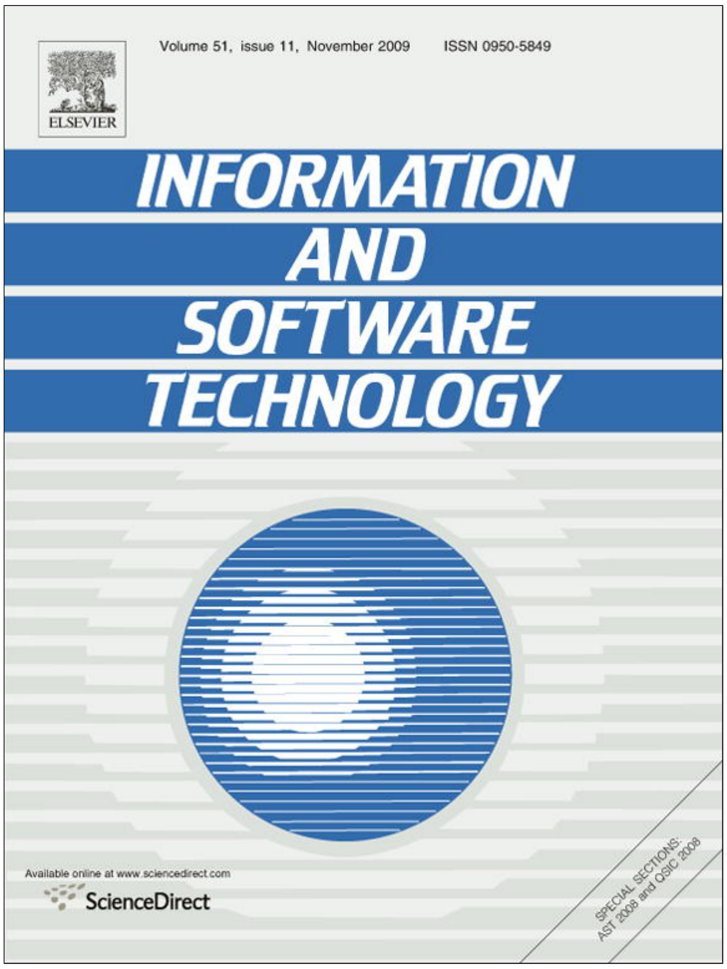

Volume 51, issue 11, November 2009
ISSN 0950-5849
ELSEVIER
INFORMATION AND SOFTWARE TECHNOLOGY
Available online at www.sciencedirect.com
ScienceDirect
SPECIAL SECTIONS: AST 2008 and QSIC 2008

Information and Software Technology 51 (2009) 1565–1572

Contents lists available at ScienceDirect

Information and Software Technology

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

Integrating top-down and scenario-based methods for constructing software specifications [☆]

Shaoying Liu

Hosei University, Tokyo, Japan

A R T I C L E I N F O

Article history:
Available online 16 July 2009

Keywords:
Software specification
Software design
Data flow
Structured method
Scenario-based method

A B S T R A C T

How to achieve a complete and consistent software specification by construction is an important issue for software quality assurance but still remains an open problem. The difficulty lies in the fact that the assurance of the completeness needs user's judgments and the specification keeps changing as requirements analysis progresses. To allow the user to easily make such judgments and to reduce chances for creating inconsistencies due to frequent specification modifications, in this paper we describe an intuitive, formal, and expressive specification method that integrates top-down decompositional and scenario-based compositional methods. The decompositional method is used at an informal level with the goal of achieving a complete coverage of the user's functional requirements, while the compositional method is used to precisely define the functionality of each scenario and to construct complex scenarios by composition of simple scenarios in a formal, intuitive language called SOFL. Combination of the decompositional and compositional processes can facilitate the analyst in completing a specification in a hierarchical structure. We present an example to illustrate how the integrated method is used in practice and describe a software support tool for the method.

1. Introduction

Achieving complete and consistent functional specifications by *construction* rather than merely by *verification* can reduce costs and time for error detection and specification modification. Although no guarantee can be made to achieve the goal by construction along, effective analysis and specification methods can help to pursue the goal. The question is what methods are capable of fulfilling this mission.

Many methods have been developed so far, and they are mainly classified into three categories from the construction techniques' point of view. The top-down decompositional methods (TDDM) (or structured methods [2]) support the principle of "functional abstraction and decomposition". It allows the analyst to have a global view of the whole system all the time and to focus on the decomposition of a high level module whenever necessary. Scenario-based methods (SBM) focus on the description of functional scenarios and their compositions, where a functional scenario (or simply scenario) represents a system service possibly implemented by a sequential operations [3]. Compared to TDDM, SBM naturally supports a bottom-up approach to the construction of specifications, but still keeps the function-driven feature as TDDM has. Object-oriented methods (OOM), however, suggest the paradigm of leading to descriptions of operations from identification and descriptions of data (objects) [4]. By OOM, the developer starts with looking for objects and defining their corresponding classes necessary for the software system to be developed, and then proceeds to establish the associations among objects and classes [5].

Many evidences show that function-driven methods, especially based upon data flow diagrams, are suitable for requirements analysis and specification while object-oriented methods are suitable for design and programming [6,7]. The reason is that clients are usually familiar with application domains in terms of functional services provided by the relevant systems rather than in terms of objects and their relations. The latter is more concerned with design, an action to determine the architecture of the system and algorithms for operations used. In order to achieve good quality of software systems for maintenance, such as reusability and information hiding, object-oriented methods are very helpful. That is perhaps one of the major reasons for some development methods based upon UML to suggest use case analysis as the first step of development and class diagrams and sequence or activity diagrams next [8]. However, the use case diagram in UML does not provide mechanisms for the description of complex functional scenarios and for structuring all the scenarios in a uniform framework. Therefore, it may not allow analysts to carry out a thorough analysis of requirements and to effectively construct the specification.

Our experience in developing the SOFL formal engineering method [9] over the last eighteen years suggests that a function-driven analysis and specification of requirements in an intuitive

[☆] This work is supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research (No. 18500027). A preliminary version of this paper was presented at the 8th International Conference on Quality Software (QSIC 2008) [1].

E-mail address: sliu@hosei.ac.jp

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.06.015

and formal language, such as SOFL (Structured Object-Oriented Formal Language), an integration of a formalized Data Flow Diagrams and VDM-SL [10,11], is beneficial to the construction of object-oriented program systems [12]. However, how to intuitively and effectively carry out a function-driven analysis and specification still remains an open problem.

In this paper, we put forward an integrated method that combines a top-down decompositional method and bottom-up scenario-based method for requirements analysis and specification based upon data flow modeling. We choose the formalized data flow diagram notation employed in SOFL, called Condition Data Flow Diagram (CDFD), as the notation to support the integrated method because CDFD allows us to concentrate on the description of "what to do" in data flow modeling rather than "how to do it" in control flow modeling. The top-down method is used at an informal and abstract level as a way of communicating with the user for requirements acquisition, while the scenario-based method is used at a formal and abstract level to concentrate on the definition of each specific scenario (including operations and their relations involved in the scenario) and their composition for more complex scenarios. The top-down method helps, rather than guarantees, to achieve a full coverage of the functional aspects of the potential system, because it allows the analyst and the user to have a global view of the entire system all the time during the analysis, while the scenario-based method helps to clarify the meaning of the informal and abstract description of each function, and to compose the defined scenarios into more complex ones. A potential benefit of the integrated method is to increase the possibility of creating consistent and valid specifications.

The rest of the paper is organized as follows. Section 2 describes the integrated method and discusses how it can be used in practice with an example. Section 3 briefly describes a software support tool for the method. Section 4 overviews the related work and compares it with our method. Finally, in Section 5 we conclude the paper and point out future research directions.

2. The integrated method

A necessary condition for obtaining complete requirements is to ensure that all the potentially desired functions, data resources necessary for realizing the functions, and constraints on both the data resources and the functions are captured and documented in the specification. Since this requires the analyst to always keep a global view of the system during the analysis phase, the top-down decompositional method is usually effective. However, if the top-down analysis is carried out in a detailed and precise manner, it may meet challenges in coping with frequent modifications of high level structures and in ensuring the consistency between high level operations and their decompositions. This problem is especially severe if the analysis is carried out based upon data flow diagrams [2]. To avoid this problem, in our method we restrict the top-down analysis only to an informal and abstract level and thus create a comprehensible document for efficient communications and discussions between the analyst and the client to achieve desired and complete functions. Even though this approach provides no guarantee for the completeness of desired functions, our experience suggests that it helps significantly. The result of the analysis is a hierarchical structure of operations in specification and each operation focuses on a single independent function. The hierarchy provides a guideline for precisely defining the potential behavior of each related operation. We adopt a scenario-based method for this task because a scenario allows the analyst to concentrate on the analysis and description of a single independent function and defined scenarios can be composed into high level operations for more complex scenarios. During the composition rigorous review [13] is employed to ensure the consistency and completeness of the specification.

In our method, we use the SOFL specification language to define operations, scenarios, and their compositions, and the fundamental principle of the method is applicable to many other analysis notations. A SOFL specification mainly consists of hierarchical CDFDs and modules where each CDFD is associated with one module. A CDFD is a directed graph that can be constructed using the components *processes*, *data flows*, *stores*, and *conditional structures* while its associated module offers formal textual specifications of all the components. A process models a transformation from its input to output; a data flow indicates a data item moving from one process to another; and a store represents a data depository (such as a file or database). A process is similar to an operation in VDM [14], but differs in that it may take input data flows and produce output data flows exclusively. This means that a process is allowed to receive several groups of input data flows, but it consumes only one of the groups in order to execute the process at a time to produce one of the output data flow groups (such a process is said to have multiple ports, each port receiving or sending one group of data flows). Data flows and stores are represented by variables in a CDFD; they are declared in the corresponding module with appropriate types. A module plays the same role as both data dictionary and process specification in conventional data flow diagrams [2], but differs in that all the definitions are given in a formal textual notation to achieve preciseness and conciseness: all the data flows and stores are defined with types and/or invariants, and processes are defined in pre- and postconditions.

It is worth mentioning that the combination of CDFDs with their associated modules as well as abundant built-in data types makes the whole SOFL language a unified notation with practical simplicity for modeling information systems, reactive systems, and object-oriented systems. CDFDs together with their associated modules can easily describe the collection, storage, processing, and presentation of data which are the common functional features of information systems. Due to the operational semantics of CDFD based upon the concept of "data availability" (i.e., available data items required enables a process) and "firing" condition (i.e., the condition for executing the process), SOFL possesses the same capacity as finite state machine (based) notations (e.g., statecharts [15]) for modeling event driven systems, such as reactive systems (e.g., vending machine) and control systems (e.g., railway crossing controller) [16]. Treating all the compound data flows, data stores, and processes (especially those with multiple ports) as objects, CDFDs can easily describe message communications among objects and object decomposition (and inversely composition) [17], but unlike the conventional object-oriented languages, CDFDs describe message communications between processes based upon data flows rather than method invocations. There are two types of data flows in CDFDs known as *active data flow* and *control data flow*, respectively. An active data flow, represented by a solid directed line, provides both the "power" to enable a process and actual data items to be used by the process, while a control data flow, represented by a dotted directed line, has only the "power" to enable a process, but not to provide actual data for use by the process. Thus, the active data flow is suitable for simulating a method invocation with actual arguments while the control data flow is suitable for simulating a method invocation with no argument. The details of the SOFL language and method are introduced in [9].

2.1. Top-down decompositional analysis

By the top-down method, we first abstract the entire system into the top-level operation, and then concentrate on its decomposition by considering how many low level operations should be included in the system with little need to define their relationships.

Each low level operation is an abstraction of a sub-system of the entire system, and it may be again treated as a target for decomposition. Continue such a process until all the bottom-level operations, which have no decompositions, are simple enough for a single independent function. The result of such an analysis is a hierarchy of operations. There are at least two important things needed to bear in mind for building a hierarchy of operations. One is that the description of each operation should be as concise and precise as possible, but should also be kept informal and abstract. Another important point is that the decomposition of the operation should cover all the desired functional aspects of the operation. This can be achieved by studying the application domain and communicating with the user, although 100% assurance may not be provided. To facilitate analysts in understanding application domains, domain modeling seems to have been recognized as an effective approach. Several techniques have been developed for domain modeling, which include ontology-based modeling [18], object-oriented modeling [19], and knowledge-based modeling [20]. Since domain modeling is beyond the scope of this paper, we omit the detailed discussion here.

Let us consider an example of a simplified IC card system for purchasing railway tickets. The original system was constructed based upon the requirements for Japan Railway (JR) services in Tokyo area by a group of three researchers in our recent project. The purpose of the system is to allow a customer to purchase railway tickets using an IC card. We first abstract the system into the top-level operation as shown in Fig. 1. The top-level operation numbered 0, written under the title "Functions", is then decomposed into three sub-operations numbered 1, 2, and 3. Note that the numbers of the functions do not indicate the order of their "executions" but merely signs to distinguish them. In other words, all the functions listed are treated as a set rather than a sequence of functions. At this stage the precise relationship between any two functions is not taken into account, but it will be defined when the informal functions are refined in the corresponding formal specification later. The operation 1 is regarded as a bottom-level operation, while operations 2 and 3 are all decomposed into smaller operations. For example, operation 2 is decomposed into operations 2.1, 2.2, and 2.3. Keeping the hierarchy well structured but informal facilitates the communication between the analyst and the user, which is very important for requirements acquisition.

In addition to the operation aspect of a system, we also need to capture and document the data aspect of the system, as well as possible constraints on operations and data. For example, the data aspect written under the title "Data resources" in the informal specification in Fig. 1 lists necessary data items, such as *buffer* and *current_accounts*, and the potentially related operations in which they are used. The constraints written under the title "Constraints" describes all necessary constraints on operations or data resources that have been captured; they are usually derived from the policy for managing and operating the real system or from the requirements of the operational environment. Due to the potential ambiguity and/or the limited abstract description of the functions and data resources, it is almost impossible to completely understand the precise relationships between the constraints and the functions or data resources, but if such relationships become clear, the analyst is encouraged to write them explicitly in the informal specification. For example, the constraint numbered 1 states that the maximum amount of the buffer of the IC card cannot be greater than 100,000 Japanese yen (to prevent big loss of money when the card is lost) and it is related to data item *buffer* and the operations 2.2, 2.3.3, and 2.3.4.

2.2. Scenario-based specification

Each bottom-level operation in the hierarchy is treated as a process in SOFL because it is intended to define a single independent function. A scenario is represented as a sequence of processes where the adjacent processes on the scenario interact with each other based upon data flow communication. In order to provide a precise meaning of a single scenario, we need to formally specify each process involved in the scenario. When constructing the specification for a process, the relevant data resources and constraints on data or operations described in the informal specification must be taken into account. To this end, we adopt the *module* structure in the SOFL specification language for process specifications.

Let us consider the IC card system for purchasing railway tickets as an example. We first define each bottom-level operation as a process in SOFL, which has both a graphical representation and formal specification, and then define scenarios in CDFD that integrate the processes. Such a scenario corresponds to a high level operation (with a decomposition) in the informal specification. For example, we define the bottom level operations 3.1, 3.2, and 3.3 in Fig. 1 as the three processes *Check_Buffer*, *Update_Buffer*, and *Issue_Ticket*, respectively. Then, we compose the three processes based upon data flow communications into a scenario represented by the CDFD in Fig. 2. The scenario defines the high level operation 3 in Fig. 1 and represents the following sequential execution. When the input data flow *ticket_price*, represented by the corresponding directed line, becomes available, the process *Check_Buffer* executes and produces either a *fail_mesg4* (if the buffer has less money than the *ticket_price*) or passes the *ticket_price* to the process *Update_Buffer*. This process then produces the data flow *issue_com* (standing for issue command). The *issue_com* is then transmitted to the

Functions

0. An IC card system for buying tickets or charging the card.
 1. Select services provided by a customer.
 2. Charge card.
  2.1 Select the type of charging the card.
  2.2 Charge card with cash.
  2.3 Charge from the bank account.
   2.3.1 Authentication.
   2.3.2 Check amount.
   2.3.3 Charge with the amount from the account, which is similar to Charge with cash.
   2.3.4 Update account after the charge.
 3. Buy tickets.
  3.1 Check the buffer of the card.
  3.2 Update the buffer of the card.
  3.3 Issue a ticket.

Data resources

1. buffer (used by operations 2.2, 2.3, 2.3.2, 2.3.3, 2.3.4, 3.1, 3.2)
2. current_accounts (used by operations 2.3.1, 2.3.2, 2.3.3, 2.3.4)
3. tickets_available (used by operations 3.3)

Constraints

1. The amount of the card buffer cannot be greater than 100,000 yen (related to data item buffer and operations 2.2, 2.3.3, and 2.3.4.).
2. The requested amount for charging the card must not be greater than the account balance if the card is charged directly from the bank account (related to data item current_accounts and operation 2.3.3).

Fig. 1. The result of an informal and abstract top-down analysis.

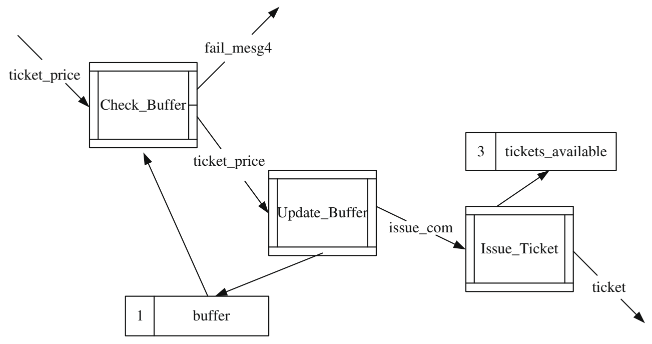


Fig. 2. The scenario for buying ticket.

process *Issue_Ticket* to issue the *ticket*. The directed line from the store *buffer* to the process *Check_Buffer* means that the content of *buffer* is read (and used) by the process (e.g., to compare it with the *ticket_price*) and the directed line from the process *Update_Buffer* to the store *buffer* indicates that the content of *buffer* is updated by the process.

The module for defining the components of the scenario, such as the store *buffer* and all the processes (as well as their associated data flows), is given as follows.

```
module Buy_Ticket_Scenario;
type
Ticket = given;
var
buffer: nat0
   /* buffer represents the amount of money in
Japanese yen*/
  tickets_available: seq of set of Ticket;
  /*tickets_available represents a sequence each of
which is a set of tickets */
process Check_Buffer(ticket_price: nat0)
            fail_mesg4: string|ticket_price: nat0
ext rd buffer: nat0
postif ticket_price<=buffer
      then ticket_price=ticket_price~
      else fail_mesg4="Your card has no enough."
money for the ticket
end_process;
process Update_Buffer(ticket_price: nat0)
            issue_com: nat0
/* issue_com indicates the type of the ticket */
ext wr buffer: nat0
post buffer=buffer - ticket_price~
end_process;
process Issue_Ticket(issue_com: nat0) ticket:
Ticket
ext wr tickets_available: seq of set of
post ticket=get(tickets_available [issue_com])
     /*get one element from the set */
end_process
end_module
```

The module is named *Buy_Ticket_Scenario* and is composed of type declaration (in the **type** section), store variable declaration (in the **var** section), and process specifications. *Ticket*=**given** means that *Ticket* is a given type whose values can be treated as tokens. Such a given type needs to be refined into a concrete data structure during the implementation phase. Each process is specified using pre- and postconditions. For example, the precondition of the process *Check_Buffer* is *true* (omitted as convention), indicating no specific constraint on the input data flow *ticket_price* and the store *buffer* is required before its execution, and the postcondition describes that if *ticket_price* is less than or equal to the buffer, the output data flow *ticket_price* is produced with the same value as the input data flow *ticket_price* (denoted by the decorated variable *ticket_price~*); otherwise, the error message *fail_mesg4* is generated.

When constructing the specification for a process, we need to ensure that every output data flow and every **wr** (writable) type of external variable of the process is defined in every possible situation. We also need to ensure that the whole process specification is feasible in order to guarantee the existence of an implementation for the specification. A process specification is feasible iff for every input (including all the values for the input data flows and external variables) satisfying the precondition, there must exist an output (including all the values for the output data flows and external variables) that satisfies the postcondition.

Similarly, we also build the scenario in Fig. 3 to specify the operation 2.2 in Fig. 1 and then compose it with other relevant processes into the scenario in Fig. 4 to specify the operation 2.3 in Fig. 1. These two scenarios are then composed together with another process *Select_Charge_Type* into a more complex scenario in Fig. 5 to specify operation 1 in Fig. 1. We compose the scenarios in Figs. 2 and 5, which result from the previous compositions, into the high level scenario represented by the CDFD in Fig. 6. In this CDFD the dotted directed line named *charge_com* denotes a *control data flow*. Finally, the scenario is abstracted into the top level CDFD in Fig. 7. During the abstraction the input data flow *ticket_price* of the process *Buy_Ticket* is treated as an input data flow that should be introduced at this level CDFD while the data flow *service_sel* is treated as an input data flow that should be introduced at the higher level CDFD. This is because that *ticket_price* needs another input data flow produced by a proceeding process in the same CDFD to enable the process *Buy_Ticket* while *service_sel* alone can enable and execute the process *Select_Service* (and therefore the whole CDFD). For this reason, *ticket_price* is treated as a local data flow to the CDFD in Fig. 6 and hence it is not exposed in the CDFD in Fig. 7. Since all the corresponding module specifications for the composed scenarios in the figures mentioned above can be constructed in the same principle as used for writing the module specification for the CDFD in Fig. 2, we omit them here for brevity.

An interesting issue here is how to compose the existing scenarios into a more complex scenario. In general, each of the existing scenarios is abstracted into a high level process and all the high level processes resulting from the abstraction are then integrated into a CDFD to define a more complex scenario. During this process new but necessary data flows and stores may be introduced to the CDFD. The high level processes of the existing scenarios are treated as components for the construction of more complex scenarios, but how the components are put together using what data flows need to be worked out by the analyst based upon his or her understanding of the required function. For example, to form the scenario in Fig. 5, we first abstract the scenario in Fig. 4 into the high level process called *Charge_from_Bank_Account* in Fig. 5 where all the input

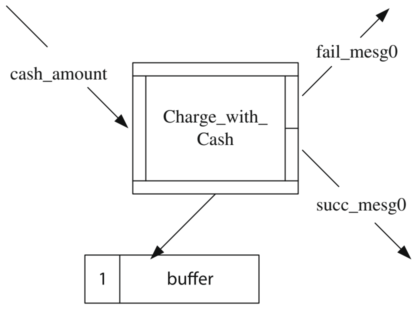


Fig. 3. The scenario for charging with cash.

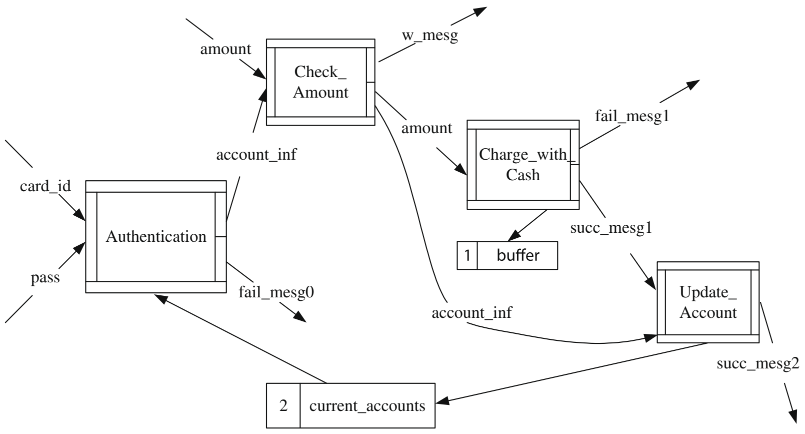


Fig. 4. The scenario for charging from bank account.

data flows of the scenario, including *card_id*, *pass*, and *amount*, are treated as input data flows of the process, and all the output data flows of the scenario, including *w_mesg*, *fail_mesg0*, *fail_mesg1*, and *succ_mesg2*, are treated as the output data flows of the process. Since the output data flows are produced exclusively by the scenario, they are abstracted into the output data flows outgoing from different output ports (each port is represented by a narrow rectangle on the right side of the graphical representation of the process). The process models a transformation from the input data flows to one of the output data flows outgoing from different output ports.

The resulting specification composed of both the hierarchical CDFDs and modules can be implemented either using a structured programming language like C (for embedded systems, for example) or an object-oriented programming language like Java (for information systems, for example). Generally, a process can be implemented as a function in C and a CDFD as well as its associated module can also be implemented as a function in which other functions are called. When transforming a SOFL specification to an object-oriented program, several approaches can be taken. One of the straightforward approaches is to implement each module as a class and the associated processes as methods. We can also treat each data flow, store, and even process with multiple input ports as an object and define their classes in the program. Since the transformation issue is beyond the scope of this paper, we do not discuss it in detail here. The reader can refer to our previous publication [9] for details.

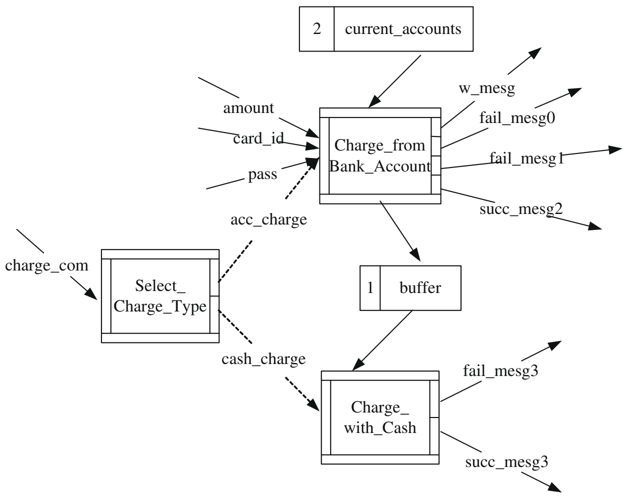


Fig. 5. The charge card scenario.

2.3. Rigorous review for consistency and validity

The composition of scenarios is usually done manually by the analyst, albeit possibly with a tool support. This is because human decisions for process integration are required. Errors are therefore

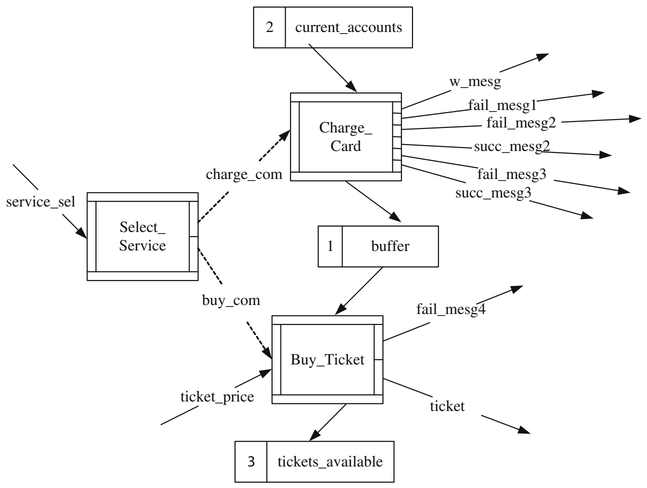


Fig. 6. The integrated Charge_Or_Buy_Ticket scenario.

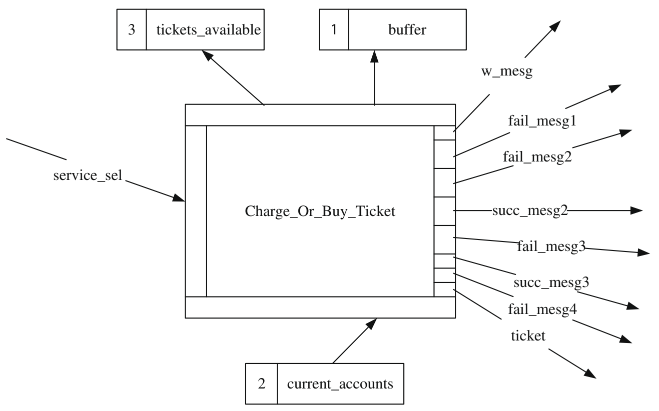


Fig. 7. The IC card system integrating all the functional scenarios.

likely to be injected during the construction of CDFDs due to human mistakes, which may violate the consistency and validity of the specification being written. Although completely eliminating all errors from the specification is important for quality assurance, we must also consider time constraint and customers and project managers' patience in practice. To strike a good balance, we adopt a "light weight" formal verification and validation technique known as *rigorous review* [13]. Using this technique to review the consistency and validity of a CDFD and its corresponding module specification, the reviewer first needs to derive all the consistency properties, such as the feasibility of a process specification, type-invariant-conformance consistency (i.e., type invariants are sustained by processes), integration consistency (i.e., input data flows of a process must satisfy its precondition), and compositional consistency (i.e., the composed high level process must be refined by the low level CDFDs from which it is formed by composition.). Not all of these properties can be easily checked by formal verification (e.g., feasibility), and even this is possible, the verification process would be tedious and time consuming. In our review approach, we emphasize the importance of human reviewer's involvement and judgement in finding and removing errors. Together with the error detection and prevention techniques supported by the tool described in the next section, rigorous review can help to find most obvious errors in formal specifications. The important point in our integrated method is that such a review activity is embedded into the scenario composition process in building a complete specification.

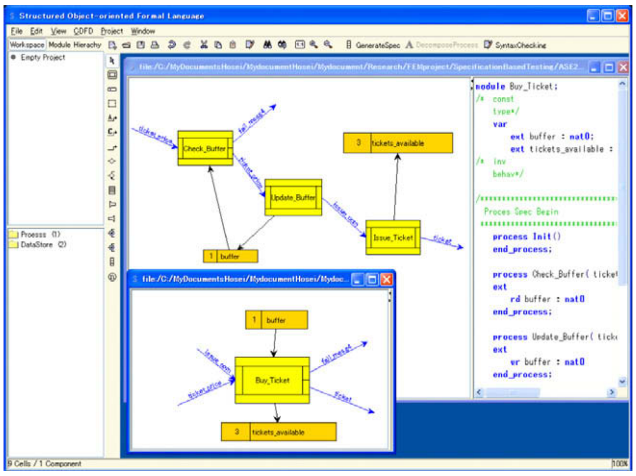
3. A supporting tool

Our tool supports both the top-down method and the bottom-up scenario-based method for construction of specifications in the SOFL specification language. Using SOFL, we can express the hierarchical structure of informal operations like in Fig. 1. We use a top-level process as an abstraction of the whole system under specification. The process is given a name reflecting the overall meaning of the system, but a relatively detailed description of the system function is given in the "comment" part of the process. The comment part of a process is generally used to interpret the formal specification of the process, so it is an appropriate place to informally describe the function of the process. At this level, all potential data flows and stores are not drawn in order to keep the description of the process abstract.

The top-level process is then decomposed into a set of low level processes, each of which has mere a name but no data flows and stores. If one of the processes needs to be decomposed further, it is again decomposed into a set of low level processes. Such a decomposition process continues until no process needs to be decomposed.

On the basis of the hierarchy of processes, a bottom-up scenario-based method is used to define all the bottom-level processes, compose them into scenarios (represented by CDFDs), and abstract existing scenarios into high level processes and compose them into more complex scenarios. This process continues until an appropriate top-level process is finally constructed.

To enhance the efficiency and quality of specification construction, our tool supports the incorporation of verification and validation activities within the specification construction process based upon *error detection and prevention*. Error detection and prevention are implemented in three activities: constructing a CDFD, writing a module specification, and abstracting a scenario into a high level process. Fig. 8 shows a snapshot of the GUI of the SOFL tool illustrating a situation of abstracting a scenario in Fig. 2 into a high level process. Since the introduction of the tool is beyond the scope of this paper, readers can refer to our paper [21] for details.



```
module Buy_Ticket;
/* const
   type*/
   var
      ext buffer : nat0;
      ext tickets_available :
/* inv
   behav*/

/*********************************
 Proces Spec Begin
 *********************************
   process Init()
   end_process;

   process Check_Buffer( ticke
   ext
      rd buffer : nat0
   end_process;

   process Update_Buffer( ticke
   ext
      wr buffer : nat0
   end_process;
```

Fig. 8. A snapshot of the SOFL tool illustrating the process of abstracting a scenario into a high level process.

4. Related work

Both top-down and scenario-based analysis and specification techniques have been extensively researched, but not much work on the combination of the two complementary approaches has been reported in the literature. For the sake of space, our overview in this section is limited only to the related work that have mostly influenced our research in this paper.

KAOS is a goal-directed requirements acquisition technique proposed by Dardenne [22]. The process of a requirements analysis usually starts with the description of the top-level goal, describing the overall objective to achieve, and then decomposes the goal into necessary sub-goals. Defining and structuring goals lead to assignments of responsibilities and designs of actions and artifacts, including concerned objects and actions, agents and their capabilities, and constraints [23]. The most challenging issues facing the KAOS method is how to identify goals and how to determine when they are completely specified. Anton describes several techniques to address those issues [24], such as scenario analysis, identification of goal obstacles and constraints, and goal operationalization, but the techniques are discussed at informal level and therefore difficult to be supported by software tools in depth. Unlike the goal-driven approach, scenario-based analysis, on the other hand, provides a way to capture requirements and the corresponding specifications by focusing on the analysis of scenarios [3]. The use case analysis approach adopted in UML is an example of scenario-based analysis [25], but its main purpose is to help identify objects and their relations necessary for a system under construction. One challenge to the scenario-based analysis is that scenarios are extremely labor-intensive to capture and document [26,27]. To address this challenge, Sutcliffe et al. developed a method and software assistant tool for scenario-based requirements engineering that integrates with use case approaches to object-oriented development [28]. Scenarios are used to represent paths of possible behavior through a use case, and these are investigated to elaborate requirements. The method commences by acquisition and modeling of a use case. The use case is then compared with a library of abstract models that represent different application classes. Watahiki and Saeki argues that goal-oriented analysis and use case analysis are complementary in the sense that goal-oriented methods are suitable for eliciting constraints to a system, while use case analysis methods elicit concrete system behaviors [29]. They propose to integrate the two methods into a new method for stronger capability in requirements elicitation. The characteristic of the new method is that constraints on the system are refined in goal-oriented manner, while system behaviors are described with hierarchical use cases; decompositions of goals and use cases progress hand in hand. Sun and Dong proposed an combination of interaction-based and state-based modeling approaches to provide a basis for automatic design synthesis [30]. The two modeling approaches are complementary in the sense that the former focuses on global interactions between system components while the latter concentrates on the internal states of individual components. The combined approach handles systems with intensive interactive behaviors as well as complex state structures.

Compared to the above work, our integrated method shares with Watahiki and Saeki's view that top-down decompositional methods and scenario-based methods are complementary, but differs in the way of integrating the two methods. The distinct features in our method are mainly twofold. One is that we use top-down decompositional method for capturing and structuring functional operations, data resources, and constraints only at an abstract level, while a precise specification of the allocation of data resources to operations, the functions of operations, the constraints on data and operations, and the communications among operations are constructed using a bottom-up scenario-based method. The advantage of this approach lies in its capability to avoid unnecessary modifications of high level operations and inconsistency between existing scenarios and their compositions. Another feature is that our method makes good use of the CDFD in the SOFL specification language that allows to express compositions of exclusive scenarios into a single high level process. The resulting specification presents a hierarchy of CDFDs that can be either implemented in a structured programming language like C or an object-oriented programming language like Java.

5. Conclusion and future work

Top-down decompositional methods and bottom-up scenario-based methods are complementary in requirements analysis and specification. In this paper, we describe how the two existing methods can be integrated together to benefit requirements elicitation and specification construction. Our integrated method emphasizes the simplicity in a top-down decompositional process by advocating the use of simple English expressions in a hierarchical structure. It also emphasizes the importance of capturing precise and complete scenarios in an intuitive formal specification language such as SOFL, including their behaviors, associated data, constraints, and communications between processes involved, in a bottom-up compositional manner. Due to the well-defined formal syntax and semantics of the SOFL specification language, we have built a software support tool for our method with several error detection and prevention capabilities. Both the method and the tool have been effectively applied in our recent project for developing an IC card system (including the three sub-systems: ATM system, railway ticket system, and shopping system), and we take part of the system as an example to present how our integrated method works in practice.

To enhance the application capability of our method, we will continue to work on more effective techniques for top-down analysis in future, including those for identifying necessary requirements from a domain description documentation, for effectively communicating with the client, and for accurately measuring the coverage of the desired functions in the obtained operations. We are also interested in evolving our tool to support the new techniques, such as model checking [31,32], and in carrying out more empirical studies using our method and tool.

Acknowledgements

We would like to thank all the research students, in particular Kenji Miyamoto, Xiang Xue, and Bin Gong, for their great contributions to the implementation of the software support tool for SOFL. We also thank all the senior students in our research group for their hardworking in applying the integrated method to build the IC card system.

References

[1] S. Liu, Integrating top-down and Secnario-based methods for constructing software specifications, in: Proceedings of 8th International Conference on Quality Software (QSIC 2008), IEEE CS Press, Oxford, UK, August 12–13, 2008.
[2] E. Yourdon, Modern Structured Analysis, Prentice Hall, 1989.
[3] K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, Scenarios in system development: current practice, Software 15 (2) (1998) 34–45.
[4] Grady Booch, Object-Oriented Analysis and Design with Applications, Addison Wesley Longman, Inc., 1994.
[5] Meilir Page-Jones, Fundamentals of Object-Oriented Design in UML, Dorset House Publishing, 2000.
[6] T.H. Tse, J.A. Goguen, Functional Object-Oriented Design, Technical Report 35, International Conference and Research Center for Computer Science, Wadern, Germany, 1992.
[7] Haifeng Qian, Eduardo B. Fernandez, Jie Wu, A combined functional and object-oriented approach to software design, in: Proceedings of First IEEE International Conference on Engineering of Complex Computing Systems, Fort. Lauderdale, Florida, IEEE Computer Society Press, November 6–10, 1995, pp. 167–178.
[8] J. Hunt, Guide to the Unified Process Featuring, UML Java and Design Patterns, Springer-Verlag, 2003.
[9] S. Liu, Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, 2004, ISBN 3-540-20602-7.
[10] J. Dawes, The VDM-SL Reference Guide, Pitman, 1991.
[11] The VDM-SL Tool Group, Users Manual for the IFAD VDM-SL Tools, The Institute of Applied Computer Science, February 1994.
[12] S. Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, M. Ohba, SOFL: a formal engineering methodology for industrial applications, IEEE Transactions on Software Engineering 24 (1) (1998) 337–344 (January 1998, Special Issue on Formal Methods).
[13] S. Liu, An automated rigorous review method for verifying and validating formal specifications, in: Farn Wang (Ed.), Proceedings of Second International Symposium on Automated Technology for Verification and Analysis, Taipei, Taiwan, ROC, Oct. 31–Nov. 3, LNCS 3299, Springer-Verlag, 2004, pp. 15–19.
[14] C.B. Jones, Systematic Software Development Using VDM, second ed., Prentice Hall, 1990.
[15] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 237–274.
[16] S. Liu, M. Asuka, K. Komaya, Y. Nakamura, Applying SOFL to specify a railway crossing controller for industry, in: Proceedings of 1998 IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98), IEEE Computer Society Press, Boca Raton, Florida USA, October 20–23, 1998.
[17] H. Gomaa, S. Liu, M. Shin, Integration of domain modeling method for families of systems with the SOFL formal specification language, in: Proceedings of Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2000), IEEE Computer Society Press, Tokyo, Japan, Sept. 11–14, 2000.
[18] D. Djuric, D. Gasevic, V. Devedzic, Ontology modeling and MDA, Journal of Object Technology 4 (1) (2005) 109–128.
[19] S.A. DeLoach, T.C. Hartrum, A theory-based representation for object-oriented domain models, IEEE Transactions on Software Engineering 26 (6) (2000).
[20] H. Gomaa, L. Kerschberg, V. Sugumaran, Knowledge-based approach to domain modeling: application to NASA's payload operations control centers, Telematics and Informatics 9 (3–4) (1992) 3.
[21] S. Liu, X. Xue, Automated software specification and design using the SOFL formal engineering method, in: Proceedings of 2009 World Congress on Software Engineering (WCSE 2009), Xiamen, China, IEEE CS Press, May 19–21.
[22] A. Dardenne, Goal-directed requirements acquisition, Science of Computer Programming 20 (1–2) (1993) 3–50.
[23] T. Delor, R. Darimont, A. Rifaut, Software quality starts with the modelling of goal-oriented requirements, in: Proceedings of the 16th International Conference on Software and Systems Engineering and their Applications, December 2–4, 2003, p. CETIC (B).
[24] A.I. Anton, Goal-based requirements analysis, in: Proceedings of the Second IEEE International Conference on Requirements Engineering (ICRE '96), Colorado Springs, USA, April 1996, pp. 136–144.
[25] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.
[26] P.A. Gough, F.T. Fodemski, S.A. Higgins, S.J. Ray, Scenarios – an industrial case study and hypermedia enhancements, in: Proceedings Second IEEE Symposium on Requirements Engineering, IEEE CS Press, 1995, pp. 10–17.

[27] T. Royer, Using scenario-based designs to review user interface changes and enhancements, in: Proceedings of the DIS95: Designing Interactive Systems, 1995, pp. 237–246.
[28] A.G. Sutcliffe, N.A.M. Maiden, S. Minocha, D. Manuel, Supporting scenario-based requirements engineering, IEEE Transactions on Software Engineering 24 (12) (1998) 072–1088.
[29] K. Watahiki, M. Saeki, Combining goal-oriented analysis and use case analysis, IEICE Transactions on Information and Systems E87-D (4) (2004) 822–830.
[30] J. Sun, J.S. Dong, Design synthesis from interaction and state-based specifications, IEEE Transactions On Software Engineering 32 (6) (2006) 349–364.
[31] J. Sun, Y. Liu, J.S. Dong, An analyzer for extended compositional processes, in: Proceedings of International Conference on Software Engineering (ICSE'08), ACM Press, Leipzig, Germany, 2008, pp. 919–920.
[32] Z. Duan, C. Tian, L. Zhang, A decision procedure for propositional projection temporal logic with infinite models, Acta Informatica (45) (2008) 43–78.