The Journal of Systems & Software 178 (2021) 110975

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A three-step hybrid specification approach to error prevention☆

Shaoying Liu

Graduate School of Advanced Science and Engineering, Hiroshima University, Japan

ARTICLE INFO

Article history:
Received 27 August 2020
Received in revised form 21 February 2021
Accepted 6 April 2021
Available online 16 April 2021

Keywords:
Requirements analysis
Hybrid specification
Formal methods
Software productivity
Software reliability

ABSTRACT

Effectively preventing errors in requirements analysis and design is extremely important for enhancing software productivity and reliability, but how to fulfill this goal remains an open problem. In this paper, we propose a concept of *hybrid specification* and describe a novel *three-step hybrid specification approach* to address this problem. We discuss how the three-step approach can be used to effectively prevent errors in the early phases of development. The expected effect of the approach is to strike a good balance between enhancing productivity and ensuring the reliability of the program implemented. We present a controlled experiment to evaluate the effectiveness of the approach. The result of the experiment shows that our method can detect and prevent 28.36% more errors than a comparable traditional requirements analysis method.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Producing error-free software has always been developers' goal in software projects but how to achieve it still remains an open problem (Jha et al., 2014; Abu-Faraj, 2019). Although no perfect solution is found yet, it has been well recognized that both error prevention and error detection are necessary in order to effectively and efficiently remove errors from software systems. As far as error detection is concerned, researchers have developed many kinds of techniques to deal with it, such as the main stream techniques *testing* (Liu and Nakajima, 2020) and *review* (Parnas and Weiss, 1985; Liu et al., 2012) for systems in general, and the formal techniques *model checking* (Atlee and Gannon, 1993; Clarke et al., 2000) and *formal verification* (Eakman et al., 2015; Liu, 2016; Wang and Liu, 2018) for critical systems. But for error prevention, the challenge is even greater. Error prevention is a general concept and different researchers may interpret it differently. In this paper, by error prevention we mean the measures and techniques that stop errors to be introduced into the finally implemented code. To facilitate our discussions, we do not make a detailed distinction of the term *error* from other relevant terms, such as *bug*, *fault*, and *defect*, but use the term *error* to mean all kinds of possible mistakes committed in software systems. Since error prevention depends on the accurate understanding of the user's requirements by the developer, appropriate documentation for requirements and design has been recognized largely as an effective solution (Dhanalaxmi et al., 2015). However, due to many challenges facing documentation in practice, this view may not always be agreed by all researchers and practitioners.

The first challenge is how the documentation process can easily involve the user for decision making. Only when the user and the developer make right decisions, can errors be prevented in documentation and code. Documentation in natural language may allow the user to read, but does not guarantee the correct understanding by the user due to ambiguities in the documentation structure and terminologies used. Documentation in a diagrammatic notation (e.g., UML) may improve the readability, but is still unlikely to avoid the ambiguity (Sibertin-Blanc et al., 2008). It also suffers from the poor efficiency and inconvenience in document construction and evolution due to the complexity and large space taken. Documentation in a formal notation is expected to achieve precision for accurate understanding, but may increase the difficulty in writing and understanding the documentation by practitioners due to mathematical complexity. Agile approach seems to take a radical attitude toward documentation in its manifesto by emphasizing the value of "working software over comprehensive documentation" (Anon, 2001). Coding from the beginning and evolution throughout the entire development process are advocated in order to easily involve the user in the process, but this paradigm lacks precise guidelines for the responsibility and technical role of each developer in software projects and suffers from inevitable frequent code modifications (Miller, 2013).

The second challenge is how the documentation process can be made short enough to curb the overall cost of projects. In general, the longer the documentation process is, the better the requirements will be understood to prevent more errors. However, the rapid changing and competitive software market seems to always put pressure on companies to discourage them to spend more time and make more efforts in requirements analysis and documentation. Therefore, the agile approach can be attractive for

☆ Editor: W. Eric Wong.

E-mail address: sliu@hiroshima-u.ac.jp.

https://doi.org/10.1016/j.jss.2021.110975
0164-1212/© 2021 Elsevier Inc. All rights reserved.

adoption, although its weaknesses can produce considerable negative impact on companies in practice. Experienced researchers and practitioners may still believe or want to believe the value of high-quality requirements analysis and documentation (Kipyegen and Korir, 2013), but the key point is whether the documentation can avoid creating unnecessary time delay for the development project.

The last challenge is how the documentation can be written in a manner that helps achieve precision and reduce unnecessary changes. The precision ensures less or no misunderstanding by relevant developers (e.g., programmers, testers) and reducing changes can help shorten the time for documentation.

To deal with the three challenges above for industrial software projects, in this paper we put forward the notion of *hybrid specification* and a *three-step hybrid specification approach* (TSHSA) to constructing hybrid specifications. Our previous collaborative projects with industry in both Japan and China on the application of the SOFL method to constructing specifications for a railway crossing system, a train interlocking system and other related systems (Liu et al., 1998, 2010; Luo et al., 2016) have helped us understand a strong demand from practitioners for using hybrid specifications in practice, but how such a hybrid specification can be achieved with the effect of preventing errors in software development has been remaining an open problem. On the basis of our previous work on agile formal engineering method Liu (2018), we have come to a conclusion that *specification-based agile development* (SBAD) is a well-balanced effective development paradigm for both software productivity and reliability. The underlining principle of the SBAD is that a comprehensible and precise specification is first achieved and then incremental implementation on the basis of the specification will be carried out. During the implementation, small cycles of coding, review, and testing will be performed. The key point here is how the specification can be achieved in the fashion that deals with the challenges existing methods face and meanwhile effectively prevent errors and lay a solid foundation for the incremental implementation. The TSHSA for constructing hybrid specifications proposed in this paper is expected to fulfill this goal.

The hybrid specification integrates semi-formal and formal specifications as well as the GUI design in order to provide a well-structured specification that is both precise enough and easy to construct. In principle, for non-critical operations of a software system, semi-formal specification is used to define their behavior, but for critical operations (e.g., safety-critical, security-critical operations), formal specification is adopted. The GUI design shows the style and necessary actions for human–machine interactions. Thus, the usability of the specification and its effect in ensuring the software quality can be well balanced to save time and enhance reliability.

The three-step hybrid specification approach advocates the strategy of building a hybrid specification by taking the three steps: informal specification, semi-formal specification with the GUI design, and formal specification. The informal specification defines the user's requirements at an abstract level, aiming to provide a foundation for the subsequent steps. The semi-formal specification aims to refine the informal specification and define the meaning of the statements in the informal specification. The GUI design is intended to explore the specific communication style between the potential system and the user. The result of this will help improve the constructed semi-formal specification. The formal specification will take advantage of both the semi-formal specification and the GUI design to further clarify the semantics of operations that provide critical services. The resultant hybrid specification, which is a combination of the semi-formal specification, GUI design, and the formal specification, is intended to serve as a solid foundation for efficient implementation and verification.

The major contributions of this paper are twofold. One is the proposal of the novel three-step approach to constructing hybrid specification through which the error prevention effect can be achieved. The other is a controlled experiment through which the effectiveness and the usability of the approach are validated.

The remainder of the paper includes five sections. Section 2 elaborates on the style and the role of hybrid specifications. Section 3 describes the new three-step approach to constructing hybrid specifications and discusses how each step can help prevent errors. Section 4 discusses how the new specification approach can be used to prevent errors. Section 5 describes a controlled experiment for the evaluation of the proposed approach. Section 6 presents the related work. Finally, Section 7 concludes the paper and points out future research directions.

2. Hybrid specification

A hybrid specification in our work is defined as a well-disciplined combination of semi-formal specification, GUI design, and formal specification. The semi-formal specification is a set of SOFL modules with the feature that all of the data items are formally defined while all of the logical statements are expressed informally in a certain style. The GUI design is a graphical representation of the human–machine interface. It shows how the user of the potential software system interacts with the system. The formal specification is characterized by all data items and operations being defined using the SOFL formal language and necessary dependency relation between processes in the same module being depicted by a condition data flow diagram (CDFD).

The rule for governing the construction of a hybrid specification specifies the role of each document in the hybrid specification. According to the rule, the semi-formal specification is used as the main technique to define the parts of the system under construction that are not critical but functionally necessary. The GUI design is used to define the way of interactions between the user and the system. The formal specification is used as an auxiliary means to define critical properties and operations as well as their data flow dependency.

Since constructing a hybrid specification is mainly to help the human developer learn and understand the user's requirements and the potential software system to be built, building what kind of specification for what part of the whole system should be dependent on the developer's engineering judgment. It is almost impossible to provide a uniform rule that is more precise than the rule mentioned above to govern the process of hybrid specification construction.

Below we discuss the characteristics of each kind of document in a hybrid specification, and then in the subsequent sections, we will discuss how a hybrid specification can be achieved efficiently and how the process of achieving such a hybrid specification can help prevent errors in early phases of software projects.

2.1. Semi-formal specification

As mentioned above, a semi-formal specification is a set of modules. Each module is a structure and mechanism for defining operations and the related data items, which is similar to a static class in Java. Each operation is called *process* and defined using *pre-* and *post-conditions* . The pre-condition defines a constraint on the input variables while the post-condition defines a constraint on the output variables. Each data item used by a process is represented by a variable that is declared with a type. There are two kinds of variables. One is the input or output variables of processes, and the other kind is called external variables, similar to the class variables in a static class of a Java program or global variables in the C language. Each type is either a built-in type

provided by the SOFL language, such as the numeric types, or a user-defined type. A type can be defined in the *type* section of the module.

The semi-formal specification is characterized by the feature that all of the variables and types are formally defined using the SOFL formal language while the pre- and post-conditions of all of the processes are defined informally, for example in natural language or natural language-based tabular notation. The invariant, which is a property that must be sustained throughout the entire system, is also defined informally. A comprehensible example to be given in Section 3 helps explain the characteristics of semi-formal specifications.

2.2. GUI design

Modern software systems usually need to provide a graphical user interface (GUI) for comprehensible and efficient interactions between the human user and the system. GUI is also an important means to enhance the *usability* of software systems. A reliable software may lose its practical value if its GUI is not well designed to allow the user to use the system in a comprehensible and efficient manner. Therefore, GUI should also be regarded as part of the important functions provided by the system. Furthermore, the GUI design will also help improve the interface design of processes in the semi-formal specification, which will facilitate the developer to write an appropriate formal specification later on for some critical processes. Our experience suggests that using a commonly used software like *PowerPoint* for ordinary developers should be a convenient choice for the GUI design, since it allows the developer to easily and quickly express the design and to carry out a simple animation to show the GUI page transitions for certain use cases. Of course, the GUI design can also be performed by using other software available.

2.3. Formal specification

Formal specification is characterized by formally defining the data flow dependency relation between processes in the same module using a CDFD and formally specifying the functionality of critical bottom level processes. Each CDFD is associated with a module in the sense that the necessary data flow dependency between processes is described by the CDFD and the functionality of each process occurring in the CDFD is either semi-formally or formally specified using pre- and post-conditions in the corresponding module. That is, some processes in the module are semi-formally specified while some other processes are formally specified. The formalization of processes is expected to clarify the ambiguities involved in the pre- and post-conditions and to detect deficiencies in the corresponding semi-formal specification.

A process needs to be formalized if and only if it satisfies the following two conditions:

- The process is a bottom level process (i.e., it has no decomposition).
- The process is a critical process.

The bottom level process is chosen because its definition entirely depends on the informal pre- and post-conditions that may involve ambiguities. A high level process with a decomposition is not chosen for formalization because its function is defined by its decomposition (therefore no need to be defined again) and is usually complex that can hardly be properly formalized. However, we do not propose to formalize all of the bottom level processes contained in the semi-formal specification. Instead, only the critical processes need to be formalized since the failure of these processes will likely cause the unacceptable damage to the user. This approach can help us achieve a reasonable balance between the cost and the quality assurance. Of course, if the budget and time are allowed to carry out a formalization for all of the bottom level processes, then it would not be a bad idea to formalize all of the processes. An example is given in Section 3 to illustrate this approach for writing formal process specifications.

3. Three-step hybrid specification approach

In this section, we describe the three-step hybrid specification approach (TSHSA) to constructing a hybrid specification. As briefly mentioned in the Introduction, we believe that the *specification-based agile development* (SBAD) is a promising development paradigm that can deal with the challenges facing the existing development paradigms. A software project using SBAD mainly has two phases: *hybrid specification* followed by *agile implementation*. The first phase aims to construct a hybrid specification that clearly and completely defines what needs to be built in the potential software system. The agile implementation phase attempts to implement the defined functions with the desired properties or constraints in an incremental and small cycle fashion. Since the discussion of the agile implementation is beyond the scope of this paper, we only concentrate our discussion on the TSHSA.

The three steps in the TSHSA are *informal specification*, *semi-formal specification together with a GUI design*, and *formal specification*, as illustrated in Fig. 1. To construct a clear and complete hybrid specification, the developer must have a clear idea of what is required by the user (the end-user). To obtain the right idea from the user using our TSHSA, an *informal specification* is proposed as the **first step** to be taken by the developer in natural language. Proposing the informal specification is not necessarily a strict writing process, but a process of evolving and confirming with the user. In the beginning, an initial, usually partial, informal specification is written by the developer, and then it is shown to the user for confirmation. After getting the feedback from the user, the developer will evolve the specification to address the concerns reflected in the feedback. Repeat this process until all of the user's requirements are recorded at an abstract level. Note that the goal of this step is to achieve the completeness of the requirements at an abstract level. In other words, this step is intended to get as complete coverage of the desired requirements as possible.

To assist the developer to fulfill this goal, the informal specification is designed to include three sections: *Functions*, *Data resources*, and *Constraints*. The functions section presents a set of functions that are required to be offered by the potential software system. Each high level function can be decomposed into low level functions. As a result, the whole functions section is usually organized as a hierarchy of functions. The data resources section gives a list of data items necessary for the realization of the functions captured in the functions section. Each data item may be an atomic data item (i.e., no decomposition) or a composite data item with several fields. The constraints section shows necessary restrictions on the functions or data resources, such as constraints on safety, security, efficiency, and reliability. All of the three sections are written in natural language but should be both concise and comprehensible.

Let us take a simplified *Universal Card Railway Services* (UCRS) as an example to explain every step of the TSHSA. The UCRS provides various railway services that can be used with a *universal card* (UC). For the sake of space and readability, we only present the necessary items derived from the original UCRS. Fig. 2 shows an informal specification for the UCRS system mentioned above. There are twelve high level functions, five data items, and two constraints given in the specification. The eleventh function (i.e., function 1.11) is decomposed into four low level functions.

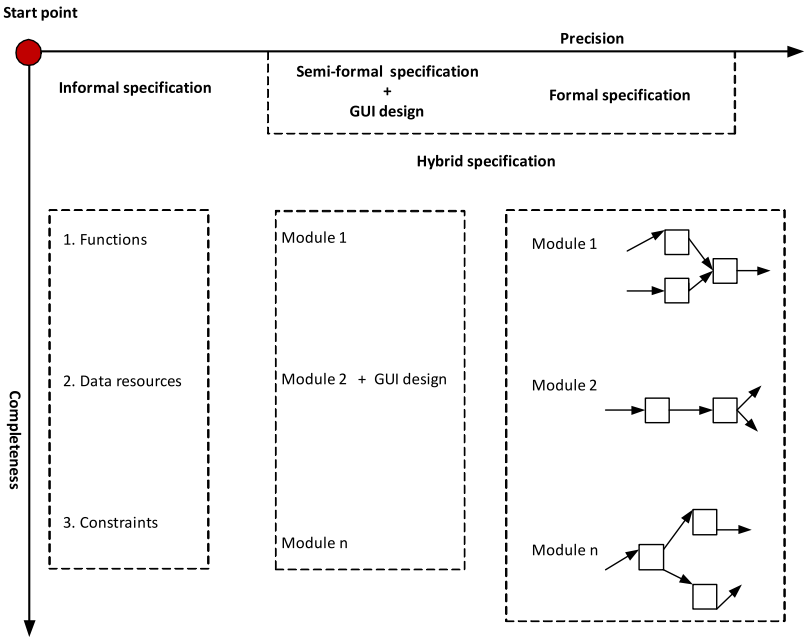

Fig. 1. The process of the TSHSA.

Informal Specification of the Universal Card Railway Services System

1 Functions
1.1 Make a new UC card.
1.2 Cancel a UC card.
1.3 Charge the UC card using cash .
1.4 Cancel the charge of the UC card and return the cash .
1.5 Charge the UC card from the related bank account .
1.6 Cancel the charge and r eturn the money to the related bank account .
1.7 Use the UC card to buy a ticket .
1.8 Cancel the ticket and return the money to the UC card .
1.9 Use the UC card to buy a fixed -period commute pass with the choice of one -month, three-month, and six-month.
1.10 Cancel the bought fixed -period commute pass
1.11 Use the UC card as a railway pass to go through the control of entrance of a railway station .
1.11.1 Check the UC card to see whether it can be used .
1.11.2 Use the UC card as a fixed -period railway pass .
1.11.3 Use the UC card as a normal non fixed -period railway pass .
1.11.4 Use the UC card as the combination of the fixed -period and non fixed -period railway pass .
1.12 Use the UC card to go out of the railway station.

2 Data Resources
2.1 Personal information (F1.1)
2.2 UC card (F1.1 to F1.12)
2.3 Station price table (F1.5, F1.12)
2.4 UC card ID table (F1.3 to F1.12)
2.5 Maximum amount constraint table (F1.3, F1.5)

3 Constraints
3.1 The maximum amount of the UC card buffer is 50,000 JPY (F1.3, F1.5, D2.5).
3.2 The money in the buffer of the UC card must first be used when buying a ticket with cash (F1.7).

Fig. 2. An informal specification for the UCRS system.

The **second step** of the TSHSA is to refine the informal specification into a semi-formal specification. To this end, three actions must be taken. One is to group the related functions, data items, and constraints given in the informal specification into a module in the semi-formal specification. In general, a function is defined by a process in the module; a data item is represented by a variable; and a constraint is either defined as an invariant or part of some process specification. Another action is to define all of the necessary types and then use them to declare all of the necessary variables in the module. It should be ensured that all of the data items given in the informal specification must be represented by some variables in the module. The last action is to specify every process using pre- and post-conditions, where the two conditions are usually expressed informally. Fig. 3 illustrates a semi-formal specification derived from the refinement of the informal specification.

For brevity, we only show the semi-formal specifications of the two processes *Railway_Entrance_Control* and *Charge_Card_With_Cash* in the module. The first process deals with the action when a passenger enters the railway station and the second process is intended to deal with charging the given *UC_card* with cash.

The process *Railway_Entrance_Control* takes *UC_card* and *entering_station* as input and sends out *pass_reject_meg* and *used_UC_card* as output. The input *UC_card* and the output *used_UC_card* are both declared with the type *UniversalCard*, which is defined as a composite type containing the four fields: *first_name*,

```
module Universal_Card_Railway_Services;

type
UniversalCard =  composed of
                  first_name: string
                  last_name: string
                  card_id: string
                  buffer: nat0 /*JPY only*/
                 end;
  commute_pass: CommutePeriod* StartDate * EndDate
  commute_start_station: Station
  commute_end_station: Station
  communte_middle_station: Station
  current_entering_station: Station
  /*recording the entering station when the passenger goes
    through the railway entrance control*/
 ChargeReceipt= composed of
                  input_amount: nat0
                  Updated_UC_card_buffer: nat0
                 end; /*As a result of charging UC card*/
 Station = string /*A station is represented by its name*/
 UCCardIDTable = set of string /*containing the identifiers of all the acceptable UC cards*/

var
ext # constrained_amounts: map ItemNames to nat0 =
    {<UC card buffer maximum> -> 50,000 JPY}
/*The content of this initial value comes from the constraint part of the informal specification*/

process Railway_Entrance_Control(UC_card: UniversalCard,
                  enteringstation: Station)
                  pass_reject_meg: string
                  used_UC_card: UniversalCard
ext rd UC_card_type_ID_table: UCCardIDTable
    rd constrained_amounts: map ItemNames to nat0
pre true
post
   The UC_card is not valid according to UC_card_type_ID_table and
   the message pass_reject_meg =  Entering is failed  is displayed and
    the UC_card is unchanged
        or
   The UC_card is valid according to UC_card_type_ID_table and used in a standard way and
   the message pass_reject_meg =  Entering is successful  is displayed and
   the UC_card is used to pay for the least charge according to constrained_amounts
        or
   The UC_card is valid according to UC_card_type_ID_table and used as a fixed period commute ticket and
   the message pass_reject_meg =  Entering is successful is displayed and
    the UC_card is unchanged
        or
   The UC_card is valid according to UC_card_type_ID_table and used in a combinatorial way and
   the message pass_reject_meg =  Entering is successful  is displayed and
   the UC_card is updated properly based on the enteringstation
end_process;

process Charge_Card_With_Cash(UC_card: UniversalCard, charge_amount: nat0)
                  charged_UC_card: UniversalCard
                  charge_receipt: ChargeReceipt
                  unsuccessful_meg: string
ext rd constrained_amounts: map ItemNames to nat0
pre true
post if the total amount of UC_card.buffer and charge_amount is
       not over the UC card's buffer maximum defined in the constrained_amounts
     then the buffer of the UC_card will be updated to reflect the increased charge_amount and
          the charge_receipt is issued properly
     else the error message represented by the output unsuccessful_meg Will be issued
end_process;



end_module;
```

Fig. 3. A semi-formal specification for the UCRS system.

last_name, *card_id*, and *buffer* (keeping the money for the card) in the **type** section of the module. The functionality of the process is defined using the pre- and post-conditions. As the specification details, it basically checks the validity of the *UC_card* and determines how the *UC_card* is used to pay for the least charge required to enter the station. The pre- and post-conditions are basically written in English, but we adopt the following three measures to enhance their readability:

- all of the input, output, and external variables declared in the process signature must properly be referred to in both the pre- and post-conditions.
- the post-condition is written as the disjunction of functional definitions (FD), where each FD describes how the output variables are defined under what condition satisfied by the input variables.
- conditional expression *if-then-else* can be adopted if only two functional definitions are required in the post-condition.

For example, in the expression "The *UC_card* is valid according to *UC_card_type_ID_table* and used in a combinatorial way and the message *pass_reject_meg* = "Entering is successful" is displayed and the *UC_card* is updated properly based on the *entering_station*", the relevant input, output, and external variables are properly referred to. Further, the post-condition is expressed as a disjunction of four functional definitions, each defining a specific functional behavior dealing with a specific "use case".

The process *Charge_Card_With_Cash* needs both the *UC_card* and *charge_amount* as input, and provides either the *charged_UC_card* and *charge_receipt* or *unsuccessful_meg* as output. The output *charge_receipt* is declared with the type *ChargeReceipt* also defined as another composite type in the module. In addition, the process also needs to get the maximum amount the card buffer is allowed to have from the external variable *constrained_amounts*, which is declared as a map from type *ItemNames* to *nat0* (natural numbers including 0). These variables and the related types are all precisely defined using the SOFL language, which reflects the formal part of the semi-formal specification. The pre- and post-conditions are given in English, but a conditional expression *if-then-else* is adopted because only two different functional definitions need to be given in the post-condition.

On the basis of the semi-formal specification, the design of a GUI for the system is undertaken as **part of the second step**. The focus of the design should be put on the appropriate reflection of a convenient and comprehensible interaction between the user and the system. To this end, we need to take the following issues into account:

- which processes defined in the semi-formal specification need direct interactions with the user of the system.
- If a process needs direct interactions with the user, what information should be provided by the user and what information should be given by the system as a response.
- What structure of the entire GUI should be designed to facilitate comprehensible human–machine interactions.

In general, the processes in the semi-formal specification can be classified into two categories. Processes in one category need direct interactions with the user while processes in the other category are not required to directly interact with the user (which are only used as internal operations in the system). To design a satisfactory GUI, all of the processes needing direct interactions must be identified beforehand. The signature of such a process will provide a basis for determining what information needs to be provided by the user and what information will be provided by the system as response, but the final decision must be approved by the end-user in agreement with the developer. Since the usability is an important quality of a software system, the structure of the GUI should be well planned to provide as much convenience as possible to the user of the system.

Following this guideline, we have designed a GUI with several inter-connected GUI pages as shown in Figs. 4 to 7. For the sake of space, we only show part of the GUI design as an example to illustrate our essential ideas. Fig. 4 shows the top page of the GUI design. The page is numbered No. 1 for reference in the design specification to be discussed later in this section. Four different services can be used through the GUI page: *Purchase Ticket*, *Purchase Pass*, *Purchase Card*, and *Charge Card*. Clicking on one of the four buttons, the corresponding service, as indicated by its name on the button, will be provided through another GUI page at the next level. For example, selecting the *Charge Card*, the user will be led to another GUI page as shown in Fig. 5. This page shows the selections of different charge amount. If one of them is selected and the cash is put into the device, then either page in Fig. 6 is displayed if the charge is successful; otherwise, the page in Fig. 7 will be given to show an error message: "The selected

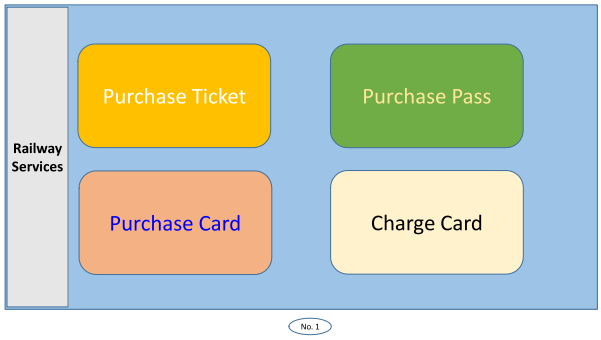

Fig. 4. The first page of the GUI design for the UCRS.

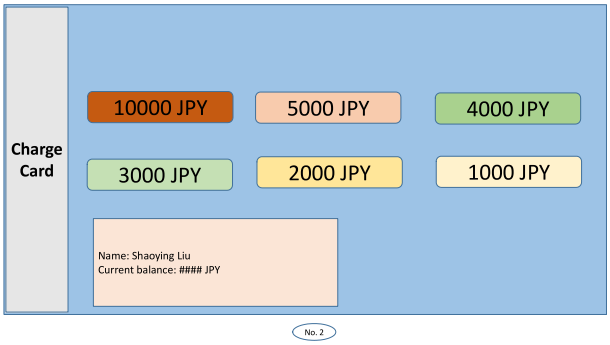

Fig. 5. The charge card page of the GUI design for the UCRS.

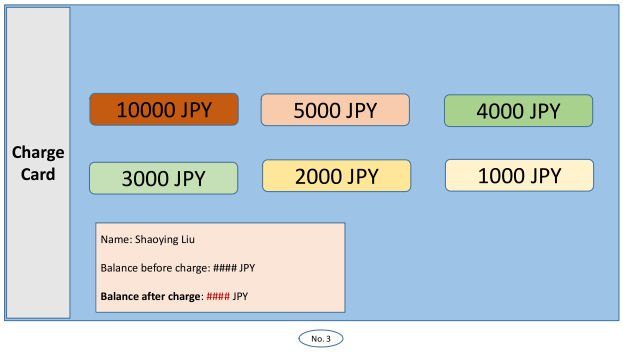

Fig. 6. The 3rd page of the GUI design for the UCRS.

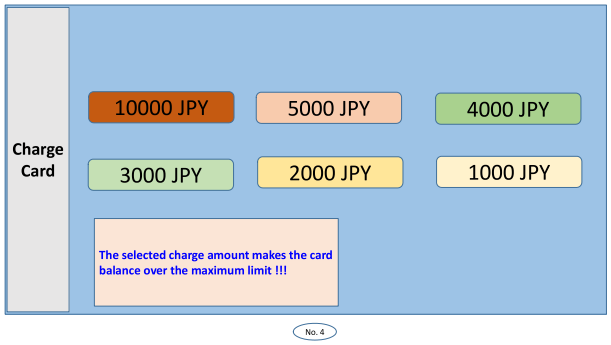

Fig. 7. The fourth page of the GUI design for the UCRS.

charge amount makes the card balance over the maximum limit !!!".

Actually, this interpretation of the GUI pages and their operational connections should be specified in the semi-formal specification. To keep a good readability, we design a new module called GUI_Design in the specification. In the module, all of the important operations and the page transitions should be clearly specified. For example, Fig. 8 presents a semi-formal specification for the GUI design. Two necessary processes are chosen as an example to be specified. The process *SelectService* describes the action to select a service from the top page. If the item *Charge Card* is selected, the GUI page will be changed to page No. 2 as shown in Fig. 5. The process *SelectChargeAmount* specifies the action to select one of the displayed amount to charge the card. If the charge is successful, the GUI page will change to page No. 3 to show the relevant details of the card; otherwise, the GUI page will change to No. 4 to issue an error message indicating that the selected charge amount makes the balance of the card buffer exceed the required maximum amount of the buffer. For brevity, we omit the discussions of the other relevant processes.

The **third step**, also the final step, is to define the data flow dependency between processes in the same module and to refine the semi-formal specifications of the critical bottom level processes into formal specifications. Since the dependency relation between processes may be part of the user's requirements and may affect the design of the process signature (e.g., the number and/or type of the input or output variables), firstly defining the dependency will be helpful for refining the semi-formal specification of the process. During the refinement, all of the ambiguities in data items and their relation with necessary operations involved must be clarified. The structure of the formal specification will also be more easily transformed into code than the natural language expressions in the semi-formal specification.

For instance, one of the use cases of the railway service system can be reflected by the process *Railway_Entrance_Control* in Fig. 8 being dependent on the process *Charge_Card_With_Cash*. Such a dependency is depicted in the CDFD in Fig. 9. Note that the output variable *charged_UC_card* of the process *Charge_Card_With_Cash* needs to be directly connected to the process *Railway_Entrance_Control* as its input in the CDFD, this will inevitably cause the inconsistency between this input variable and the original input variable *UC_card*, although these two input variables should mean the same data item. The reason why the inconsistency occurs is that both the process signatures are defined in the semi-formal specification, before the drawing of the CDFD. At that time, the focus is on individual processes rather than their connections. But when the processes are connected each other based on the data flow dependency, such an inconsistency then occurs. It is important to remove such an inconsistency in the final hybrid specification. For example, Fig. 9 shows the final hybrid specification of the UC card railway service system in which the original input *UC_card* of the process *Railway_Entrance_Control* is replaced with *charged_UC_card*, thus keeping the consistency with the CDFD.

For instance, in the UCRS system, the process *Charge_Card_With_Cash* is a bottom level process and also regarded as a functionally critical process because the failure of the process would possibly cause the loss of the money. Therefore, we refine its semi-formal specification into a formal one. As a result, a hybrid specification is constructed as shown in Fig. 10. To ensure the readability, we write the post-condition of the process *Charge_Card_With_Cash* in a disjunction of *functional definition* (FD). One FD describes that if the sum of the card buffer and the charge amount is less than or equal to the UC card buffer maximum amount, then the output variables *charged_UC_card* and the *charge_receipt* will be defined properly as shown in the formal specification. The other FD states that if the sum exceeds the UC card buffer maximum amount, then the output variable *unsuccessful_meg* will be defined to show an appropriate error message. The semi-formal specification of the pro-

```
module GUI_Design;
type
TopPage =  {<Purchse Ticket>, <Purchase Pass>, <Purchase Card>, <Charge Card>};

Amount_Items = {<10000 JPY>, <5000 JPY>, <4000 JPY>, <3000 JPY>, <2000 JPY>, <1000 JPY>};

ChargeCardPage = composed of
                    amount_sel: seq of Amount_Items;
                    message: string
                    end;

AnotherPage = given;  /*indicating that AnotherPage is a given type. */

var
UC_card: Universal_Card_Railway_Services.UniversalCard;
          /*using the type defined in the module Universal_Card_Railway_Services*/

inv
 forall[cp: ChargeCarPage] |
   cp.amount_sel = [<10000 JPY>, <5000 JPY>, <4000 JPY>, <3000 JPY>, <2000 JPY>, <1000 JPY>]
 /* The field amount_sel of any sequence in the sequence type ChargeCardPage must always be the
    given fixed sequence. */

process SelectService(top_page: TopPage)
                        charge_amount_sel: ChargeCardPage | another_page: AnotherPage
pre true
post If top_page = <Charge Card>, i.e., the charge card item on the top page of the GUI is selected,
     then the page charge_amount_sel (No. 2) showing the charge amount items will be shown
          with the necessary card information.
     else another page showing the corresponding information will be displayed.
end_process;

process SelectChargeAmount(charge_amount_sel: ChargeCardPage, amount_item: Amount_Items)
                                charge_amount_result: ChargeCardPage
ext wr UC_card
pre true
post If amount_item is one of the items in the sequence charge_amount_sel.amount_sel and
        the addition of the amount_item and the current buffer of the UC_card is not over 50000 JPY
     then the message of the resultant page (No. 3), charge_amount_result, displays the amount of
          the buffer of the UC_card after the charge
     else the message of the resultant page (No. 4), charge_amount_result, displays an error message, indicating
          that the charge is over the limit.

end_module;
```

Fig. 8. The semi-formal specification of the GUI design for UCRS.

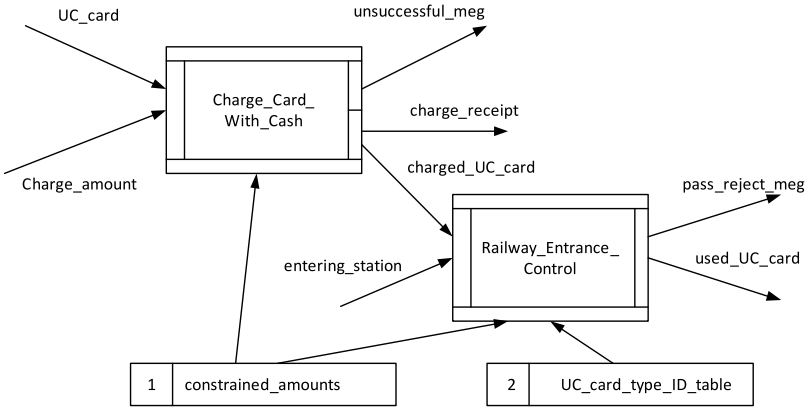

Fig. 9. A CDFD defining the dependency between two processes.

cess *Railway_Entrance_Control* remains unchanged in the hybrid specification.

4. Error prevention

In this section, we discuss how the TSHSA can be used to prevent software errors in two aspects. We first discuss the underlying principle and then use examples to explain how the related techniques work.

4.1. Underlying principle

The discussion of the underlying principle of our proposed technique is based on the following understanding of software errors in the requirements phase:

```
module Universal_Card_Railway_Services

type
UniversalCard = composed of
                  first_name: string
                  last_name: string
                  card_id: string
                  buffer: nat0 /*JPY only*/
                end;
  commute_pass: CommutePeriod* StartDate * EndDate
  commute_start_station: Station
  commute_end_station: Station
  communte_middle_station: Station
  current_entering_station: Station
  /*recording the entering station when the passenger goes
    through the railway entrance control*/
  ChargeReceipt= composed of
                   input_amount nat0
                   Updated_UC_card_buffer: nat0
                 end; /*As a result of charging UC card*/
  Station = string /*A station is represented by its name*/
  UCCardIDTable = set of string /*containing the identifiers of all the acceptable UC cards*/

var
ext # constrained_amounts: map ItemNames to nat0 =
   [<UC card buffer maximum> -> 50,000 JPY]
/*The content of this initial value comes from the constraint part of the informal specification*/

process Railway_Entrance_Control(UC_card: UniversalCard,
                    entering_station: Station)
                    pass_reject_meg: string
                    used_UC_card: UniversalCard
ext rd UC_card_type_ID_table: UCCardIDTable
    rd constrained_amounts: map ItemNames to nat0
pre true
post
   The UC_card is not valid according to UC_card_type_ID_table and
   the message pass_reject_meg = Entering is failed is displayed and
    the UC_card is unchanged
        or
   The UC_card is valid according to UC_card_type_ID_table and used in a standard way and
   the message pass_reject_meg = Entering is successful is displayed and
   the UC_card is used to pay for the least charge according to constrained_amounts
        or
   The UC_card is valid according to UC_card_type_ID_table and used as a fixed period commute ticket and
   the message pass_reject_meg = Entering is successful is displayed and
    the UC_card is unchanged
        or
   The UC_card is valid according to UC_card_type_ID_table and used in a combinatorial way and
   the message pass_reject_meg = Entering is successful is displayed and
   the UC_card is updated properly based on the entering station
end_process

process Charge_Card_With_Cash(UC_card: UniversalCard, charge_amount: nat0)
                    charged_UC_card: UniversalCard,
                    charge_receipt: ChargeReceipt,
                    unsuccessful_meg: string
ext rd constrained_amounts: map ItemNames to nat0
pre true
post UC_card.buffer + charge_amount <= constrained_amounts(<UC card buffer maximum>) and
     charged_UC_card = modify(UC_card, buffer -> UC_card.buffer + charge_amount) and
     charge_receipt= mk_ChargeReceipt(charge_amount, charged_UC_card.buffer)
        or
     not UC_card.buffer + charge_amount <= constrained_amounts(<UC card buffer maximum>) and
     unsuccessful_meg = The input charge amount makes the card buffer exceed the 50,000 JPY limit.
end_process



end_module
```

Fig. 10. The hybrid specification of the UC card railway service system.

- *If a desired function is not acquired, the software to be constructed will encounter errors* (**E1**).
- *If a desired function is acquired but defined wrongly in the specification, the software will encounter errors* (**E2**).
- *If a necessary data item is not acquired, the software will encounter errors* (**E3**).
- *If a necessary data item is acquired but defined wrongly in the specification, the software will encounter errors* (**E4**).
- *If a constraint is not acquired, the software will encounter errors* (**E5**).
- *If a constraint is acquired but defined wrongly in the specification, the software will encounter errors* (**E6**).

For the convenience in discussion, these error types are labeled with **E1**, **E2**, **E3**, **E4**, **E5**, and **E6**, respectively. To prevent these types of errors, we adopt the following techniques:

- *Achieving "symmetric functions".*
- *Achieving "processing functions".*
- *Achieving "function-based data items".*
- *Achieving "function-related constraints" and "data-related constraints".*
- *Obtaining the precision of functional definition, data definition, and constraint definition.*
- *Achieving the traceability during the refinement of specifications.*

On the basis of our observation and experience in both research and applications over the last three decades, we believe that every software system must supply "symmetric functions", in order to ensure that the services provided by running the system will cover the need of the user. By symmetric functions, we mean that if the system has one function performing some task, it must also possess another function performing the "opposite" task. For instance, in an information system, "registration" verse "cancellation" (e.g., membership management system), "input" verse "output" (e.g., general), "withdraw" verse "deposit" (e.g., in an ATM system), and "borrow a book" verse "return a book" (e.g., in a library system).

In addition, a software system must also possess functions that properly process the created data or information. For example, in the information system, for the "registration" function, there must be the functions processing the registered information, such as "modify" and "extend". The reason why both the symmetric functions and processing functions are necessary for a software system is that any necessary information in the system is once created, it must have possibilities to be processed and eventually eliminated from the system. Therefore, the system must provide the corresponding functions to deal with the operations.

Another important aspect of a system is data. Since data must be accessed or processed by functions (which will be realized by operations in the system), we must achieve the necessary data items for each given function. The technique known as *achieving function-based data items* requires that every function given in the specification must be examined to understand what data items it needs. This can help prevent errors in relation to functions and data items.

The third important aspect is the potential constraints on the functions or the data items. Therefore, we must achieve the constraints for each function and data item. The technique known as *achieving function-related constraints and data-related constraints* can be used in the way that every function and data item is analyzed to learn what potential constraints are needed. By a function-related constraint we mean a restriction condition that needs to be taken into account when defining the related functions. By a data-related constraint we mean a restriction condition on the related data item. These two concepts are difficult to formalize, therefore, whether a constraint is related to which function or data item will have to be judged by humans. Due to this characteristic, thinking of such constraints can help prevent errors in relation to functions and data items.

Since achieving symmetric functions, processing functions, function-based data items, function-based constraints and data-based constraints is concerned with the functional completeness of the potential system, it must be done at the informal specification step.

To ensure that any acquired function, data item, or constraint to be correctly defined in the specification, it is essential to make its description or definition precise enough, since ambiguous descriptions or definitions may cause misunderstanding. The techniques we use to achieve the precision is *semi-formal specification* and *formal specification*.

To ensure that all of the acquired functions, data items, and constraints in the informal specification are all properly refined

into the hybrid specification, the traceability between the informal specification and the hybrid specification must be established. Such a traceability should reflect the simple fact: each function, data item, and constraint is refined into what expressions in the hybrid specification. In general, the establishment of the traceability can hardly be performed automatically because it requires human judgment, but an appropriate software tool would be helpful.

4.2. Examples

To help the reader further understand the discussion of the underlying principle, we use specific examples to explain how each of the mentioned technique for error prevention is applied in practice.

- **Achieving "symmetric functions"**

As shown in the informal specification of the UCRS system given in Fig. 2, the function "1.1 Make a new UC card" is required and therefore we provide the symmetric function "1.2 Cancel a UC card." The function "1.3 Charge the UC card using cash." and its symmetric function "1.4 Cancel the charge of the UC card and return the cash." are both acquired. For the function "1.7 Use the UC card to buy a ticket.", a symmetric function "1.8 Cancel the ticket and return the money to the UC card." is captured.

- **Achieving "processing functions"**

A UC card is made by the function "1.1 Make a new UC card.", therefore, we need to provide desired processing functions, such as functions 1.2 to 1.12. Further, the function "1.11 Use the UC card as a railway pass to go through the control of entrance of a railway station" makes the UC card available on the entrance control device, therefore we need to provide the necessary process functions, such as functions 1.11.1 to 1.11.4.

- **Achieving "function-based data items"**

For each function given in the informal specification, we must raise the question: *what data items are required to fulfill the function and what data items must be supplied by the function as a result*. By trying to answer this question, the analyst is "forced" to learn and identify the necessary data items for the function either through communications with the user or by studying the relevant domain-related knowledge. For instance, consider the first function "1.1 Make a new UC card." Since making a UC card needs the card user's personal information, the *personal information* should be the relevant data item. Further, the function will provide a UC card as its result, therefore, the *UC card* should also be regarded as a relevant data item. Both items are written in the Data Resources section in the informal specification. The other functions in the informal specification can be analyzed similarly to obtain their relevant data items.

In fact, each data item can be used by different functions. For example, the *UC card* is used by all of the functions given in the specification. In order to clearly show the relationships between the data items and the functions, the information on the related functions can be attached to each data item in the Data Resources section. For instance, in the data item "2.2 UC card (F1.1 to F1.12)", the "F1.1 to F1.12" is such information, where F means "Function" and F1.1 means the 1.1 function in the Functions Section. Thus, this data item expression means that the UC card is a data item and it is used by functions 1.1, 1.2, …, 1.12 given in the Functions section.

- **Achieving "function-related constraints" and "data-related constraints"**

Each function proposed in the informal specification is possible to have some constraint on the service expected to provide. Similarly, each proposed data item is also possible to have some constraint on its value. In order to acquire necessary constraints, we analyze each function and each data item to identify the possible constraints. The ways of the analysis usually include communication with the user, studying the domain knowledge, and learning from the developer's experience. For example, by analyzing the function "1.3 Charge the UC card using cash." and learning from the domain knowledge, we realize that a possible constraint on the function is "3.1 The maximum amount of the UC card buffer is 50,000 JPY." In fact, in the end of the analysis process, we find that the same constraint is related to the 1.3 function, 1.5 function, and the 2.5 data item, therefore, we add the related functions and data item information, "(F1.3, F1.5, D2.5)", to the end of the above constraint expression, as shown in Fig. 2.

- **Obtaining the precision of functional definition, data definition, and constraint definition**

To prevent errors caused by misinterpretation of the requirements due to ambiguities, refining the informal expressions into a hybrid specification to ensure the precision of the requirements is an effective means. Because of the need for making the abstract and possibly ambiguous expressions into more specific and precise expressions during the refinement, the developer will be "forced" to communicate with the user to clarify the meaning of the requirements. This way will considerably help the developer to eliminate the chances of making errors. For example, the function "1.11 Use the UC card as a railway pass to go through the control of entrance of a railway station." is refined into the semi-formal specification of the process *Railway_Entrance_Control* . To complete the semi-formal specification, all of the input variables, *UC_card* and *entering_station*, and their types, *UniversalCard* and *Station*, must be properly decided and precisely defined; all of the output variables, *pass_reject_meg* and *used_UC_card*, and their types, *string* and *UniversalCard* , must be properly determined; and all of the external variables, *UC_card_type_ID_table* and *map ItemNames to nat0*, must be properly indicated and defined. Since this process is regarded as a less critical process, its pre- and post-conditions are given informally but with a comprehensible structure, that is, a disjunction of several functional definitions.

For a critical process, such as *Charge_Card_With_Cash*, not only its all variables must be precisely declared, but its pre- and post-conditions must also be formally specified to avoid errors of misinterpretation. For instance, the semi-formally specified pre- and post-conditions of the process *Charge_Card_With_Cash* in Fig. 3 is refined into the formal specification as shown in Fig. 10.

- **Achieving the traceability during the refinement of specifications**

Another kind of error is concerned with the phenomenon that some item in the informal specification is not refined into any expression in the hybrid specification. The occurrence of such an error implies that some requirements are not reflected in the hybrid specification, therefore, it would unlikely be implemented properly in the final program. To prevent this kind of error, a traceability between the informal specification and the hybrid specification must be established to show that each function, data item, and constraint is refined into what processes, variables and types, and invariants and processes, respectively. For example, the function "1.11 Use the UC card as a railway pass to go through the control of entrance of a railway station." is refined into the process *Railway_Entrance_Control*. The data item "2.2 UC card" is represented by the variable *UC_card* and its type *UniversalCard* in the hybrid specification. The constraint "3.1 The maximum

amount of the UC card buffer is 50,000 JPY" is reflected by the variable *constrained_amounts*. These three relationships should be recorded to contribute to the traceability. In the same way, all the other items in the informal specification can be linked to the corresponding parts of the hybrid specification.

5. A controlled experiment

In this experiment, we choose to compare our method with a commonly applied approach that uses Data Flow Diagram (DFD) together with Structured English to describe the requirements. Since our method uses the SOFL specification language that uses the CDFD, a formalized data flow diagram, together with the module providing textual, formal specifications for all of the involved operations, it is comparable with the DFD-Structured English approach and the result of the comparison will be useful for practice.

To make our discussion below convenient, we use the abbreviation DFD-SE to represent the DFD-Structured English approach and TSHSA to represent our method, respectively.

5.1. Experiment setup

We choose fourteen postgraduate students who have studied the course "Formal Engineering Methods for Software Development" as the subjects to join our experiment. In this course, all of the subjects studied both the methods we choose to compare in our experiment. Therefore, they all have the same technical background and experience.

We choose a simplified Stock Reservation and Purchase (SRP) system as the target system to apply both the methods for requirements analysis and specification. In the beginning, a rather informal description of the SRP system in English is given to all of the subjects and they are asked to carry out the requirements analysis and specification using the DFD-SE. After about one month later, the same subjects are asked to carry out the requirements analysis and specification using the TSHSA for the same SRP system. They are required to start with the similar informal description of the SRP system.

During the experiment, all of the subjects are required to work independently of other subjects. They are only allowed to communicate with the person who provides the informal description of the SRP system as the end-user because the subjects sometimes need to get the clarification of some ambiguous expressions in the informal description from the end-user. After the experiment is finished, an independent researcher checks both the DFD-SE and the TSHSA specifications of each subject in order to collect the necessary data for comparison. To ensure that the researcher will make fair judgments, a *checklist* containing 100 questions is made based on the user's detailed requirements and used by the researcher during his checking of both specifications. Each question on the checklist indicates a desired aspect of a required function or data item to be defined in the specification. The necessary data we have collected include the following:

- The number of correct answers to the questions raised in the checklist in both the DFD-SE specification and the TSHSA hybrid specification, respectively.
- The number of correct answers to the questions that are provided in the DFD-SE specification but not in the TSHSA hybrid specification.
- The number of correct answers to the questions that are given in the TSHSA hybrid specification but not in the DFD-SE specification.
- The time spent for completing the DFD-SE specification and the TSHSA hybrid specification, respectively.

Table 1
Table Caption.

Subjects	Correct answers in DFD-SE	Correct answers in TSHSA	Correct answers in DFD-SE but not in TSHSA	Correct answers in TSHSA but not in DFD-SE
S1	15	28	0	13
S2	47	80	0	33
S3	40	75	0	35
S4	58	84	0	26
S5	18	52	0	34
S6	50	64	0	14
S7	27	59	0	32
S8	52	83	0	31
S9	54	84	0	30
S10	55	87	0	32
S11	41	77	0	36
S12	30	54	0	24
S13	34	70	0	36
S14	75	96	0	21

Table 2
Table Caption.

Methods	Average	Median	Minimum	Maximum
DFD-SE	42.57	44	15	75
TSHSA	70.93	76	28	96

By a correct answer to a question in the specification we mean some expressions defining the desired goal expected by the question. If a correct answer to a question is not provided in the specification, we consider it as an error. By knowing how many correct answers are provided in the specification, we can learn how many errors are prevented. Having understood the difference between the numbers of the correct answers to the same set of questions in both the DFD-SE specification and the TSHSA hybrid specification, we can have an overall idea of how much improvement our method has made in detecting and preventing errors in the early phase of software development.

5.2. Experiment result and analysis

Table 1 shows the details of the fourteen subjects' results in the experiment. On the basis of the data in this table, we derive the following four aspects of the subjects' performance as a whole:

- Average number of correct answers provided in each specification,
- Median number of correct answers provided in each specification,
- Minimum number of correct answers given in each specification,
- Maximum number of correct answers given in each specification.

The details are reflected in Table 2.

From Table 2 we can see a clear difference between the two methods. Apparently, our method TSHSA is superior to the DFD-SE due to the facts that it prevents 28.36 (70.93–42.57) more errors than the DFD-SE method on average and can prevent all the errors that are prevented by the DFD-SE method. This is not surprising because the TSHSA requires the subjects to pay close attentions to the details of all the aspects of the software system under development, including functions, data resources, and constraints. However, writing the hybrid specification using the TSHSA tends to take 1.39 h longer time than that using the DFD-SE on average, but the difference does not seem to be significant and can be a trade off to the high error prevention rate of our method. Table 3 shows the details of the time spent on the

Table 3
Table Caption.

Subjects	Time (h) for DFD-SE	Time (h) for TSHSA
S1	3	5
S2	5.5	6
S3	10	10.5
S4	7	8.5
S5	3	5
S6	4	5.5
S7	4	4.5
S8	5	6
S9	5	6.5
S10	6	7.5
S11	4	6
S12	3.5	5
S13	7	9
S14	6.5	8

Table 4
Table Caption.

Methods	Time-average	Time-median	Time-minimum	Time-maximum
DFD-SE	5.25	5	3	10
TSHSA	6.64	6	5	10.5

DFD-SE specification and the TSHSA specification by each subject, respectively, and Table 4 shows the derived overall data of the consumed time for both the DFD-SE and the TSHSA methods.

5.3. Threats to validity

There are several potential threats to the validity of our experiment, including the selection of the experiment subjects, the way to arrange the target system to apply both methods, the order of applying both methods, and collection of data from the experiment results. There might be other factors to affect the result of our experiment, but since these four factors are the main threats to the validity of our experiment, we focus our discussions on these four factors.

Since both the methods we have used for our experiment must be applied by humans, the selection of the subjects becomes an important factor to affect the result of the experiment. Apparently, if the subjects have different level of the knowledge and experience with the two methods, the result might not be suitable for measuring the effect of the methods because it might be considerably affected by the subjects' personal capability. To curb this threat, we select 14 subjects who are all postgraduate students in our graduate school and have studied the course entitled "Formal Engineering Methods for Software Development" taught by the author of this paper. In this course, both methods have been well introduced in detail and all of the subjects have been required to master both methods by means of assignments, class discussions, and small projects. Therefore, it is reasonable enough to assume that all the subjects are at the same level in terms of their background knowledge and experience when they join the experiment.

Properly arranging the domain system for the application of the two methods is also important because an inappropriate way to arrange the domain system would create unwanted elements to affect the evaluation of the effect of the methods in the experiment. We choose the Stock Reservation and Purchase (SRP) system as the target system that must be specified by all of the subjects independently using the two methods, respectively. In this way, we will be able to see the effectiveness of both methods statistically. The SRP system contains the necessary functions for managing customers, managing stock information, reserving stocks, purchasing stocks, and sell stocks by customers. Considering the time and cost limits, the scale of the SRP system is deliberately designed not to be large but realistic enough for the experiment.

Using the single target system in our experiment also faces a problem of how to select the order of applying the two methods since the experience of using the first method would be utilized unconsciously to benefit the application of the second method. This can be avoided by using two different target systems each of which is specified using only one of the two methods, but this will introduce another additional factor, the nature of the target systems (e.g., scale, complexity, suitability for using the methods), that would affect the evaluation of the experiment result. To mitigate the threat in our experiment, we take two measures. One is to provide the subjects with one version of the informal description of the SRP system as the domain document for applying the DFD-SE and with another version derived from a modification of the informal description for applying the TSHSA. Although the two versions describe the same domain system, the way to describe the system in each version is different. The other measure is that we let the subjects first study the DFD-SE method and then apply the same method to the SRP system. After this, we let the subjects learn the TSHSA method and then apply the same method to the same target system, after one month later. We choose to use the DFD-SE first because the DFD-SE is a rather informal approach and its application does not have a power stronger than the TSHSA in terms of making the subjects gain deep understanding of the system. The reason why we believe the truth of this statement is that the TSHSA involves semi-formal and formal specifications that usually require the subjects to pay attention to the details of the requirements and the system, according to our long term experience in both research and applications. However, the arranged order of applying the two method in our experiment is not bias-free either. Since the use of the DFD-SE can help the subjects to gain an understanding of the requirements and the system to some extent, this experience will definitely benefit the use of the TSHSA. To mitigate the impact of the experience with the DFD-SE, we deliberately choose the two different versions of the domain document and design a one-month break between the applications of the two methods in the hope of the subjects forgetting the experience of using the DFD-SE to a certain extent when they begin to apply the TSHSA. Apparently, this effort would not completely eliminate the threat but can merely help mitigate it.

How to collect the data of the experiment in our case is also important for the comparison. To allow a fare evaluation of the effect of using both methods, we let the end-user design a checklist in advance. The checklist contains a set of questions and each question asks whether a desired aspect of the requirements and the system is properly expressed in the specification. The same checklist is applied to the result of using both methods. To avoid the misinterpretation of the questions, the end-user together with the experienced researcher work together to evaluate the effect of the two methods by collecting the data of faults and time given in Tables 2 and 3.

5.4. Discussion

In this subsection, we describe our experience in using the TSHSA and the lessons learned. We believe that both will be useful for applying the TSHSA in practice.

Like most software engineering technologies, the TSHSA must be used by humans because many human judgments are required in using it. For example, when writing the items in relation to functions, data resources, and constraints in an informal specification, how briefly each item should be described needs the developer's engineering judgment. Since the items in the informal specification mainly play the role of reminding the developer

of what to be done in the project, our experience suggests that comprehensible keywords should be used to describe the items and usually the description should be kept within one line or at most two. The real clarification of the meaning of the items will be achieved in the hybrid specification. Similarly, when choosing which processes defined in a semi-formal specification for a complete formalization, it also requires the developer to make a judgment based on the general principle of choosing the bottom critical processes.

Our another important experience is that sufficiently using all of the input, output, and external variables declared in the signature of a process in the pre- and post-conditions of its semi-formal specification can considerably help the developer achieve rather complete definition of the functionality of the process and therefore prevent many errors. The use of the variables can also help the developer achieve the comprehensibility of the pre- and post-conditions.

When writing a formal specification for the bottom level critical processes, CDFD should first be used to define the data flow dependency relation between relevant processes and then the corresponding formal specifications of the relevant processes should be constructed. The reason is that if the formal specification of the relevant processes is first written in which all of the input, output, and external variables must be determined, then there will be a risk to cause the inconsistency in the interface of the connected processes when the data flow dependency between the processes need to be defined using a CDFD. To resolve the inconsistency, the formal specification of the related processes need to be modified, which will create unnecessary workload.

We have also learned a lesson through our experiment, which is concerned with the difference in the subjects' performance using our method. Although all the subjects have achieved the result of preventing more errors using our method than the DFD-SE method, the difference between the performances of different subjects sometimes can be rather big. For example, subject S1 could only prevent 28 errors while subject S14 could prevent 96 errors as shown in Table 1. Considering the performances of the same subjects using the DFD-SE method, we can find a common phenomena that the subjects with a better performance using our method also perform better using the DFD-SE method. For instance, subject S1 prevents 15 errors using the DFD-SE while subject S14 prevents 75 errors. We cannot give a definitive reason for this situation without a rigorous study, but from our informal observation during the experiment, the performance difference seems to be attributed to personal capability in using both methods.

6. Related work

In accordance with the study in Dhanalaxmi et al. (2015), several techniques can contribute to error prevention, including requirements analysis, early prototyping, clear specification, fault-tolerant design, and defensive programming, but to the best of our knowledge, there are few results that provide detailed discussion of error prevention techniques. Nevertheless, in this section we give a brief review of the related work and compare our work in this paper with the related studies.

Walia and Carver put forward an initial requirement error taxonomy (RET) to provide structure to the process of identifying human errors in the process of developing software requirements specification (Walia and Carver, 2009) and to validate the RET through experiments. To overcome the deficiency of the RET, Hu et al. propose an improved model known as Human Error Taxonomy (HET) and analyze through an empirical study if the HET is effective in classifying errors and for guiding developers to find additional faults (Hu et al., 2016). The result shows that the HET is effective in identifying and classifying requirements errors and faults, thereby helping to improve the overall quality of the software requirement specification. Warriach et al. briefly propose a proactive fault-prevention framework to help predict potential low-level hardware, software and network faults and prevent them via dynamic adaptation (Warriach et al., 2014). The proposed technique remains only as a framework and was not discussed in detail. Nixon et al. describe an idea of utilizing functional specification language and its environment to investigate and express important properties of parallel fault tolerant systems through which requirements-related faults may be identified Nixon et al. (1994), but does not discuss how the language and the environment can be used to find appropriate properties and prevent errors during the process. Chakraborty et al. suggest a "live" prototyping technique for requirement elicitation that emphasizes the importance of breaking down the prototyping activity into five steps from the pre-prototype up to the fourth prototype (Chakraborty et al., 2013). Each step prototype is intended to capture certain kind of requirement-related information from the user. Alagar et al. argue that the formal specification techniques can help achieve trustworthiness and dependability of software systems and present a case study on the specification of a robot-based assembly system to discuss how the formal method VDM can be used to write a precise specification for a complex software system (Alagar et al., 1994). Saha reviews and discusses many fault-tolerant design techniques for software systems, including design diversity, the recovery block scheme, the multiple-version programming scheme, and the n self-checking programming scheme (Saha, 2005). These techniques can be used for fault prevention, but mostly for run-time failure prevention. Sahu and Tomar present a way of using defensive programming to handle the web application vulnerabilities identified from an attack tree analysis (Sahu and Tomar, 2014). The defensive programming results in a secure code which will be able to prevent coding vulnerabilities, loss of services, compromise of confidentiality and damage to the systems.

Compared with the related work above, our TSHSA offers a unique systematic way to capture complete and precise requirements and to prevent requirement-related errors by means of informal specification, semi-formal specification together with a GUI design, and formal specification for critical operations. The TSHSA not only possesses the power to prevent errors, but can also help construct a hybrid specification as a firm foundation to facilitate the implementation.

7. Conclusion and future work

We have put forward a novel three-step hybrid specification approach (TSHSA) to requirements analysis and definition, and discussed how this approach can be used to prevent software errors in the early phases of a software development project. The TSHSA is characterized by gradually gaining the understanding of the desired requirements through the building of an informal specification, a semi-formal specification together with a GUI design, and a formal specification for the selected bottom level critical operations. During this process, several specific techniques can be properly applied to prevent errors, particularly the requirement-related errors. We have discussed the principle of each technique and demonstrated its usage with examples. To validate the TSHSA, we have conducted an controlled experiment. The result shows that the TSHSA can help the developers effectively prevent requirement-related errors.

To support the efficient use of the TSHSA in practice, we plan to build a supporting tool for the TSHSA in the future. The tool

is expected not only to support the TSHSA, but also the specification-based agile development paradigm. The tool is also expected to take advantage of the technologies of expert systems and machine learning in artificial intelligence to enhance significantly software productivity and quality.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The author would like to thank all of the students for their contributions in the controlled experiment. This work was supported by JSPS KAKENHI Grant Number 26240008.

References

Abu-Faraj, M., 2019. A survey on software quality assurance. J. Eng. Appl. Sci. 14 (15), 5111–5122.

Alagar, V.S., Periyasamy, K., Ramanathan, G., 1994. Formal specification techniques for complex software systems. In: 9th Annual International Conference on: 'Frontiers of Computer Technology'. IEEE Xplore, Singapore pp. 1008–1013.

Anon, 2001. Manifesto for Agile software development.

Atlee, J.M., Gannon, J., 1993. State-based model checking of event-driven system requirements. IEEE Trans. Softw. Eng. 19 (1), 24–40.

Chakraborty, N.R., Latif, S., Islam, Y.M., 2013. Requirement elicitation: A "live" prototyping approach. Int. J. Comput. Appl. 72 (13), 38–44.

Clarke, E.M., Grumberg, O., Peled, D., 2000. Model Checking. MIT Press.

Dhanalaxmi, B., Naidu, G.A., Anuradha, K., 2015. A review on software fault detection and prevention mechanism in software development activities. IOSR J. Comput. Eng. 17 (6).

Eakman, G., Reubenstein, H., Hawkins, T., Jain, M., Manolios, P., 2015. Practical formal verification of domain-specific language applications. In: NASA Formal Methods - 7th International Symposium. NFM 2015, In: LNCS, vol. 9058, Springer, Pasadena, USA, pp. 443–449.

Hu, W., Carver, J.C., Anu, V.K., Walia, G.S., Bradshaw, G., 2016. Detection of requirement errors and faults via a human error taxonomy: A feasibility study. In: ESEM 2016. ACM Press, Ciudad Real, Spain, pp. 1–10.

Jha, D., Bhagat, V., Bidua, A., 2014. Survey on software quality assurance. Int. J. Comput. Sci. Commun. 5 (1), 147–149.

Kipyegen, N.J., Korir, W.P.K., 2013. Importance of software documentation. Int. J. Comput. Sci. Issues 10 (5), 223–228.

Liu, S., 2016. Testing-based formal verification for theorems and its application in software specification verification. In: Proceedings of the 10th International Conference on Tests and Proofs. TAP 2016, In: LNCS, vol. 9762, Springer, Vienna, Austria, pp. 112–129.

Liu, S., 2018. Agile formal engineering method for software productivity and reliability. In: Proceedings of 2018 Central and Eastern European Software Engineering Conference Russia. CEE-SECR 2018, ACM Press, pp. 64–69.

Liu, S., Asuka, M., Komaya, K., Nakamura, Y., 1998. An approach to specifying and verifying safety-critical systems with the practical formal method SOFL. In: Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'98, IEEE Computer Society Press, Monterey, California, USA, pp. 100–114.

Liu, S., Chen, Y., Nagoay, F., McDermid, J., 2012. Formal specification-based inspection for verification of programs. IEEE Trans. Softw. Eng. 38 (5), 1100–1122.

Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., Nakajima, S., 2010. Automatic transformation from formal specifications to functional scenario forms for automatic test case generation. In: 9th International Conference on Software Methodologies, Tools and Techniques. SoMet 2010, IOS International Publisher, Yokohama City, Japan, pp. 383–397.

Liu, S., Nakajima, S., 2020. Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. IEEE Trans. Softw. Test.

Luo, J., Liu, S., Wang, Y., Zhou, T., 2016. Applying SOFL to a railway interlocking system in industry. In: Proceedings of the 6th International Workshop on SOFL+MSVL. SOFL+MSVL 2016, In: LNCS, vol. 10189, Springer, Tokyo, Japan, pp. 160–180.

Miller, G.J., 2013. Agile problems, challenges, and failures. In: PMI Global Congreess 2013.

Nixon, P., Birkinshaw, C., Croll, P., Marriott, D., 1994. Rapid prototyping of parallel fault tolerant systems. In: Second Euromicro Workshop on Parallel and Distributed Processing. IEEE Xplore, Malaga, Spain.

Parnas, D.L., Weiss, D.M., 1985. Active design reviews: Principles and practices. In: Proceedings of the 8th International Conference on Software Engineering, pp. 215–222.

Saha, G.K., 2005. Approaches to software based fault tolerance - a review. Comput. Sci. J. Moldova 13 (3), 193–231.

Sahu, D.R., Tomar, D.S., 2014. Defensive programming to reduce PHP vulnerabilities. Int. J. Adv. Comput. Netw. Secur. 4 (2), 71–75.

Sibertin-Blanc, C., Hameurlain, N., Tahir, O., 2008. Ambiguity and structural properties of basci sequence diagrams. Innov. Syst. Softw. Eng. (4).

Walia, G.S., Carver, J.C., 2009. A systematic literature review to identify and classify software requirement errors. Inf. Softw. Technol. 51 (7), 1087–1109.

Wang, R., Liu, S., 2018. TBFV-SE: Testing-based formal verification with symbolic execution. In: 2018 IEEE International Conference on Software Quality, Reliability and Security. QRS 2018, IEEE Press, Lisbon, Portugal, pp. 59–66.

Warriach, E.U., Ozcelebi, T., Lukkien, J.J., 2014. Fault-prevention in smart environments for dependable applications. In: 2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems. IEEE Xplore, London, UK, pp. 183–184.

Shaoying Liu is currently a Professor of Software Engineering at Hiroshima University, Japan. He received the Ph.D. in Computer Science from the University of Manchester, U.K in 1992. His research interests include Formal Methods and Formal Engineering Methods for Software Development, Specification Verification and Validation, Specification-based Program Inspection, Automatic Specification-based Testing, Testing-Based Formal Verification, and Intelligent Software Engineering Environments. He has published a book entitled "Formal Engineering for Industrial Software Development" with Springer-Verlag, twelve edited conference proceedings, and over 200 academic papers in refereed journals and international conferences. In recent years, he has served as the General Chair of QRS 2020 and ICFEM 2017, Steering Committee Chair of ICECCS, and PC member for numerous international conferences. He is currently an Associate Editor for IEEE Transactions on Reliability and the Journal of Innovations in Systems and Software Engineering, respectively. He is IEEE Fellow, BCS Fellow, and member of JSSST and IPSJ.