A Gray Box-based Approach to Automatic Requirements Specification for a Robot Patrol System

Soojin Park
Graduate School of Management of Technology
Sogang University, Seoul, Korea

***Abstract*—The black box-based requirements specification models representative by the use case model focus on specifying system behaviors exposed outside. While these models are sufficiently effective in specifying requirements for business applications behavior, they are limited in specifying requirements for embedded systems with relatively very short interaction sequences with users. To solve this problem, we have proposed a gray box-based requirements specification method to specify the inner logic of an embedded system, including a tool for automatic generation of requirements specification from some analysis models in our previous work. This study proves the benefits of the proposed software requirements specification method by applying it to a robot patrol system and showing the possibility of general use of the proposed method in the embedded system domain. Compared with our previous work, we enhance the tool for automatic generation of requirements specification, called *SpecGen*, and prove the benefit of the proposed method from multiple aspects. The application result on the robot patrol system case is quantitatively demonstrating that our proposed requirements specification method improves development productivity and enhances overall software product quality, including code quality.**

Keywords*—Embedded system; automatic requirement specifications generation; mobile robots; use case specification

I. INTRODUCTION

Embedded software refers to software embedded in various electronic products, from small appliances, including mobile phones, digital cameras, and MP3s to robotics systems [1]. Scenario-based specification techniques widely used in business applications are often used in the requirements specification for embedded software. The primary purpose of the scenario-based specification technique represented by the use case model [2] is to describe the interaction between the system and the environment in which the system is used. Business applications are realizing real-world business as services supported by software systems. Hence, most required behaviors of a business application can be captured from the statements for specifying interactions between the user and the system. Although the service provided by the embedded system results from each event generated by the user, it cannot be observable from the outside of the system which internal action the system performs until the service result is derived. In other words, requirements extractable from visible interactions between an embedded system and environmental factors in which it is used are relatively limited [3, 4].

Thus, when the requirements for an embedded system are specified using a use-case model, the amount of information identified in the requirements specification is insufficient as a specification for developers [5]. As is shown in Fig. 1, for an example of a generic flow of events for "Power On" use case of an embedded system is specified as: (1) A user pushes the power-on button to start the system; (2) The system is invoked and waits for the user's other request. Such use case specification is insufficient to be used as a requirements specification to guide the development team designs the embedded system. To overcome this lack of information when the use-case model is applied to the embedded system requirements specification, most existing studies [6-9] pre-populate various design diagrams such as state, sequence, class, or data flow diagrams .etc.

Even if we select a suitable design diagram to specify the inner mechanism of exposed system behavior, we should decide the depth of each design diagram. The deeper the depth of the diagram, the more sophisticated the system's behavior can be included in the requirements specification. We usually face a "what versus how" dilemma [10] in specifying requirements, which means a "how" in the preceding step means again the "what" in a subsequent step. Over-elaborated design diagrams in the requirement stage could violate the definition of a requirement in that it specifies solutions, not problems [11]. Furthermore, it can cause a raising initial system development cost. The requirements model, which includes the requirements set for embedded software developers, should provide the interaction requirements between the system's internal components. As the developers could refine the interaction between components by digging the depth, guidelines for the appropriate elaboration depth are needed to obtain the necessary information in the requirements stage.

Our previous work [12] proposed an extended requirements specification model from use case specification to satisfy these needs. The use case specification guides us to maintain a view of the target system as a single black box [13] when we specify requirements. In contrast, we named the proposed model a "gray-box" based requirements specification in the sense that it is a model based on a perspective that partially looks at the interaction between top-level components among interactions inside the embedded system. It suggests the trade-off between the elaboration depth and the effort to design interactions among internal components of an embedded system when utilized as a requirements specification.

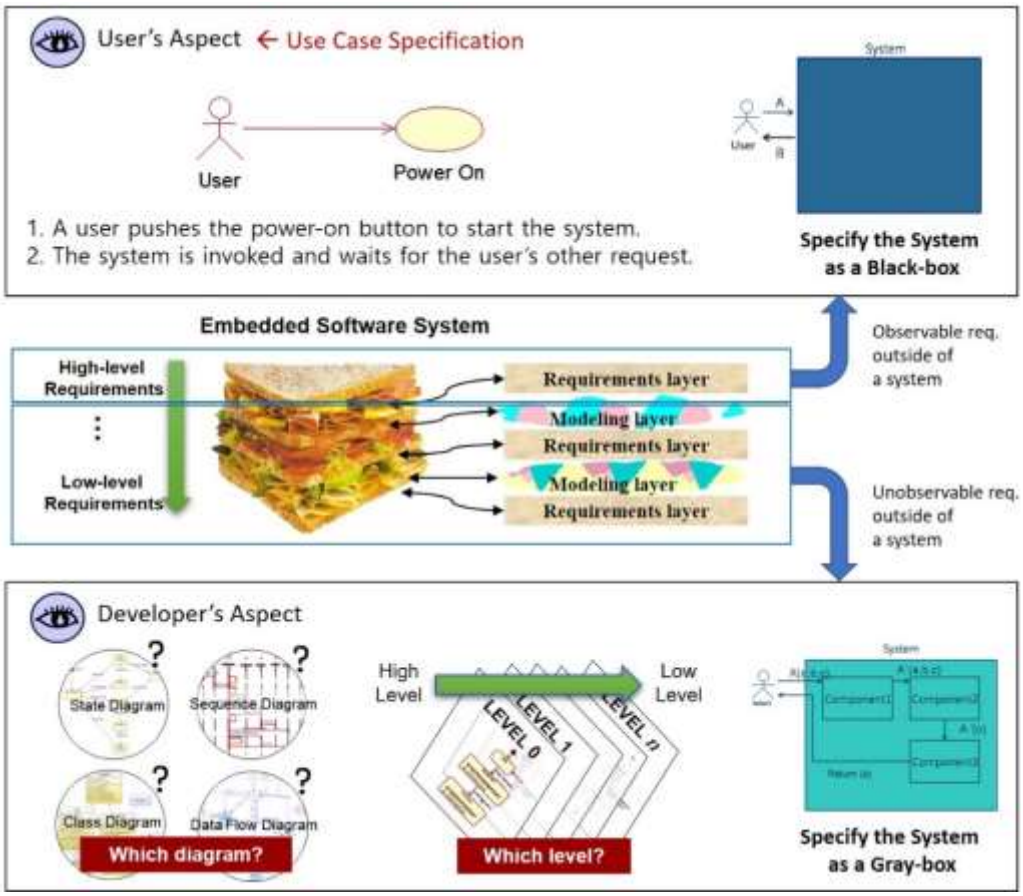


Fig. 1. Different Aspects of a Black Box-based Requirements Specification and a Gray Box-based Requirements Specifications.

The work reported here extends our previous work [12] in the following aspect:

- To show the extensibility of the proposed model through the application case of the more complex embedded system. While the scope of the case study was limited to a module of a mobile phone in [12], in this study, we extend the applicable domain area of the proposed model to the whole of a robot patrol system that is a different domain from the previous work. To prove the extensibility of the applicable area of the model is to prove that the proposed gray box-based requirements specification model can be a general method to specify requirements of embedded systems, not for only a specific system or domain.
- To prove the benefit of the proposed model using more various aspects. The author in [12] explains the benefit by showing that each elapsed time for the software development phases following the requirements phase is decreased when the proposed model is provided to the developers. Besides the enhancement of the productivity of developers, in this study, we show the software product quality and the code quality are also enhanced from using the proposed model through the more sophisticatedly designed experiment.
- To update the automatic generation of the gray box-based requirements specifications, which is renamed SpecGen. We re-developed the tool for utilizing a more prevalently used and better supported UML authoring tool when developers make an analysis model that is the source of the gray box-based requirements specifications. This work could be valuable in helping more developers use the proposed model and the supporting tool.

The rest of the paper is organized as follows: Section 2 investigates the trend of existing studies. Section 3 gives an overview of the gray box-based requirements specification method. Section 4 explains an automatic transition from a design diagram in UML to Microsoft Word typed tabular specifications implemented in *SpecGen*. Section 5 shows each step of requirements specification for a robot patrol system and some fragments of the state diagram and automatically generated specifications by *SpecGen*. Section 6 explains how we designed an experiment for showing the effectiveness using the proposed requirements specification model for the robot patrol system and discusses the results of the accomplished experiment. The conclusion of the paper is discussed in Section 7.

II. Related Work

A. Model-driven Development based Approaches

In embedded software development, model-driven development (MDD) based approaches are now widely used. MDD has the merit that developers can find the software's

essential features, thanks to information on the complicated system structure as an abstracted model [14]. The most typical MDD methods are the COMET method by H. Gamaa [15], which integrates object-oriented and concurrent processing concepts. The OCTOPUS method [16] models the system using a structural, functional, and dynamic model.

The research that addresses the requirements specification problems based on the model created by applying an MDD based approach can be found in [17-19]. Lattemann and Lehmann [17] define controller, actuator, and sensor as three main components that comprise the embedded system and suggest that the controller that controls the entire system should be intensively specified among the three roles. Lavi and Kudish [18] classify the model to be analyzed into the E-level representing the external structure and behavior of the system and the S-level representing the conceptual model of the system inside. They suggest an automatic documentation method for requirements specifications based on activity diagrams and state diagrams for specification and analysis of E-Level processes. Glinz [19] utilizes hierarchical activity diagrams after the relation between system state and objects that comprise the system with a source for the specification of requirements in an embedded system is identified.

Existing works only refer to the necessity that the entire system should be divided into lower systems. Each modeling phase should be recursively applied for the requirement specification of the embedded system. But there is no guideline for stopping the recursion for the elaboration depth of the model to be built. To solve this problem, we have started this study from the work that defines the elaboration depth of the analysis model for requirements specification, which was ignored in the previous studies while preserving their advantages.

B. Requirements Pattern-based Approaches

Another notable approach for requirements specification for embedded systems is requirements pattern-based one. Denger et al. [20] propose a natural language pattern to specify requirements in the embedded systems, including 1) meta models for the description of requirements and 2) meta-models for events and responses that we use to verify the completeness of the pattern language. The proposed patterns seem slightly less common compared to commercial phrase requirements. Matsuo et al. [21] use natural language controlled for requirements, limiting how they can combine simple sentences into more complex sentences. They proposed three different types of frames: noun frame, case frame, and feature frame, and they use the frames to parse requirement specifications, and organize them according to different perspectives, and verify requirement completeness. However, there exists a limit that the frame-based approaches seem to be more difficult for non-specialists to understand and apply. Konrad and Cheng [22] define formal specification pattern systems for embedded systems. These patterns are used to describe system properties mapped to linear time logic. Patterns are classified into qualitative (occurrence or order) and real-time (period, periodic, or real-time) patterns. There is a limit that we should specify the supporting model in a UML 12 variant. Postet al. [23] provide the successful application case of this system to automotive requirements. However, the application coverage is not complete.

A pattern is a set of solutions that are commonly applicable to recurring problems. Therefore, the pattern-based approach has an inherent limitation in the scope of its application. This study is different from the pattern-based approach in that it aims to develop a specification method generally applicable to the embedded systems.

III. Overview of Gray box-based Software Requirements Specification Model

The proposed gray box-based software requirements specification for embedded systems leverages partially cultivated analysis artifacts. However, designing all aspects of an embedded system is not proper in the requirements specification phase, considering that software requirements should focus on what services should be provided in the future. Thus, we have limited the design area to the following two diagrams to which the collaboration behavior between the inner components is to be extracted:

A. Top-level State Diagram of a Controller

A state diagram that shows state changes in the system corresponding to events occurring inside and outside the system is a typical diagram used to design dynamic views of the embedded system. Therefore, we selected it as the diagram to specify the internal behavior of the system corresponding to the event specified in the use case specification. After choosing to use state diagrams as the source of the requirements specification, another remaining issue was identifying which component could represent the state transition of a whole embedded system. A state of an entire system is a specific situation where specific values are assigned to all attributes of components comprising the system. Therefore, the question, which component should have the ownership of the state of a whole system is a controversial issue. Referencing Broy and Stauner [24], we have classified the roles of the main components in an embedded system into controller, actuator, or sensor. Among the three stereotypes of components, we defined the controller coordinating behaviors of other actuators and sensors as the component possessing the states of a whole system.

B. Sequence Diagrams Specifying Interactions of all Top Leveled Components - Controllers, Sensors, and Actuators

The proposed model specifies events exchanged among external subjects in a time sequence. As a flow of events in a use case is reflected in a sequence diagram, each internal interaction invoked by external stimuli from an actor is also designed using a sequence diagram to keep the same context. The owners of the events on the sequence diagram created in this step are actors, controllers, and sensors/actuators (that execute the controller's commands). The states in the top-leveled state diagram of the controller are added as annotations on the lifeline of the controller object in the sequence diagram, as depicted in Fig. 2. In the next step, the sequence diagram added state transitions of a whole embedded system is the source of the automatically generated requirements specification.

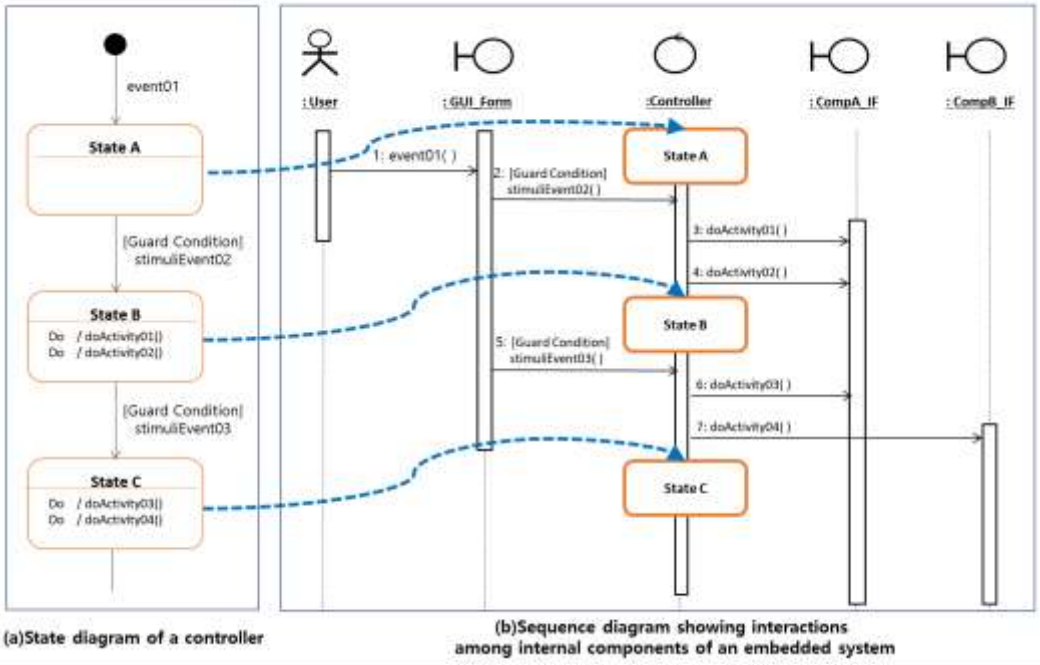


Fig. 2. The Relationship of Information Specified in (a) A State Diagram of a Controller and (b) A Sequence Diagram for showing Interactions.

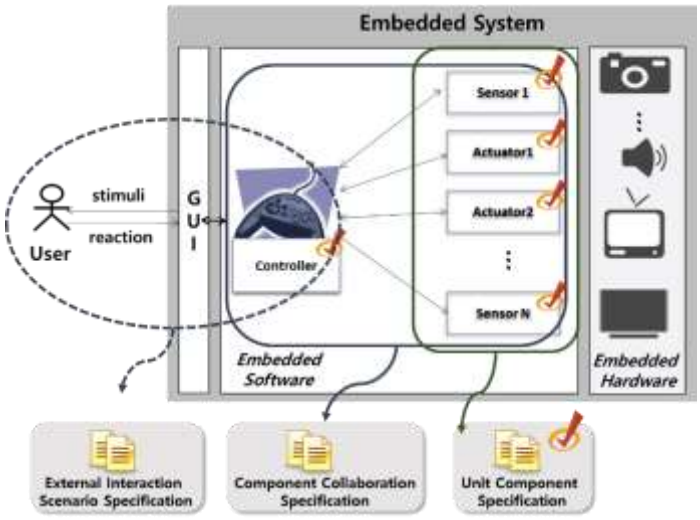


Fig. 3. The Coverage of Three different Requirements Specifications of the Proposed Gray Box-based Requirements Specification Model.

As shown in Fig. 3, once the two kinds of design diagrams are completed by developers, the following three different software requirements specifications can be automatically generated.

- External Interaction Scenario Specification: specifies the interaction between a system and an actor corresponding to the system's external environment. The information included in this specification is equivalent to the information contained in the use case diagram.
- Component Collaboration Specification: specifies the state changes of a controller due to inter-component interaction. The information included in this specification contains state-related information included in the state diagram for the component of the controller that controls the actuator and sensor of the embedded system. In addition, the information recorded in the sequence diagram, which is the result of designing the sequence of commands that the controller receives external stimuli and sends commands to other actuators and sensors, is extracted as this specification.
- Unit Component Specification: specifies the behaviors to be implemented by a specific component. This specification is written by classifying all operation calls in the previously extracted component collaboration specification by corresponding to the receiver and binding them. These unit component specifications are APIs for each class or component in the development stage, in other words.

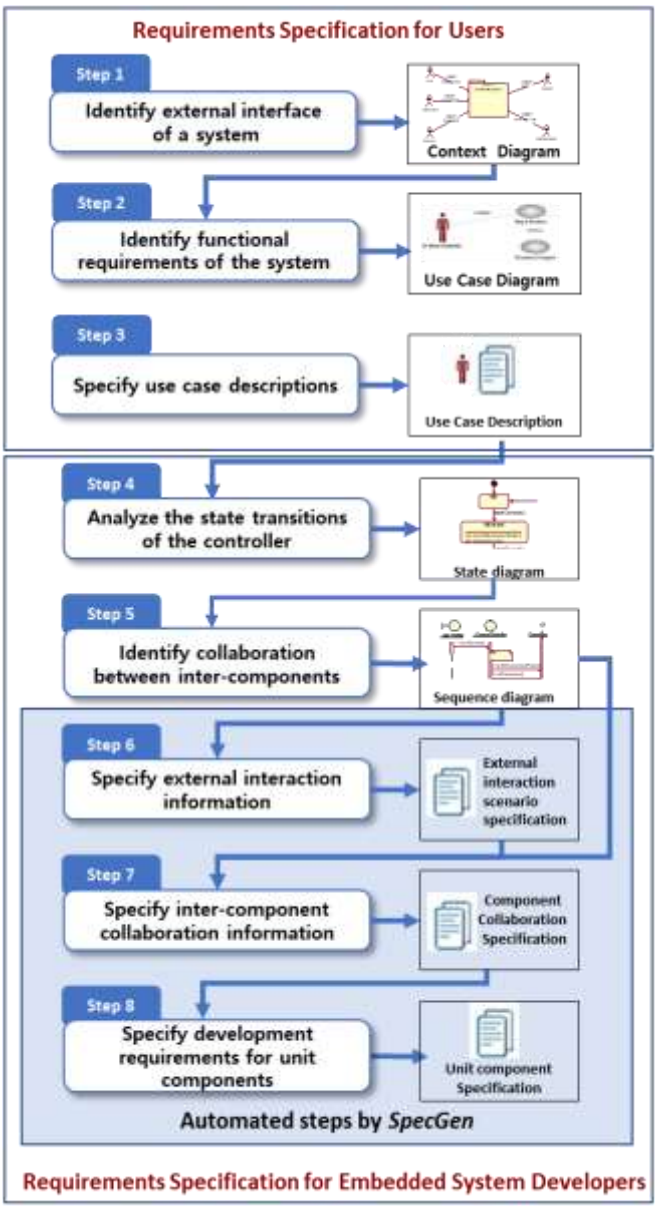


Fig. 4. The Process to Build the Proposed Gray Box-based Software Requirements Model.

Our proposed method does not exclude the steps of the existing black-box requirement specification but includes them. Fig. 4 illustrates each step to construct those specifications. A context diagram and a use case model are specified due to steps 1~3 in Fig. 4. A context diagram represented by UML (Unified Modeling Language) shows which users or interfacing systems are engaged to provide a service in the target system. Specifying the context diagram is not included in the general guidelines of the use case model. However, embedded systems are literally "embedded" in a hardware system. Thus, clear identification of external objects that an embedded system should interface with is critical. We also marked interface systems outside as actors in the context diagram for consistency with the use case model. A use case diagram that specifies the system's services is the same as a typical use case diagram. Steps 4~5 are for developing a design artifact, including a top-level state diagram of a controller and sequence diagrams for specifying collaboration between the controller and other sensors/actuators to respond to each stimulus outside of an embedded system. The following steps 6~8 are to automatically extract the three specifications explained above from the designed diagrams through steps 4~5. We also developed an automatic tool, *SpecGen*, to support these steps. The requirements specifications generated by *SpecGen* define the internal behaviors of an embedded system, which will be utilized as a guideline set for embedded system developers in the following development phases.

IV. SpecGen: A Tool for Automatic Generation of Requirements Specification from Design Diagrams

One of the originalities of our work is to provide a tool for top-leveled design artifacts to be transformed to requirements specifications automatically. This feature is important from three perspectives:

- In writing the requirements specification as a development guideline for developers, the information included in the design diagram created by the developer or designer is linked without loss.
- Since most developers refer to automatically generated requirements specifications and development proceeds, as a result, it does not matter if very few members with design ability use various UML diagrams in the development team.
- And, the support of automated tools can minimize the effort required to write requirements specifications in hand.

Our previous study [12] utilized ArgoUML [25] as the authoring tool for designing diagrams. In this study, we changed the authoring tool to StarUML 4.0 [26] as ArgoUML has not been versioned up. Fig. 5 shows a fragment of the transformation from a UML diagram in StarUML to a requirements specification as a Microsoft Word file by SpecGen. For using SpecGen, the first step is to extract diagrams authored using StarUML to an XMI file. To extract needed information from the XMI file, we should understand the structure of each object in the XMI file representing the UML model extracted from StarUML. Although we can catch the owner object of the first lifeline is "User" intuitively from the given sequence diagram, we can find it out after tracing several lines in the exported XMIs, as depicted in Fig. 5. It depicts the example with the shortest trace selected due to space constraints, but in some cases, the desired information is extracted through a traverse of more than ten lines of XML. Similarly, we analyzed all relevant XMI structures and compared the attributes in each requirements specification we defined. We implemented the transformation rules identified as such with SpecGen.

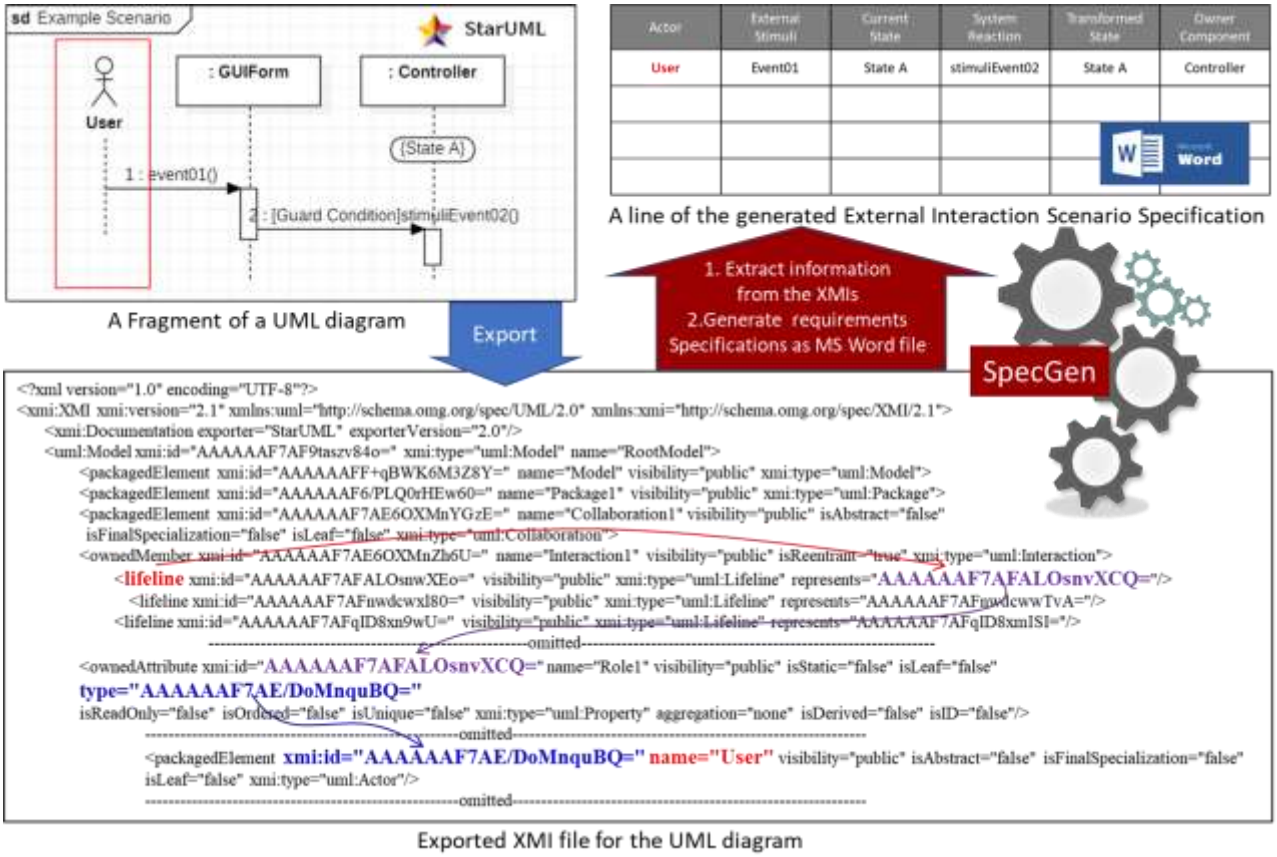


Fig. 5. A Fragment of the Transformation from a UML Diagram to a Requirements Specification by SpecGen.

V. Case Study: Automatic Requirements Specification for a Robot Patrol System

We selected a robot patrol system (RPS) as a target system to which the proposed requirement specification method is applied. RPS is a robot system that provides a service by sending out an alarm when an intruder is detected as it patrols the designated section. The reason to choose a robot system as a target system is that it is one of the typical system domains requiring the three components - controller, sensor, and actuator - in an embedded system as defined by Broy and Stauner [24]. Whereas an operation given for a robot patrol system is simple as "Keep patrolling here," many inner-sided interactions invisible to users are required to patrol within an area. These features are consistent with the feature of the target domain area to which we apply the proposed method.

The followings are the results and explanations of each step in Fig. 4 of the proposed model applied to RPS.

Step1: Identify the external interface of a system

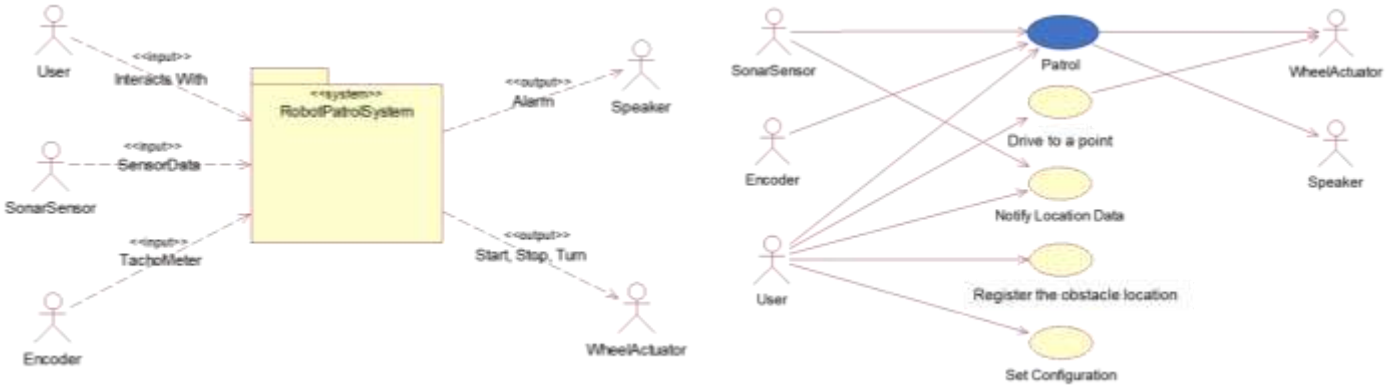
Fig. 6(a) is the context diagram (level 0 data flow diagram) to show the external interface of RPS. To keep the consistency with the following use case diagram, we specify all external entities as actors. The context diagram defines which entities are the sources of data and which entities are the data destinations. In RPS, whereas, *User*, *SonarSensor*, and *Encoder* are the data sources, *Speaker* and *WheelActuator* are the data destinations.

Step2: Identify functional requirements of the system

Fig. 6(b) is the use case diagram, which specifies functional services be provided by the target system. The use cases of RPS are: Patrol, Drive to a point, Notify location data, Register the obstacle location, Set configuration. And, the active actors that invoke a use case are the data source of the context diagram. So, the active actors of RPS are User, SonarSensor, and Encoder. The data sources of the context diagram come to be passive actors being the systems to be interfaced in the use case diagram. In RPS, Speaker and WheelActuator are the passive actors.

Step 3: Specify use case descriptions

Table I shows the use case description of the "Patrol" service. We select a tabular style, and most compartments of the use case specification are specified, including pre-conditions and post-conditions. There are one basic flow and two alternative flows in the "Patrol" use case.



(a) (b)

Fig. 6. Artifacts of Black Box-based Requirements Specification for Robot Patrol System: (a) Context Diagram and (b) Use Case Diagram.

TABLE I. ARTIFACTS OF BLACK BOX-BASED REQUIREMENTS SPECIFICATION FOR ROBOT PATROL SYSTEM: USE CASE SPECIFICATION FOR "PATROL"

Use Case Name	Patrol
Actor	User
Brief Description	The robot patrols the area based on the range of user input.
Basic Flow of Events	This use case begins when the user enters the destination range for patrol by GUI and gives the order to start. [1] The robot patrolling system(RPS) saves the start and destination positions and switches direction to destination positions. And the RPS gives the start command to Wheel Actuator. [2] The RPS reads the sensor values and identifies the intruder. If the intruder is detected, the flow goes to [A1]. [3] The RPS reads the current location and checks whether the RPS arrives at the destination. [3.1] If the robot arrives at the destination, give the stop command to Wheel Actuator, where the use case ends. [3.2] If the robot does not arrive at the destination, the flow goes to [2].
Alternative Flow 1	**[A1] Intruder detection** [1] If the intruder is detected, the RPS gives a stop command to the Wheel Actuator. [2] The RPS causes alarm bells through the speaker, at which point the use case ends.
Alternative Flow 2	**[A2] User's stop command** If the user gives the order to stop at any time, the RPS gives a stop command to the Wheel Actuator, at which point the use case ends.
Exception Paths	N/A
Extension Points	N/A
Pre-conditions	The RPS was initialized state. The robot's starting position is (0, 0). And the direction of the robot is assumed to be 90 degrees.
Post-conditions	The RPS is the stationary state according to user instructions.

With the artifacts depicted in Fig. 6 and 7, we can see what should be developed for the RPS. However, it does not provide sufficient information to guide what should be implemented because it defines only the interactions between actors and the system. If only these artifacts are given to developers, comparatively many decisions should be made by individual developer's capability to realize the specified requirements. If only these artifacts are provided to the developer, the individual developers must make a relatively large number of decisions, which could be a significant burden. The burden to the developers comes from the lack of details in requirements specification will be discussed in Section 6 with experimental results.

Step 4: Analyze the state transitions of the controller

In an embedded system, various sensors and actuators are equipped. However, only the controller has a meaningful state in an embedded system during the system's execution as other sensors or actuators are passive objects that receive commanders from the controller. For this reason, the state diagram of a controller should be created as a diagram explaining the behaviors of a whole embedded system. Fig. 7 shows the top-level state diagram of PatrolSystemController, which controls all other components in RPS. There are five meaningful states while the PatrolSystemController runs: Idle, Initialized, Patrolling, StoppedAtTheDestination, StoppedByIntrution.

Step 5: Identify collaboration between inter-components

After analyzing the state transitions of the controller, the next step is to identify collaboration between components. According to the use case specification in Table I, there are a basic flow and two alternative flows in the "Patrol" use case. Fig. 8 is the sequence diagram for the scenario combining the basic flow and alternative 2 (intruder detection). The one different point comparing with typical sequence diagrams is that the states are additionally annotated on the lifeline of the

controller. We can confirm that the four states- Idle, Initialized, Patrolling, StoppedByIntrution – which are related to the scenario, are annotated on the lifeline of the PatrollSystemController in Fig. 8. The collaboration in Fig. 9 shows that PatrollSystemController controls the sequence of messages to WheelActuatorIF, SonarSensorIF, and SpeakerIF, identified as actors as external modules to interface in the use case diagram. DirectionCalculator and IntrusionDetectionIF newly identified in designing the sequence diagram are also identified as the objects collaborating to provide the "Patrol" service.

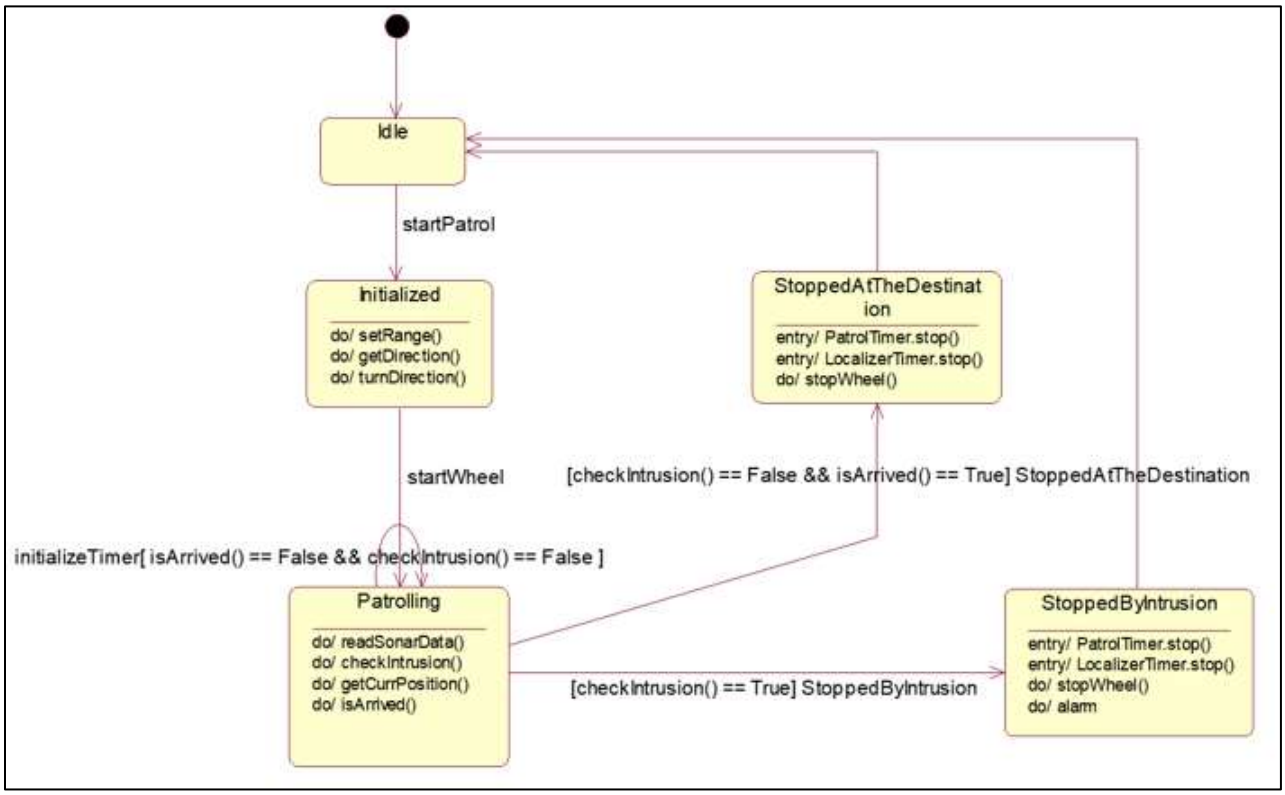


Fig. 7. The State Diagram for Patrol System Controller.

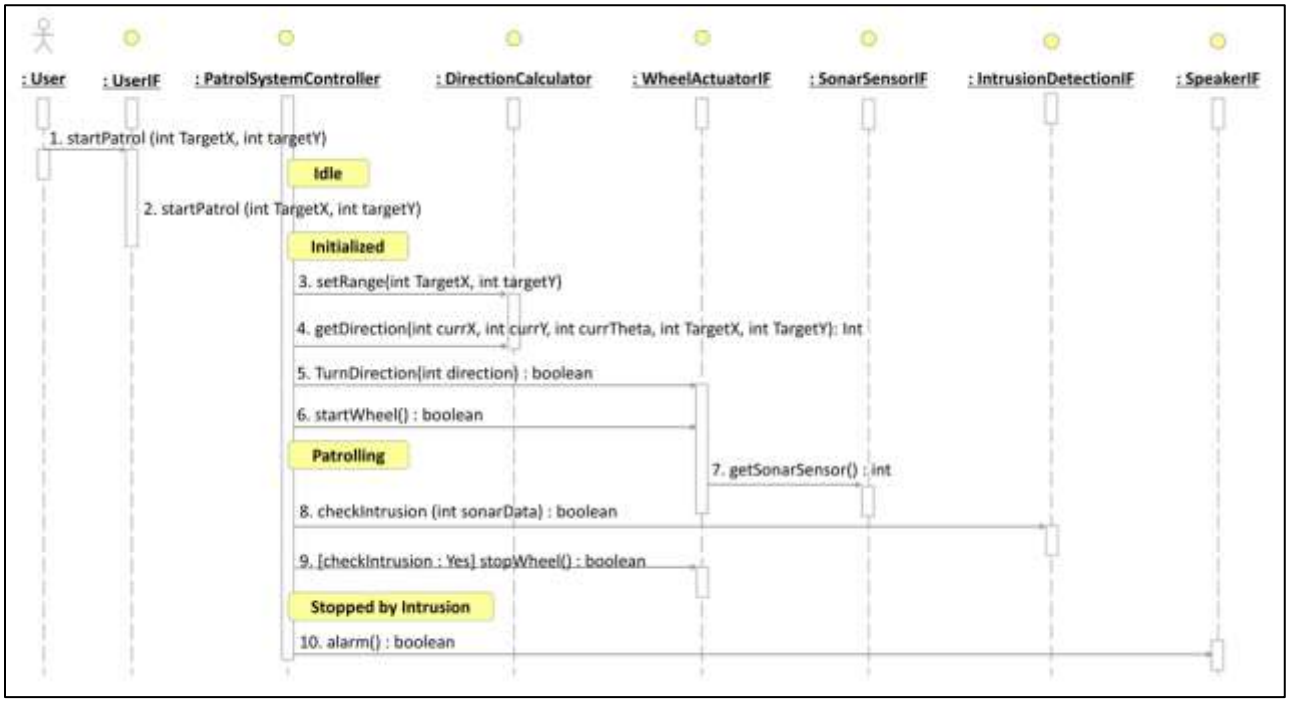


Fig. 8. The Sequence Diagram for the Scenario of the Composition of the Basic Flow and the Alternative Flow 2 in the "Patrol" use Case.

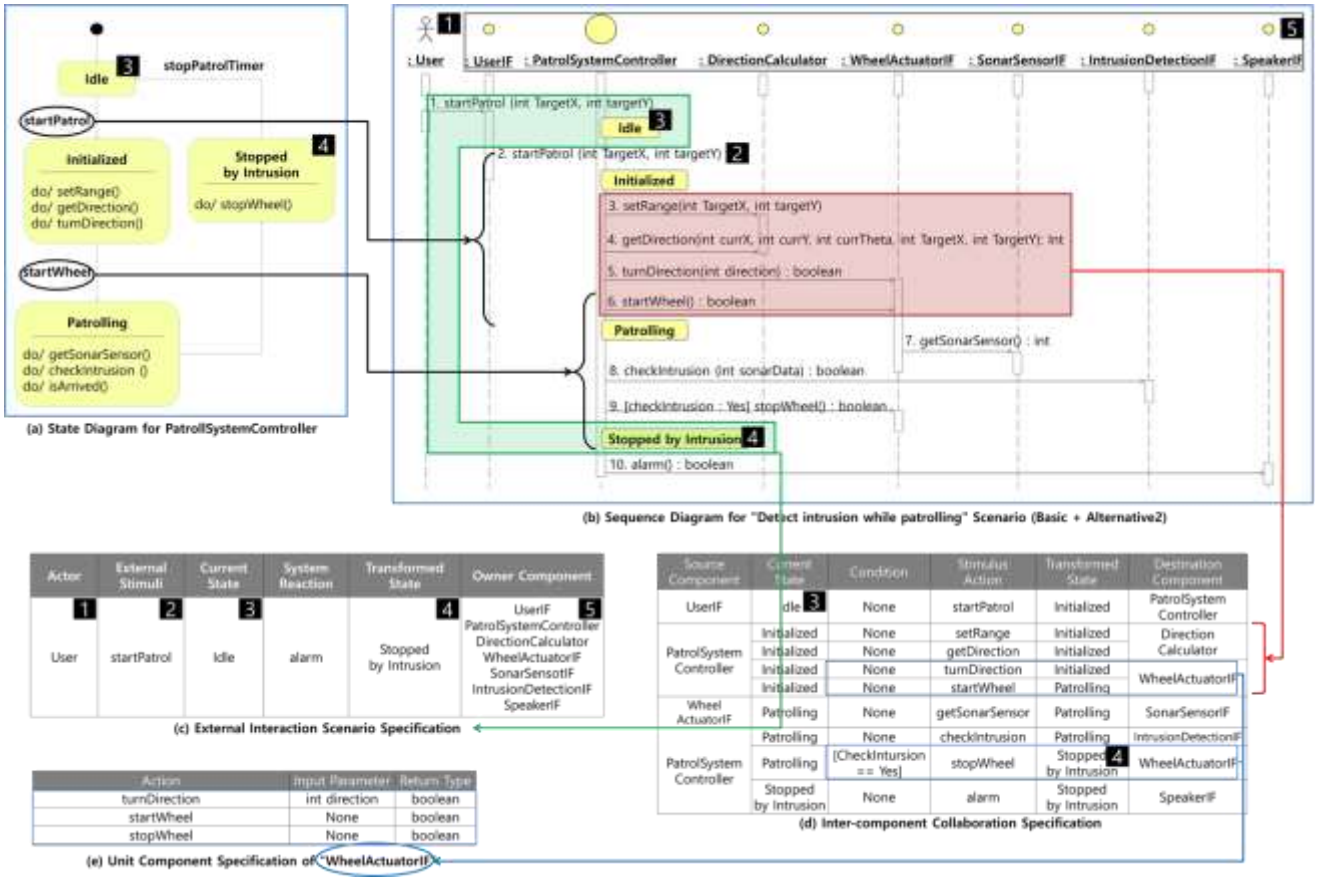


Fig. 9. Information Transformation from the Sequence Diagram to Requirements Specification for "Detect Intrusion While Patrolling" Scenario in the Robot Patrol System.

Unlike the general sequence diagram, one more thing to note is that the operation connected to each message in the sequencediagram must match the operation in the state diagram designed earlier. The operations that appeared in the state diagram and the state diagram are identical. As a result, the sum of the sequence diagrams created as many times as necessary contains all the information identified in the state diagram. Thus, the source of automatic generation of specifications through the following steps is the set of sequence diagrams as the result of step 5.

When step 5 is completed, sequence diagrams are created for each scenario combined based on the flow of events of the use case. As described above, the sequence diagram guided by the proposed model additionally specifies the controller's state transition information in the timeline. Using SpecGen, three additional requirement specifications are created through steps 6-8 based on the controller's state transition and the message sequence information that the controller controls to perform the scenario.

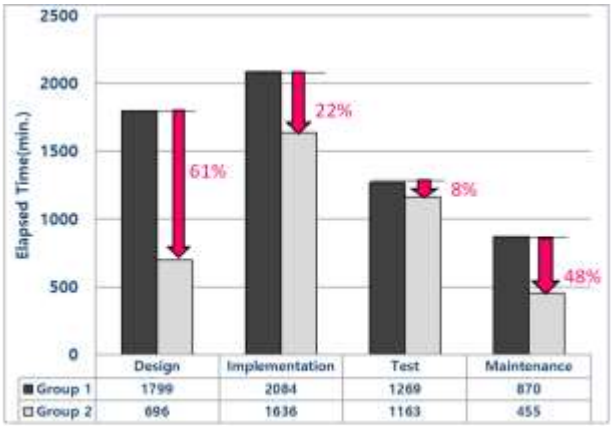
Step 6: Specify external interaction information

First, the contents of the external interaction scenario specification described in Fig. 9(c) are the same as the previous use case specification information. The external interaction scenario specification table specifies the stimuli and reactions between external actors and the whole system. Fig. 10 shows the relationship between the diagram and the automatically extracted and generated fragment of each specification. For understanding, the matching information is denoted by the same black-boxed number. The generated row in Fig. 9(c) specifies that User invokes startPatrol (External Stumilus) when the system is idle (Current State). Then, UserIF, PatrolSystemController, DirectionCalculator, WheelActuator-IF, SonarSensotIF, IntrusionDetectionIF, SpeakerIF (Owner Component) collaborate each other. The system's last reaction is to alarm (System Reaction), and the final state is stopped by intrusion.

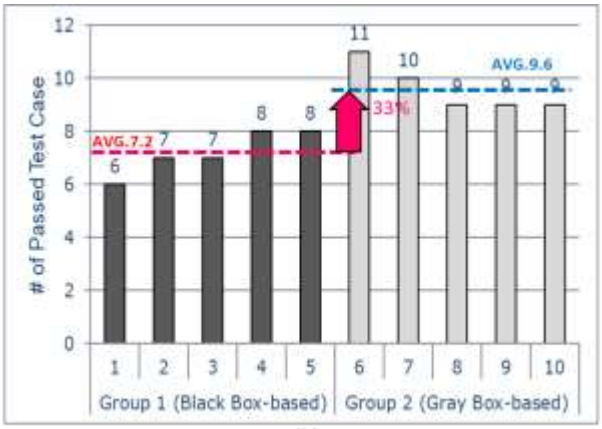
The specified content covers just a part of the "Patrol" use case. Fig. 9(c) specifies the external interaction scenario specification, including another scenario for the regular patrolling without any intrusion.

Step 7: Specify inter-component collaboration information

The second specification is generated from the message passing information in the sequence diagram. As annotated in the sequence diagram, the state transitions of the controller are also reflected in the inter-component collaboration specification. It is extracted one-to-one from each message on a sequence diagram. The first action to invoke each collaboration starts at the message from GUI (Graphical User Interface) object to the controller, not the message from an actor already specified in the external interaction scenario specification.

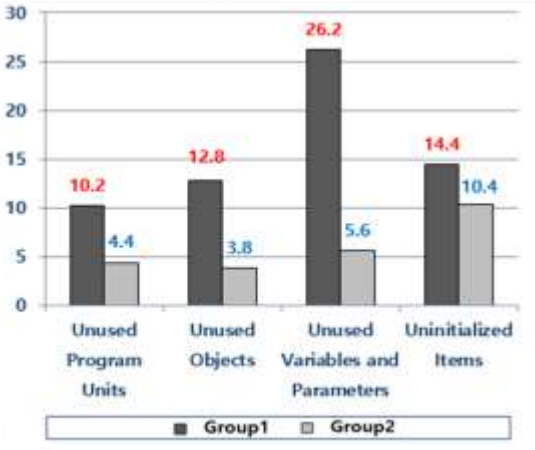


(a)



(b)

Group	Team	Unused Program units	Unused Objects	Unused Variables & Parameters	Uninitialized Items	Total number of Bad Symptoms
1 (Black -box)	1	0	3	22	2	27
	2	18	22	43	47	130
	3	10	10	22	13	55
	4	0	2	3	0	5
	5	23	27	41	10	101
Average		**10.2**	**12.8**	**26.2**	**14.4**	**63.6**
2 (Gray -box)	6	5	3	3	2	13
	7	3	2	6	10	21
	8	6	5	4	9	24
	9	4	7	12	19	42
	10	4	2	3	12	21
Average		**4.4**	**3.8**	**5.6**	**10.4**	**24.2**



(c)

Fig. 10. Comparison of the Experiment Results from Groups 1 and 2: (a) Elapsed Development Time, (b) The Number of Passed Test Cases, (c) The Number of Harmful Symptoms in the Code from Static Analysis.

Fig. 9(d) shows the inter-component collaboration specification for the "detect intrusion while patrolling" sequence diagram. A user invokes the startPatrol event. And then, the event makes UserIF trigger startPatrol message when the controller's state is Idle. Fig. 9(d) captures the message passing after the triggering according to the information in Fig. 9(b). The state transition of the PatrolSystemController in executing the "detect intrusion while patrolling" scenario is depicted in the Transformed State column in the specification. The state is transit from the Idle state to the states of Initialized → Patrolling → Stopped by Intrusion in the sequence.

Fig. 9(d) is the inter-component collaboration specification for the "Patrol" use case that includes the messages extracted from another sequence diagram for the regular patrolling scenario, denoted by shading. The content of Fig. 9(d) could be the key development specification for the developer in charge of the "Patrol" use case.

Step 8: Specify development requirements for unit components

The third specification automatically generated by SpecGen is the unit component specification. SpecGen classifies each message captured in the inter-component collaboration specification according to the destination component. For example, Fig. 9(e) is a table of the group of messages - turnDirection, startWheel, stopWheel- of which destination component is the same, WheelActuatorIF. The table becomes the unit component specification of WheelActuatorIF later if all of the other incoming messages to WheelActuatorIF appeared in other sequence diagrams are added. Fig. 9(e) is the completely generated unit component specification for the WheelActuatorIF of RPS. In the case of WheelActuatorIF, the extracted actions to implement the scenario in Fig.10 are equivalent to the whole set of actions in RPS. The three actions required to implement WheelActuatorIF are the same as the APIs for WheelActuator, which means that the developer in charge of the WheelActuator should implement each action in the unit component specification. And the other developers can reference the specification when they need to call any actions of WheelActuator.

VI. Evaluation of the Automatically Designed Requirements Specification for a Robot Patrol System

A. Experimental Design

To evaluate the effectiveness of the proposed requirements specification method, we designed an experiment. The scope of the experimental development is a basic flow and an alternative flow of the "*Patrol*" use case, which is described in

Table I. The participants of this experiment were 48 third or fourth-year university students from a computer engineering program, and they took the 8-week UML education course before they participated in the experiment. So, they can be classified as novice-level developers with more or fewer experiences in software development. 4 to 5 of them created a team.

Consequently, ten teams participated in this experiment. We divided the ten teams into two subgroups, one of which (group1) was given only the existing use case specification we named as the black box-based specification. The other group (group2) was given the artifact set of the provided requirements specification method as the gray box-based specification. The followings are the lists of the provided requirements specification artifacts for the two groups:

- For group1: black box-based requirements specification
    - Use case diagram
    - Use case specifications
- For group2: gray box-based requirements specification (black box-based requirements specification + additional requirements specifications generated by SpecGen)
    - Use case diagram
    - Use case specifications
    - External interaction scenario specification
    - Inter-component collaboration specification
    - Unit component specification

The results we wanted to confirm through this experiment and the corresponding measures we used were as follows:

- Enhancement of software development productivity: We compared the development time by asking each team to record each development phase's required time on the PSP sheet (Personal Software Process) [27]. The objective of the comparison is to confirm that the automatically generated requirements specifications from SpecGen contribute to decreasing embedded software development time.
- Enhancement of software product quality: We have defined 12 test cases for the given "Patrol" use case and tested the result from ten teams according to the test cases. And then, we compared the number of passed test cases related to functional aspects by each group, groups 1 and 2. We wanted to check if the provided requirements specifications from SpecGen can help the number of passed test cases increase.
- Enhancement of code quality: We expected that the requirements specifications generated from SpecGen contribute to enhancing the implemented code quality. If our expectation is correct, the number of bad symptoms from group 2 will be less than group 1. To confirm the assumption, we used the static analysis tool, Understand [28], to measure the number of bad symptoms inherent in the implementation codes from the two groups.

B. Experimental Results

- Enhancement of software development productivity: According to the PSP record documented during the two-week experimental development, the elapsed time of group1 in each development phase was shorter than group2. The teams' average total elapsed development time in group1 and group2 to develop the same use case, "Patrol," were 6,022 minutes and 3,950 minutes, respectively. The development time of group1 is the same as 66% of the elapsed time of group2.

As shown in Fig. 10(a), it is confirmed that group2 performed all steps, which are successive to the requirements specification, in less time than group1. In particular, the time taken for group2 to perform the design activity was less by 61% compared to group1. This decrease in development time can be interpreted as the benefit of the additionally provided requirements specifications generated by *SpecGen*, which include the analysis model in the early stage.

On the other hand, in the test phase, the execution time of group2 was less by only 8% compared to group1. In this experiment, the teams accomplished only integration tests since only a use case, "Patrol," was the development scope.

Since the use case specification is the exact requirements artifact provided for both groups, there is little difference in the information used for testing, so we can understand that there is no significant difference in the time taken for testing. To compare the time required for maintenance, we made the same request to change the requirements for each team belonging to both groups. The time to reflect the change request to the implementation code and test the changed code was measured as the maintenance time. As the teams in group2 can utilize the component collaboration specification generated from *SpecGen*, they took almost half the time of group1.

To ascertain that the elapsed time declines do not come from individual student's programming capabilities, we performed a simple regression analysis to analyze the correlation between the development time of each group and individual grades in programming-related courses. As a result, the R-Square value is 0.01, which explains no influence between students' development time and individual capabilities. Thus, the development time decline could be interpreted as the benefit of the proposed software requirements specifications.

- Enhancement of software product quality: We checked the number of passed test cases without any detected errors. The counted test cases are related to the functional requirements of the RPS, and the total test cases were 12. Fig. 10(b) shows that 7.2 test cases, averagely, are passed through the test in the product of group1. On the other hand, the average number of the passed test case in group2 was 9.6. This result means that compared with the development result of group1, the development result of group2 satisfies the given requirements by 33% more completely.

- Enhancement of code quality: We used Understand, a static analysis tool, to evaluate each group's quality of source codes. As a result, the number of detected bad symptoms of the source code implemented by group1 was 2.6 times larger than group2. The types of detected errors were unused program units, unused variables and parameters, unused objects, and uninitialized items, as shown in Fig. 10(c). These errors can be risks in software maintenance or reduce the efficiency of memory utilization in the future. Moreover, there is a wide variation in the number of detected errors extracted from five individual teams in group1, from 5 to 130. These figures produce evidence that there was no design guideline for developers (students), which can cause the quality of source code to depend wholly on individual developers' capability. On the other hand, we found a comparatively slight variation, from 13 to 42, in the numbers of detected total errors from five teams in group2. It shows that the proposed requirements specifications helped developers in group2 to construct uniformly qualified codes.

C. Comparison with Related Work

As proved by the experimental result, the proposed method help enhance the productivity of embedded software development and the quality of the product itself and implementation code. We analyze that the enhancement comes from providing (1) guidelines for the degree of detail for each analysis diagram, (2) support of an automating tool for the creation of specifications from the analysis diagrams, and (3) the specification methods for each development phase. Table II shows the results of comparing several related works and this study, based on the satisfaction of the features as mentioned above. Compared with that other related work [13, 18, 29, 30] limits providing guidelines for the degree of detail for each diagram and supporting an automatic tool for the proposed specification methods, Table II shows that this study acquires originality by providing the critical features mentioned above.

TABLE II. COMPARISON REQUIREMENT SPECIFICATION METHODS

	[13]	[18]	[29]	[30]	This Study
Guidelines for the degree of detail for each diagram	X	X	X	X	O
Automatic creation of specifications	X	X	X	X	O
Specification method for each deployment phase	O	O	O	X	O

VII. CONCLUSION

This study presents a gray box-based software requirements specification method for embedded system domain and guidelines for constructing an analysis model in the requirements phase, which can be a source of requirements extraction. The case study on a robot patrol system development demonstrates how the proposed guidelines are realized during the analysis model development and which information is documented as a requirements specification from the analysis model. An experiment to show the quantitative benefits of applying the proposed specification method and the revised supporting tool is conducted. The result of comparing this study and several related works based on the critical success features that brought about the enhancement demonstrated by the experimental results is also discussed.

Compared with our previous study, the originalities in this work could be captured in that:

- It proves the extensibility of the proposed gray box-based approach to automatic requirements specification by showing the result from applying it to the whole system of a robot patrol system different from the case study in the previous work. It shows that the proposed model is not a solution dedicated to a specific domain.
- It shows the evaluation results of the proposed approach with more various aspects. In addition to the decrease of the elapsed time for the software development phases after requirements, this study shows that the number of passed test cases of the target system can be increased by using the requirements specification automatically generated by the *SpecGen*, an automating tool for supporting the proposed model. Furthermore, the evaluation result shows that the source code's detected bad symptoms are decreased by a meaningful amount in the development group using the proposed approach compared with the other group not using it. All of the findings were measured quantitatively on an actual robot patrol system development, not a contrived system only for an experiment, which can be one of the originalities of our work.
- It provides more accessibility for embedded software developers by utilizing a more popular open-source UML authoring tool. In the previous work, the automating tool runs with ArgoUML. But, ArgoUML is not a widely used tool, and the upgrading is stopped. In this work, we re-build the automating tool, *SpecGen*, integrating with StarUML, one of the most popular UML authoring tools. Thus, more developers who already experienced StarUML can easily adopt *SpecGen* in their development.

REFERENCES

[1] E. A. Lee, "What's ahead for embedded software?," Computer, vol. 33, no. 9, pp. 18-26, 2000.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, The unified modeling language user guide, Pearson Education India, 2005.

[3] T. Pereira, F. Alencar, and J. Castro, "Requirements Engineering for Embedded Systems: The REPES Process," in *Proceedings of the 21st Workshop on Requirements Engineering*, 2018.

[4] S. Ernst, B. Tenbergen and K. Pohl, "Requirements engineering for embedded systems: An investigation of industry needs," in *Proceedings of the Int. Working Conf. on Requirements Engineering: Foundation for Software Quality*, pp. 151-165, 2011.

[5] S. Ferg, "What's wrong with Use Cases?" Available at: http://jacksonworkbench.co.uk/stevefergspages/papers/ferg--whats_wrong_with_use_ cases.html (accessed 25/08/2021, 2021).

[6] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Transactions on Software Engineering, vol. SE-8, no. 3, pp. 250-269, 1982.

[7] J. M. Thompson, M. P. E. Heimdahl, and S. P. Miller, "Specification-Based Prototyping for Embedded Systems," in *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering 1999*, pp. 163-179, 1999.

[8] J. Lavi, and J. Kudish, "Systems modeling & requirements specification using ECSAM: an analysis method for embedded & computer-based systems," Innovations in Systems and Software Engineering, vol.1, pp.100-115, 2005.

[9] M. R. Sena Marques, E. Siegert, and L. Brisolara, "Integrating UML, MARTE and SysML to improve requirements specification and traceability in the embedded domain," in *Proceedings of the 12th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 176-181, 2014.

[10] L. Dean, and W. Don, Managing software requirements: A use case approach, Addison-Wesley Professional, 2003.

[11] J. M. Nicholas, Project management for business and engineering: Principles and practice, Elsevier, pp.121. 2004.

[12] S. Park, "Software requirement specification based on a gray box for embedded systems: a case study of a mobile phone camera sensor controller," Computers, vol. 8, no. 20, pp. 1-11, 2019.

[13] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth and J. D. Reese, "Requirements specification for process-control systems," IEEE Transactions on Software Engineering, vol. 20, no. 9, pp. 684-707, 1994.

[14] B. P. Douglass, Real-time UML : developing efficient objects for embedded systems, Addison-Wesley Longman Ltd., 2000.

[15] H. Gomma, "Designing concurrent, distributed, and real-time applications with UML," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 1059-1060, 2006.

[16] J. Marin, T. Blanco, and J. J. Marin, "Octopus: A Design Methodology for Motion Capture Wearables," Sensors, vol. 17, no. 8, pp.1875, 2017.

[17] F. Lattemann, and E. Lehmann, "Methodological approach to the requirement specification of embedded systems," in *Proceedings of the International Conference on Formal Engineering Methods(ICFEM)*, pp. 183-191, 1997.

[18] J. Z. Lavi, and J. Kudish, "Systems modeling & requirements specification using ECSAM: a method for embedded computer-based systems analysis," in *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 2-11, 2004.

[19] M. Glinz, "Statecharts for requirements specification-as simple as possible, as rich as needed," in *Proceedings of the ICSE2002 workshop on scenarios and state machines: models, algorithms, and tools*, 2002.

[20] C. Denger, D. M. Berry, and E. Kamsties, "Higher Quality Requirements Specifications through Natural Language Patterns," in *Proceedings of the 2003 IEEE International Conference on Software - Science, Technology and Engineering*, pp. 80-90, 2003.

[21] Y. Matsuo, K. Ogasawara, and A. Ohnishi, "Automatic Transformation of Organization of Software Requirements Specifications," *in Proceedings of the 4th International Conference on Research Challenges in Information Science*, pp. 269-278, 2010.

[22] S. Konrad and B. H. C. Cheng, "Facilitating the Construction of Specification Pattern-based Properties," in *Proceedings of the 13th International Conference on Requirements Engineering*, pp. 329-338, 2005.

[23] A. Post, I. Menzel, and A. Podelski, "Applying Restricted English Grammar on Automotive Requirements - Does it Work? A Case Study," in *Proceedings of the Requirements Engineering: Foundation for Software Quality*, pp. 166-180, 2011.

[24] M. Broy, and T. Stauner, "Requirements engineering for embedded systems," Informationstechnik und Technische Informatik, vol. 41, pp. 7-11, 1999.

[25] ArgoUML. Available at: https://sourceforge.net/projects/argouml/ (accessed 25/08/2021, 2021).

[26] StarUML. Available at: https://staruml.io/ (accessed 25/08/2021, 2021).

[27] W. S. Humphrey, The Personal Software Process (sm), vol. 11, Carnegie Mellon University, Software Engineering Institute, 2000.

[28] Understand. Available at: https://www.scitools.com/ (accessed 25/08/2021, 2021).

[29] F. Lattemann, and E. Lehmann, "A methodological approach to the requirement specification of embedded systems, " in *Proceedings of the 1st IEEE International Conference on Formal Engineering*, pp.83-191, 1997.

[30] M. Glinz, "Statecharts for requirements specification-as simple as possible, as rich as needed," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)Workshop: Scenarios and state machines: models, algorithms, and tool*, pp.1-6, 2002.