# Cardion.spec: An Approach to Improve the Requirements Specification Written in the Natural Language Through the Formal Method

Masao Ito(✉)

Nil Software Corp., Tokyo, Japan
nil@nil.co.jp

**Abstract.** As the reliability and safety of a system become more important, we need the requirement specification that is clear to read and has no error. So far several approaches have been proposed; the formal modelling technique is one of the effective approaches to get the correct specification. But, in the embedded system field, we cannot expect that the most of system engineers know about the formal specification technique. They are the experts of the system, not the specialist of system or software engineering. In our paper, we provide the new approach to mitigate this problem. The user uses only the natural language, and a tool provides the check the text on the fly and points out the sentence that has a problem. The formal model is constructed in the background. The user doesn't need to know the description of the formal model. In this paper, we introduce this approach and the tool, Cardion.spec that we implement our idea.

**Keywords:** Requirements specification · Natural language · Formal specification · SPARK · Six-variable model

## 1 Introduction

In the safety-critical system, such as the airplane, automobile, railroad and so on, the reliability and safety is the important factor. The error that affects them in the early development phase increases the cost of re-work, and sometimes it is hard to do it again because the development time is usually very long. And, of course, if we cannot notice the error, it affects the reliability and safety of the system.

To solve this problem, many methods are being proposed up to now. Formal approach is the powerful approach to create the correct requirement specification because it has a mathematical background (e.g. axiomatic set theory, algebra, logic and so on). The people of formal method people say that the formal approach provides the better way to improve the quality of specification and eliminate the error in it. But usually the requirements specification is written in the natural language [1].

In this chapter, we briefly check the several issues of the natural language text and formal method. And at the end of this section we introduce the SPARK language [2], which we choose for the formal approach for our approach. The point is that the formal approach works in the background in it.

C. Kreiner et al. (Eds.): EuroSPI 2016, CCIS 633, pp. 58–69, 2016.
DOI: 10.1007/978-3-319-44817-6_5

### 1.1 Issues of the Requirement Specification Written in Natural Language

In the IEEE 830 guidelines [3], the eight guides on describing the requirements specification is provided:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

First, we focus on the unambiguousness. This guideline says, "Requirements are often written in natural language (e.g., English). Natural language is inherently ambiguous (4.3.2.1)". And, if we can write the requirements formally we can eliminate the ambiguity, but "one disadvantage in the use of such languages is the length of time required to learn them. Also, many non-technical users find them unintelligible." That is, requirements written formally are good in the viewpoint of ambiguity, but practically we cannot expect that every people learn and use it immediately in practice. So, we have to manage to write good requirements in natural language without the ambiguity. The other example is shown in [4]. In this paper, authors define the quality model of documents and check them to select the documents that need further review. The quality has nine attributes.

- Atomicity
- Unambiguity
- Conciseness
- Testability
- Traceability
- Consistency
- Formal Correctness
- Correctness
- Completeness

They check the review results of the "automotive specifications of the Mercedes-Benz passenger car development (PCD)" and find that "the majority of defects are assigned to content quality-attributes like completeness, correctness and consistency" [5]. From this result, we focus on the consistency with unambiguity. Other attributes are also important, but they have the difficulty to check. For instance, completeness means the lack of information for a specification, but only the author knows whether it is true or not.

### 1.2 Formal Approaches

By using formal methods, we can eliminate the ambiguity and keep consistency in a model because of the mathematical-based formality. For example, Z language [6] is based on the axiomatic formal theory, and use the set as a type of data (i.e. abstract data type). The function is a given by the relationship between sets. If there is a violation of the type, tool can indicate it.

So far, there are several applications built with the formal method and the books [7, 8] show the successful result. For example, in the airplane field, the mission computer of C-130 was developed using the SPARK language. In the SAET-METEOR project, B method was used.

But there is a weak point on applying the formal approach. As IEEE 830 says, it needs the long learning period to start using a formal method. There are many people to write and read the requirements specification, for example, the hardware/software designer, supplier, testers, manufacturing engineer. At least, they can read the documents written in formal approach correctly at the same time.

And we have another difficulty. There is no universal formal approach. If the problem or solution domains differ, the suitable formal method might differ. For example, Z language, that is one of the famous state-based formal ones, is suitable for the domain that expresses the requirements with the idea of the set. The event based language, like statechart or SCR (Software Cost Reduction) [9] is applicable for the state transition based system. So, we also might need to learn the several formal languages.

## 1.3 Spark

We choose the SPARK language [2] as the formal language in our approach. SPARK (latest version is SPARK 2014) is the subset of ADA language [10], and we can use it for specifying and/or designing and implementing an application, especially high-integrity one. As we shown before, it is already used and has got good achievements.

To prevent the misuse of language, there are several restrictions in comparison with the ADA language[1].

- The use of access types and allocators is not permitted.
- All expressions (including function calls) are free of side effects.
- Aliasing of names is not permitted.
- The goto statement is not permitted.
- The use of controlled types is not permitted.
- Raising and handling of exceptions is not currently permitted.

The SPARK language has already several tools, such as GNATprove. It provides the many checking mechanism. Most notable checking is the flow analysis. The short example is showing below.

The keyword “Depends” shows the dependency between the variables. For example, “X => Y” means “X depends Y”, that is, the value of X is determined by the value of Y. This description is useful to analyse of data flow (Fig. 1).

[1] http://www.univ-orleans.fr/sciences/info/ressources/webada/doc/spark/ug/language_subset.html#introduction-to-spark.

```
procedure Swap (X, Y : in out Integer)
   with Depends => (X => Y, Y => X)
is
   T : Integer;
begin
   T := Y; -- should be T := X;
   X := Y;
   Y := T;
end Swap;
```

```
q.adb:4:20: warning: unused initial value of "X" [unused_initial_value]
q.adb:5:12: warning: missing dependency "null => X" [depends_null]
q.adb:5:32: warning: missing dependency "Y => Y" [depends_missing]
q.adb:5:32: warning: incorrect dependency "Y => X" [depends_wrong]
...
```

**Fig. 1.** Flow analysis

## 2 Method

In our approach, we analyze the requirements specification in natural language, and increase the quality of documentation incrementally. We use the SPARK language in this process, but user doesn't have to know this language. Only the tool uses it for background checking.

First of all, we show the basic flow of our approach. As we focus on the requirements specification of the embedded system, especially the system on the automobile. The model we use in our approach is mainly the finite state machine (FSM) model.

### 2.1 Outline of Our Process

There are three steps to improve the quality of the documents. First of all, we try to find the simple error in the sentence. Next, we extract the static and dynamic model. The dynamic model is important in the embedded system. The last phase, we generate the SPARK codes and make an effort to find the problematic points.

Tool doesn't automatically correct the error. It just points it out. And it might provide the candidate sentence, if possible. It is up to the user whether he/she revises it or not.

*(Step1).* The input is the requirements specification written in natural language (currently, we test only Japanese documents). We do the lexical analysis to get the chunks of a sentence with the word class, and do the syntactic parsing to get the modification structure. In Japanese the lexical analysis is very important, because the Japanese sentence has no space between words in a sentence. Next, we check the text and find the problematic points by using the basic information of text, such as;

(a) Long phrase/sentence
(b) Consecutive kanji/hiragana/katakana characters
(c) Lack of the object of the verb

*(Step2).* After the first step, we extract the static and dynamic elements to create the FSM model. As for static elements, we distinguish the other system and find the input data for the target system and output data to the other system(s). The dynamic elements is directly relating to the FSM. We extract the state and transition between states with the event/guard condition/action. The input and output of the static model is relating to the action of state transition.

Then we create a dictionary between natural language (Japanese) phrases to an English word. For example, “Clear the setting speed” is assigned into the SET_SPEED_CLEAR or “SET_SPEED = 0”

*(Step3).* Finally we convert the FSM into the SPARK codes, and compile them. If we encounter the compile error, our tool interprets it and indicates the user the problematic part.

## 2.2 Static Model

The aim of the static model is finding the system boundary of the target system and other system, and designating the data across the system boundary.

If there is no supporting information (e.g. system structure), it is hard to identify the outer parts of the target system. For example, when we read the sentence “output the target acceleration into the engine control”, we may easily understand that the engine control is outer part of the cruise control system. Because the system engineer usually knows the structure of the system. But for the tool it is difficult to know it automatically if it doesn’t have the knowledge of the system structure. So, this is the first point that the tool asks for the specification writer.

If the system boundary becomes clear, the flow of data across it is easy to capture.

## 2.3 Dynamic Model

For the embedded system, the dynamic behaviour is important because it is usually a reactive system. And the interaction between human and the machine is also essential in the recent complicated system.

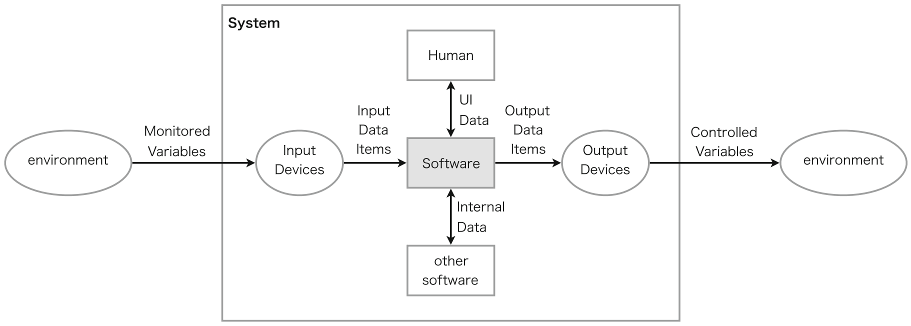


**Fig. 2.** Six-variable model

Before entering into the detail of the dynamic model, we'll explain the six-variable model of the system structure in this paper. In general, four-variable model is used; monitored and controlled variables and input and output data items [9]. But we use the six-variable model; we add the interaction of human-system (software) and the internal data to other part/system (Fig. 2). The four-variable model [11] is obviously the subset of six-variable model in a simple system that has no human interaction.

### 2.3.1 State

The Σ is the set of state. And we define the helper function for the state.

| | |
|---|---|
| id(σ) | :id |
| name(σ) | :name |
| parent(σ) | :super-state |
| children(σ) | :sub-state |
| type(σ) | :"basic" or "super" |
| default(σ) | :is default state? |

This is the simple definition of the state. We do not use the entry action/exit action/ activity.

### 2.3.2 Transition

The T is the set of transition. And we define the helper function for the transition.

| | |
|---|---|
| src(τ) | :source state of transition |
| trig(τ) | :trigger event |
| cond(τ) | :condition |
| act(τ) | :action |
| dest(τ) | :destination state of transition |
| prty(τ) | :priority |

In the high-level requirements specification, the priority is seldom used.

### 2.3.3 Mapping

Basically the elements of the transition are expressed like <*src*, *trig*, *cond*, *act*, *dest*>. It means that "if we are in the *src* state and *trig* event comes when the condition *cond* satisfies, the system does *act* asynchronously and go to the *dest* state." We extract each element of transition from the analyzed text.

Previously, we see the six-variable models. The software is relating to the four direct variables of the six-variable model. They are;

(1) Observed variable
(2) Controlled variable
(3) User-input variable
(4) External-system variable

It is possible to think that the user-input variable is a kind of the input data, but we have to remember that the input data come from the environment. So, it is appropriate to think it is the different variable.

Next we show the patterns that define the way to set the transition according to the difference of the four variables.

*(1) Pattern 1: Observed variable (comes from Environment).* We consider the following example: "In the constant speed mode, if the system find the 5 km/h difference between the speed of the self-car and the setting speed, the system clear the setting speed and go to the standby state". In this example, the speed of the self-car is the observed variable. The setting speed is an internal variable.

We assume that the observed variable are monitored periodically, so, we introduce the special keyword PERIODICAL as the trigger of the transition.

src(τ) = s11, name(s11) = "Constant_Speed"
trig(τ) = PERIODICAL
cond(τ) = "5 km/h difference between the speed of the self-car and setting speed"
act(τ) = "Clear_Setting_Speed"
dest(τ) = s12, name(s12) = "StandBy"

*(2) Pattern 2: Controlled variables.* We think about the next sentence; "In the constant speed mode, the system calculates the target acceleration and output it to the engine control."

This sentence shows that system execution an action of calculation and output to engine control periodically.

src(τ) = s21, name(s21) = "Constant_Speed"
trig(τ) = PERIODICAL
cond(τ) = nil ("nil" means no object)
act(τ) = "calculates the target acceleration and output to engine control"
dest(τ) = s22, name(s22) = "Constant_Speed"

*(3) Pattern 3: User-input variable.* The user input causes the trigger of a transition; "In the standby mode, when user push down the AR(Accel/Resume) switch, the system moves to the constant mode."

src(τ) = s31, name(s31) = "StandBy"
trig(τ) = "AR_Switch"
cond(τ) = nil
act(τ) = nil
dest(τ) = s32, name(s32) = "Constant Speed"

*(4) Pattern 4: External-system trigger.* In this final pattern, the outside of the system gives a trigger; “In the cruise control operable mode, if there is an error notification from other system, the system cancel the constant speed mode, clear the setting speed and transit to error state.”

src($\tau$) = s41, name(s41) = “cruise control operable mode”
trig($\tau$) = “Malfunction Notification”
cond($\tau$) = nil
act($\tau$) = “cancel the constant speed mode, clear the setting speed”
dest($\tau$) = s42, name(s42) = “Error”

## 2.4 Ask the Author

The state and the transition might not be decided certainly from an analysis of the natural language sentences automatically. So, the tool question the author of the requirements specification in order to fill the missing elements or select one from the multiple candidates of possibilities.

First, the integrity of the state transition is checked;

- Does all the states have the IN or/and OUT transition (except for the default state and the start/exit state)?
- Is there a unique default state?
- Is there an unconditional transition? When many transitions between the same states, there is a transition that hasn’t the event and condition.

The more complicated example is relating to the nesting (hierarchical) state transition. For example, consider the next sentence: “go to main_switch_on state and transit to standby state”. There are two possibilities. One is just transit to main_switch_on state and then standby state unconditionally. Another is transiting to main_switch_on, which is the super-state of standby state, and then transit to standby state from the default state of main_swtich_on state. In this case, the tool asks the author which is right.

## 2.5 Tool

We build the tool, Cardion.spec, to validate our idea. It consists of several parts. Checker does the basic verification of Japanese text from the viewpoint of readability (Sect. 2.1 Step1). The analyser creates the FSM internally from the results of the checker. The generator creates the various codes. One is the PlantUML[2] codes to visualize the FSM diagram, the second is the SPARK codes; both specification (ads) and body part (adb) (Fig. 3).

### 2.5.1 Dictionary

Dictionary is the user dictionary and has two roles. First, currently our tool analyses only Japanese requirements specification, so it converts Japanese text to English to generate the SPARK codes. Another purpose is creating the variable from the Japanese phrase to which FSM is relating. For example, the phrase “ON state of the engine switch”

[2] http://plantuml.com/.

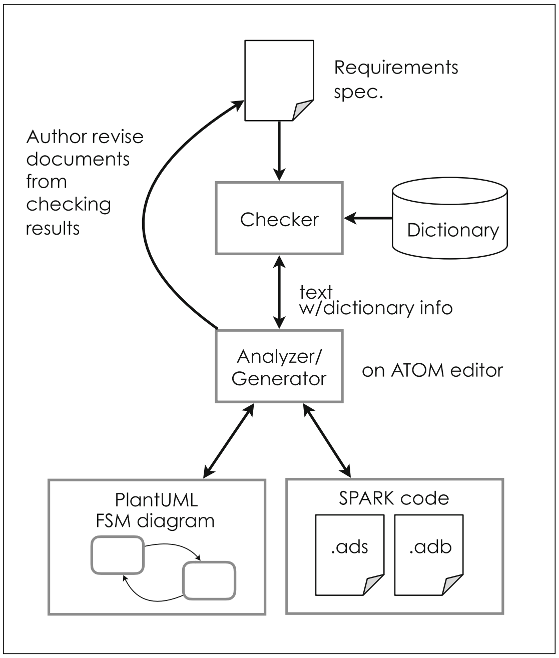


**Fig. 3.** Structure of Cardion.spec tool

becomes the variable name "S_IG_ON". In this process in which the user chooses the appropriate name, it is also good timing for the user to rethink the use of words to avoid the subtle fluctuation of words.

Here is the example of user dictionary (actually, the left part of list is the Japanese word).

@start cardion_jdic ("@start" is the keyword to indicate the start of the user dictionary)
Initial state, S_INIT
ON state of engine switch, S_IG_ON
OFF state of engine switch, S_IG_OFF
Standby state, S_NORMAL_RUN
Const speed state, S_CRUISE
Error state, S_CC_ERR
Engine switch ON, E_Ig_On

As this example shows that the state has a prefix "S_", and the event has a prefix "E_". It helps the analyzer to inspect the requirements specification.

### 2.5.2 Generation of SPARK Codes

To generate SPARK codes from FSM, we use some rules. In specification part, we define the type of the event, the state and the function as the action of transition (Fig. 4).

The transition is defined by the procedure that accepts the event and sets the new state (Fig. 5).

```
type Event_Type is (E_Ig_On, E_Ig_Off,
                       E_Cc_Sw_On, E_Cc_Sw_Off,
                       E_Ar_On, E_Cs_On,
                       E_Brake_On, E_Cancel_On,
                       E_Detect_Error);
```

**Fig. 4.** Example of generated definition of event type (specification part)

```
procedure State_Transition (E  : Event_Type) is
begin
   case IState is
      when S_INIT =>
         IState := S_IG_OFF;
      when S_IG_OFF =>
         case E is
            when E_Ig_ON =>
               IState := S_IG_ON;
            when others =>
               null;
         end case;
      when S_Ig_On =>
         IState := S_CC_NO_ERR;
      when S_CC_NO_ERR =>
         IState := S_CC_SW_OFF;
      when S_CC_SW_OFF =>
…
```

**Fig. 5.** Example of generated state transition (body part)

We expect that the generated codes are valid as the SPARK codes, and the tool, Cardion.spec, automatically compile them. And if there are some errors, we use this error information to refine the requirement specification. For example, if the event is defined in the specification part and all events don't appear in the body part, compile create error and the tool warn the user that the event is defined correctly.

There are two possibilities of problem that we have to consider as for warning to the author. One is that the document is not appropriate and the tool issued right warning. The second is the tool cannot analyze the document correctly. The latter is the false-positive indication. In our experience, it isn't possible to make this false-positive ratio into zero. But we noticed, through our experiment for one year, that it is acceptable to the author when the ratio is below the quarter of the correct indications.

## 3 Conclusion

In this paper, we explain an approach to improve the requirement specification written in natural language. In this approach, we have three steps to improve the document. First, we analyse the document by the lexical analysis and syntactic parsing to find out the simple problematic parts. As the second, we create the finite state machine and find the lack of elements (e.g. no out transition from the normal state). Finally, when compiling the generated SPARK codes from the FSM, compile errors are the candidate of error of documentation.

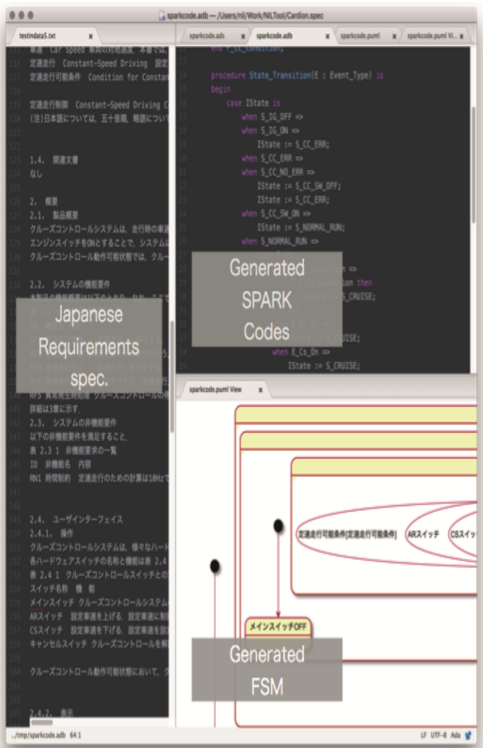
Simple, we can write the requirements specification in a formal specification language from the beginning. So, it becomes possible to process the requirements specification automatically by machine, and it is advantageous. But it is difficult for every people (i.e. writer/reader) to become familiar with the formal approach. If the reader doesn't understand the specification language correctly, he/she might misunderstand the correct document.

Our approach uses the formal approach in the background, so the user can concentrate on the natural language document and our tool assists this activity.

## Appendix

A user writes the specification in the natural language (Fig. 6, left). Cardion.spec creates the SPARK codes (Fig. 6, upper right) and FSM model (Fig. 6, lower down) internally. The user does not need to see those representations, could focus on the natural language text.

But we could use the SPARK codes for the validation purpose.



```
定速走行　Constant-Speed Driving　設定
定速走行可能条件　Condition for Constan

定速走行制御　Constant-Speed Driving C
(注)日本語については，五十音順，略語につい


1.4.　関連文書
なし

2.　概要
2.1.　製品概要
クルーズコントロールシステムは，走行時の車速
エンジンスイッチをONとすることで，システム
クルーズコントロール動作可能状態では，クル

2.2.　システムの機能要件

RF5 異常発生時処理　クルーズコントロールの
詳細は3章に示す。
2.3.　システムの非機能要件
以下の非機能要件を満足すること。
表 2.3 1　非機能要求の一覧
ID　非機能名　内容
RN1 時間制約　定速走行のための計算は10Hzで


2.4.　ユーザインターフェイス
2.4.1.　操作
クルーズコントロールシステムは，様々なハード
各ハードウェアスイッチの名称と機能は表 2.4
表 2.4 1　クルーズコントロールスイッチとの
スイッチ名称　機　能
メインスイッチ　クルーズコントロールシステム
ARスイッチ　設定車速を上げる。設定車速に制
CSスイッチ　設定車速を下げる。設定車速を設
キャンセルスイッチ　クルーズコントロールを解

クルーズコントロール動作可能状態において，ク


2.4.2.　表示
```

```
    procedure State_Transition(E : Event_Type) is
    begin
        case IState is
            when S_IG_OFF =>
            when S_IG_ON =>
                IState := S_CC_ERR;
            when S_CC_ERR =>
            when S_CC_NO_ERR =>
                IState := S_CC_SW_OFF;
                IState := S_CC_ERR;
            when S_CC_SW_ON =>
                IState := S_NORMAL_RUN;
            when S_NORMAL_RUN =>

                                 =>
                        if F_Cc_Condition then
                            IState := S_CRUISE;
                        end if;
                    when E_Ar_On =>
                        IState := S_CRUISE;
                    when E_Cs_On =>
                        IState := S_CRUISE;
```

Fig. 6. Cardion.spec tool image

## References

1. Sikora, E., Tenbergen, B., Pohl, K.: Industry needs and research directions in requirements engineering for embedded systems. Requirements Eng. **17**(1), 57–78 (2012)
2. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
3. IEEE: IEEE Recommended Practice for Software Requirements Specification (IEEE Std 830-1998). IEEE (1998)
4. Ott, D.: Automatic requirement categorization of large natural language specifications at Mercedes-Benz for review improvements. In: Doerr, J., Opdahl, A.L. (eds.) REFSQ 2013. LNCS, vol. 7830, pp. 50–64. Springer, Heidelberg (2013)
5. Ott, D., Raschke, A.: Review improvement by requirements classification at Mercedes-Benz: limits of empirical studies in educational environments. In: 2012 IEEE Second International Workshop on Empirical Requirements Engineering (EmpiRE). IEEE (2012)
6. ISO/IEC, ISO/IEC 13568:2002: Information technology – Z formal specification notation – Syntax, type system and semantics, ISO (2002)
7. Boulanger, J.-L.: Industrial Use of Formal Methods: Formal Verification. Wiley, New York (2013)
8. Iordache, O.: Methods. In: Iordache, O. (ed.) Polystochastic Models for Complexity. UCS, vol. 4, pp. 17–61. Springer, Heidelberg (2010)
9. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. (TOSEM) **5**(3), 231–261 (1996)
10. Barnes, J.: Programming in Ada 2012. Cambridge University Press, Cambridge (2014)
11. Parnas, D.L., Madey, J.: Functional documentation for computer systems engineering: version 2. McMaster University, Faculty of Engineering, Communications Research Laboratory (1991)