Integrating robotics software. *IEEE International Conference on Robotics and Automation*, 2004.
10. E.M.Clarke, O.Grumberg, and D.A.Peled. *Model Checking*. MIT Press, January 2000.
11. B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics: Why and how? *International Symposium on Robotics Research*, Oct 1995.
12. G.H. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
13. M. Kim, K. Kang, and H. Lee. Formal verification of robot movements - a case study on home service robot shr100. *International Conference on Robotics and Automation*, 2005.
14. M. Kim, J. Lee, K. Kang, Y. Hong, and S. Bang. Re-engineering software architecture of home service robots: A case study. *International Conference on Software Engineering*, 2005.
15. L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, pages 127–140, Liege, Belgium, 1995.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
17. R.T. Pack, D. Mitchell Wilkes, and K. Kawamura. A software architecture for integrated service robot development. *IEEE Internationall Conference on Systems, Man and Cybernetics*, 1997.
18. L.E. Pinzon, H.-M. Hanisch, M.A. Jafari, and T. Boucher. A comparative study of synthesis methods for discrete event controllers. *Formal Methods in System Design*, 15(2):123–267, 1999.