Extension on Transactional Remote Services in SOFL

Yisheng Wang and Haopeng Chen

School of Software, Shanghai Jiao Tong University
Shanghai, 200240, China
`easonyq@hotmail.com, chen-hp@sjtu.edu.cn`

Abstract. Software quality always attracts considerable attentions of people. Software running without any mistakes is always a dream of all developers. Besides traditional testing method using in practice such as path coverage, selection coverage, etc, people try to use some more formal and reliably method to ensure the quality. SOFL, stands for Structured Object-oriented Formal Language, is a kind of formal language which can be used to describe, validate and verify core business flow of software. As software developing model keeps changing for years, we need to make some extensions to SOFL. In this paper, we have performed extension on transactional remote services designed for SOFL. Our extension can mainly be divided into two parts: remote services and transactions. By introducing these, SOFL is able to keep pace with the changing software developing model, thus ensuring software quality in a more mathematical and different way comparing with traditional testing.

1 Introduction

A mature and practical commercial software product always needs a relatively long period of time and cooperation of many people including managers, designers, developers and testers. Nevertheless, software quality still cannot be perfectly guaranteed. Bugs and maintaining costs are bottlenecks of software industry to some extent[1].Theoretically, each software product contains potential problems and whether it would crash in the next second remains unknown[2]. People have already found a lot of ways to improve software quality such as standard developing processes and software testing methods. Using these classic methods such as RUP developing process[3], UML[4], black-box testing and white-box testing can improve software quality, but still 100% correct is unable to be reached or proved.

Actually, there are some other researches focusing on formal methods in software developing. SOFL[5] is a kind of formal language which can be used to describe, validate and verify a business workflow in a software product. Usually we use SOFL Specification and CDFD (stands for Control and Data Flow Diagram) to model a workflow. These can still be divided into some basic elements such as Module, Process, Dataflow, Datastore, etc. In general CDFD describes relationships of these elements such as dataflow connecting two neighboring processes. It should be noted that there also exists hierarchical relationship between two CDFD Diagrams. SOFL Specification carries out the detailed implementation of a module, including its pre and post condition, data store, input and output variables. It plays a complementary role together with CDFD.

After modeling a workflow using SOFL, we are able to find whether there are some potential mistakes in the model by validating and verifying which will be described afterward. The correctness of a workflow validated and verified by SOFL can be ensured by formal methods[6][7].

The programming model of software product always keeps changing. Invoking remote services helps developers to reuse codes as well as avoid duplicating time-consuming developing job, which is not supported or introduced in SOFL. Moreover, transaction is an important part in software product, especially for web-based applications with relatively complicated business logic. If SOFL can keep pace with this changing trend, its influential range will spread, thus promote fames of software formalization. This is also what our extensions aim to.

The rest of the paper is organized as follows: Section 2 lists some motivations of our work. Section 3 introduces the detailed extensions we have performed, dealing with remote services and transactions individually. Section 4 summarizes a practical case modeling with SOFL along with our extensions. Section 5 introduces related work by others on SOFL recently. Section 6 summarizes the main contribution of this paper and comments on further research.

2 Motivation

As the day passed, software developing model has changed a lot comparing with several tens of years ago. What software developers focus on now is to reuse as much existing remote services as possible rather than writing and testing same and duplicate codes. This is also the sharing notion of both component-based software engineering[8] and SOA[9]. The difference between component and service mainly lies in the location of the reusing piece of codes. Components can be withdrawn and run locally while services can not. In general case, we invoke remote services by providing input parameters and waiting for output results through network connections. Its running process is totally different comparing with local processes. Moreover, as Cloud Computing gaining more and more attentions of people, this notion has been inherited and carried forward by invoking remote services in Cloud[10].

In SOFL, there are several concepts dealing with 'external' elements. 'External Process' means a virtual process situated before the first process or after the last process. Generally, external processes are used to describe the end user or third-party system in a workflow. Users providing input data or operations and third-party system which receives output data from the workflow can be called external processes towards it. Another concept reads 'external stores' which is a kind of data store. It is used to describe 'external devices or files, such as displays, files on disks, printers, keyboards'[5]. All these seems to be not very related with the scenario of reusing. Unfortunately, support on these aspects of SOFL is quite weak. As SOA and Cloud Computing boosts in a fantastic speed, SOFL should also keep its pace with these popular notions.

Another key point towards web-based application, especially dealing with online payment or finance elements is transaction. A well-informed instance is saving, withdrawing or transferring money from a bank, which we can simply find in our database textbook. More importantly, as the introducing of invoking remote services, the possibility of invoking exceptions increased because of the unreliability of network connections. According to Probability Theory , if more than one remote services or data stores are involved in a workflow, the probability of mistakes will boosts in an exponential speed. It can be believed that transactions are required especially for workflow containing remote services or data stores. We can also inferred from the scenarios that long transactions would play a dominate role because of network delay and business logic. Assuring its correctness is also important towards a practical application. So it is reasonable for us to make extension to SOFL and enable it to deal with transactions, including long transactions.

It is clear that the requirement of extension to SOFL about remote services and transactions is reasonable and practical. We should add remote services, remote data stores and transactions into both SOFL Specification and CDFD based on the existing rules and grammars of SOFL. The detailed information of our extensions is shown in the next section.

3 Approach

According to what we have carried out in previous sections, our extensions on transactional remote services to SOFL can mainly be divided into remote elements and transactions. So we will give out our detailed ideas about these individually in the following.

3.1 Remote Elements

We have discussed the changes of programming model recently. These changes can mainly be concluded to distributed architecture (including distributed programming model such as Hadoop or distributed data store such as HDFS), using or revealing web service APIs, different database structure such as no-SQL database, etc.[11] Meanwhile, the ultimate goal of SOFL is to model and review core workflow of a software product. Thus extendibility of SOFL according to the changes made by software world are essential, otherwise it might be eliminated because of its fogyism.

SOFL is such a formal language that it does not care how much computing resources or its distributing and connecting situation a workflow actually used. This means information like whether distributed architectures are introduced, what kind of database is used cannot be revealed from SOFL. In other words, these information is transparent and inconsequential for modeling and reviewing because SOFL focuses on business logic more. But remote elements is different. Invoking remote services is actually a part of workflow, or said business process. Their differences over local services mainly lie on the unreliability of network connection and the existence of network delay. The difference between using remote data store and local ones is similar. To conclude, we do not care much about the detailed implementation of services or data stores, but care much about what we used is remote ones or local ones.

In the following chapters, we will discussed remote services and remote data stores. Both their textual and visual appearance will be displayed.

Remote Process. Service is the most basic unit in SOA. Services are well-defined business functionalities that are built as software components (discrete pieces of code and/or data structures) that can be reused for different purposes.[9] Service providers can be anyone who wants to be. They need to publish its interface and access information to the service registry. So such services which are invoked by a workflow and do not run on local node is called 'Remote Services'. By invoking remote services, a workflow need to provide input parameters and receive output results as the access information mentioned through network connection.

In SOFL, local function call is described by using 'process'. A process has its name, input parameters, output parameters, pre-conditions and post-conditions. If it uses data stores, a data flow described by a straight line with an arrow will be added in CDFD. Here pre-condition means the rule that input parameters must obey, and post-condition means the calculating process of results, thus is corresponds to function body.

In most cases, a remote service also have such five elements and their information is published in registry node. We want to point out that function body of remote service is transparent to service resumer in most cases. So the post-condition of remote service should be filled by user according to the describing and expect output information in service registry. The SOFL Specification we designed for remote services (or said remote process in SOFL's definition) is shown in snippets 1.

Algorithm 1. SOFL Specification of Remote Process

```
remote process Sample (x: $T_{i\_1}, y : T_{i\_2})z : T_{o\_1}, w : T_{o\_2}$
pre P(x, y)
post Q(x, y, z, w)
end_process;
```

We can find that the only difference towards normal process in SOFL is the keyword **remote**. It does not appear in original SOFL grammar, thus conflict and ambiguity will not be caused by adding this. It has been mentioned above, post-condition of remote process should be given out by user when modeling workflow according to the using scenario and intention of that service. More specific example about how to fill the post-condition of remote process can be seen in the next section.

After giving out the SOFL Specification extension, we continue to perform the extension on CDFD. Its difference towards original process in SOFL is only the letter 'R' in the right top side of the diagram. It can be seen in figure 1.

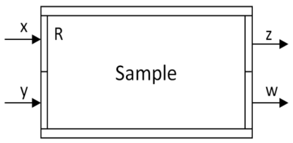

Fig. 1. CDFD of Remote Process

Besides difference in displaying symbol in CDFD and grammar, there are also some extra constraints of remote process. They are listed as follows.

1. There is an extra and common pre-condition of all remote processes, which is that the remote service must be accessible through network connection. Otherwise it can be regarded as a violation of pre-condition of process, thus leading to failure of the whole workflow.

2. Remote process in CDFD can not be decomposed into child-level further because the principle that detailed implementation of remote service is transparent to service consumer.

3. If this remote process has used data stores which does not belong to the workflow, it should not be displayed in CDFD. But if the data store belongs to the workflow, it still needs to be displayed. For instance, if a remote process needs to modify data saved in local database which is abstracted as a data store in SOFL, it should be displayed in CDFD with an arrow line linking from remote process to it.

Remote Data Store. Remote data store means data stored on remote node such as remote database or storage service. A remote data store is an external data store because its storing location is external towards the workflow. So we need to add mark # when they are declared in a module. To be distinguish from normal data stores, we add keywords **remote** before variables. Thus, the SOFL Specification of a process using remote data stores is shown in snippets 2.

The overall specification grammar is quite similar to normal data store, but different at the keyword **remote**, which has already been defined and applied in remote processes. According to the definition of external stores in SOFL, they are global variables. Therefore, remote data stores are also global variables.

We have also designed the CDFD of process declared in snippets 2. It is shown in figure 2.

Algorithm 2. SOFL Specification of Processes Using Remote Data Store

```
process Sample2 (x: $T_{i\_1}, y : T_{i\_2}) z : T_{o\_1}, w : T_{o\_2}$
ext
    remote rd #v_1 : Te_1
    remote wr #v_2 : Te_2
pre P(x, y, v_1, v_2)
post Q(x, y, z, w, ~v_2, v_1, v_2)
end_process;
```

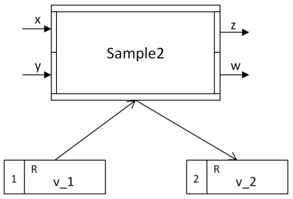

Fig. 2. CDFD of Processes Using Remote Data Store

Similar to CDFD of remote process shown in figure 1, the only different between remote data store and normal data store lies in the **remote** keyword in specification and the 'R' mark in CDFD.

3.2 Transactions

A transaction by definition must be ACID which stands for atomic, consistent, isolated and durable[12]. Usually it provides an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever. In software products especially web-based applications, transactions are widely used in order to ensure the correct running of the whole system and avoid data inconsistency or business logic chaos. The simplest and most popular example is transferring money between banks. It can be said that most software products might not run correctly without transactions.

SOFL has not introduced transactions according to its original definition. In SOFL, a workflow is running by steps from one process to its successor without operations of opposite direction such as rollback. Thus it is difficult for us to use SOFL to model a workflow with transactions. We have performed our definition of transactions in SOFL in the following paragraphs.

The most basic unit of SOFL-modeled workflow is process. So it is reasonable to introduce transaction based on processes, which means transactions in SOFL are composed with several processes. According to the feature in SOFL that processes can be further decomposed and CDFD is hierarchical, we also want to carry out some constraints on transactions as follows.

1. All processes in a transaction must be neighbored in CDFD. This is also a conventional constraint of transaction.

2. All processes in a transaction must be in a same CDFD. For example, the top level of a workflow is composed with three processes named ‘A’, ‘B’, ‘C’ individually. ‘A’ can be further divided into ‘Aa’ and ‘Ab’. ‘B’ is composed with ‘Ba’, ‘Bb’ and ‘Bc’. According to this rule, ‘Aa’ and ‘Ab’ can be in a same transaction, but ‘Ab’ and ‘B’ can not composed of a transaction. This is mainly because the inner structure of ‘A’ is transparent to ‘B’. And usually, a process is relatively independent to other process. The fact that a process has several child-level processes is mainly because it has divided its function into several parts. Just focus on the previous instance, the function of ‘A’ and ‘B’ is different and independent. It can not be very common that ‘Ab’ and ‘B’ need to be in a same transaction. Similarly, ‘Ab’ and ‘Ba’ can not be in an transaction either.

If a transaction need a lot of time to finish, we call it long transaction. There is not a very exact and strict boundary time to distinguish long transactions against normal ones. Normal transaction is implemented by executing it in memory and flush data to disk when committed. But such implementation can not be applied directly to long transactions because the performance influence of write-lock and data size. In order to ensure the ACID of long transactions, a series of operation called ‘compensating transaction’ is introduced and invoked when rollback operation is needed.

In practice, the implementations of compensating transaction is roughly divided into two. The first is writing the inverse operation by user and execute it as rollback. The other is to take snapshots before transaction starts and set all variables to that value when rollback is needed. Its detailed implementation contains the following steps. Before the first process of a transaction is started, a snapshot of the whole system is taken and saved. This snapshot is actually a set of values of current variables of the whole system. Then the transaction started by executing processes it contains in a certain order. When it need to rollback for some reasons, the snapshot is used. We define a set of operations as compensating transaction which sets the value of variable to the original state. It is determined by the snapshot taken before the starting of the transaction. After these set operations has finished, the transaction has rollback to its original state, thus guarantee the “all-or-nothing” feature.

Our extension supports both methods. If user has not written his compensating transaction, taking snapshot is used by default. The selection of rollback strategy is made by user.

There are hierarchical relations among transactions, which means nested transactions are allowed. A remarkable difference between nested transactions against normal ones mainly lies on the retry count. We assume transaction A contains transaction B and we set retry count to n. This means A is failed if and only if B has rollback for n times. In other words, failure of B only once would not cause failure of A if n is greater than 1. Moreover, in general case, nested transactions are long transactions.

We have discussed the definition and some constraints of transactions in SOFL along with long transactions and the implementation of their compensating transactions. We are ready to give out our extension on grammar in SOFL Specification. Because multiple processes can be involved in a transactions, so we need to declare a transaction first, and then refer it in the process definition block if it belongs to this transaction. In addition, because all processes of a transaction must be in a same CDFD, so the declaration of transaction can be placed in the module which is composed with these processes. A sample module with transaction is defined in snippet 3.

In snippet 3, a module named '*Sample_Module*' is defined and indicates '*Sample_Parent_Module*' as its parent module. The general procedure of declaring a module is defined in the following order: constants, types, variables, invariants, behavior and processes. Our extension is to add transactions sections between invariants and behavior and start with the keyword **transaction**. Transaction has the only attribute: name. It is also the only mark when using to distinguish from others, thus it must be unique. In order to show hierarchical structure, '*Trans_2*' behaves as the child transaction of '*Trans_1*' with retry count equaling 3.

There is also an extra transaction definition section in process definition sections just after the name and parameter section. It indicates which transaction it belongs to. Literally, '*Trans_1*' only contains one process, but actually it acts as the parent transaction of '*Trans_2*', thus it contains all these three processes by analyzing the hierarchical relationships. A process can belong to multiple transactions. The transaction section in process definition sections is able to declare all transactions it belongs to by using semicolons as their separators, but ancestor transactions need not to be declared.

Process named '*Sample_Compensating_Process*' is defined as the compensating transaction of '*Trans_2*' by using keyword **compensating transaction**. Note that each process can only declare one transaction name as its compensating transaction. There are not any compensating transactions declaring for '*Trans_1*', thus the default taking snapshots method is used.

Transactions can also be displayed in CDFD using special symbols. We define a dashed box around processes as all of these processes belong to a same transaction. Its name is marked in the dashed box to be distinguished against others. Just take the snippet 3 as an instance, its CDFD is demonstrated in figure 3.

Compensating transaction '*Sample_Compensating_Process*' is not displayed in CDFD because it is not the basic flow of this workflow. Dashed box is shown only in CDFD of this level. It would not display in other level, such as the CDFD for module '*Sample_Parent_Process*' in the example.

4 Evaluation

We have performed our detailed extension method on SOFL about transactional remote services both in CDFD and SOFL Specification in the previous section. In this section, we try to demonstrate a case modeling with extended SOFL to

Algorithm 3. SOFL Specification of Module with Transaction

```
module Sample_Module / Sample_Parent_Module
transaction
    Trans_1;
    Trans_2 / Trans_1 retry 3;
behav CDFD_Sample_Module
process Sample_Process_1 (x: $T_{i\_1}, y : T_{i\_2}) z : T_{o\_1}, w : T_{o\_2}$
transaction
    Trans_2;
ext
    rd v_1 : Te_1
    wr v_2 : Te_2
pre P(x, y, v_1, v_2)
post Q(x, y, z, w, ~v_2, v_1, v_2)
end_process;
process Sample_Process_2 (z: $T_{o\_1}, w : T_{o\_2}) v : T_{o\_3}$
transaction
    Trans_2;
ext rd v_2 : Te_2
pre P'(z, w, v_2)
post Q'(z, w, v)
end_process;
process Sample_Process_3 (v: $T_{o\_3}) u : T_{o\_4}$
transaction
    Trans_1;
ext
    wr v_3 : Te_3
pre P"(v, u, v_3)
post Q"(v, u, ~v_3, v_3)
end_process;
process Sample_Compensating_Process (v:
  $T_{o\_3}) x : T_{i\_1}, y : T_{i\_2}, z : T_{o\_1}, w : T_{o\_2}$
compensating transaction Trans_2;
ext
    wr v_2 : Te_2
pre P"(v, v_2)
post Q"(v, x, y, z, w, ~v_2, v_2)
end_process;
end_module;
```

show its using scenario and effect. By introducing this, a more clear and deep understanding of extended SOFL will be able to build.

Our case is to model a workflow of purchasing commodity. It is quite common in web-based applications. The approximate processes are listed as follows.

1. Login to our system. In order to keep pace with popular SSO technology (stands for Single Sign On), user need to send his user name and password to a

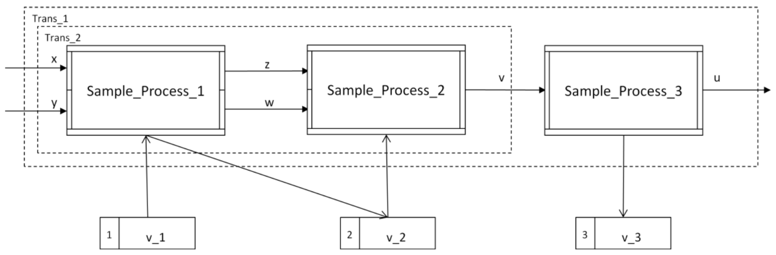

Fig. 3. CDFD of Module with Transaction

user authentication center. If authentication succeeds, UC provides our system with the user's ID along with some necessary user-related information. In general case, UC exposes a web service API to all its downstream system. Therefore a remote process of invoking that service is needed.

2. Online Payment. We have introduced a third-party online payment system to our workflow. Such systems includes PayPal or e-bank. It is also a remote process. After succeeding in payment, it returns flag indicating the successful message.

3. Increase Possessions. Add the number of the user's purchased commodity by 1 in user's profile. For data consistency, procedure No.2 and 3 must be "all-or-nothing", thus they must be in an transaction.

We use three processes to model this workflow, corresponding to three procedures mentioned above. Among these, login process and online-payment process are remote processes while the rest is local. Besides, we should introduce two local data stores saving user's authentication data and data about user's amount of possessions. The last key point is that the second and third process need to be included in a transaction to avoid situations such as failing in payment but succeed in adding possessions. The detailed information of these three processes are listed as follows.

1. Login. This is a remote process. It takes user's user name and password as input, and judge whether he is a valid user of our system. If true, it gives out detailed information of this user, otherwise an error message acts as its output. Dealing with this error message may need another process (maybe named '$Display_Message$'), but it is not the key point of our sample, thus it is omitted. '*Login*' also need to read information from data store named '*account_info*', which saves user's authentication information.

2. Online Payment. This is also a remote process. It takes user's information as input and returns a flag indicating whether the payment operation is successful along with his information for next process. Actually it needs a **wr** operation to user's data about his balance, but it should not be displayed in our system because it is transparent to our workflow.

3. Increase Possessions. This is a normal process. It takes the current user as input and gives out a flag showing whether it succeeds. It needs to search the user's possession information by user's information and overwrite it, thus a **wr** operation to data store named '*possession_info*'.

After analyzing requirements and making a rough design about the system, we try to give out its detailed SOFL Specification of this workflow. Just as what we have discussed above, this specification contains remote processes and transactions. The module is named '*Purchae*'. It is shown in snippet 4.

We have omitted the process '*Display_Message*' which deals with the error message '*Login*' gives out. Moreover, as an example, we have also simplify the workflow. For instance, we assume there is only one type of commodity. In practice, information about commodities and their prices, stocks can always be very large. To transplant it into real use, we may need to add a process dealing with what commodities the user want to buy and their total prices between '*Login*' and '*Online_Payment*'. Also, logics dealing mistakes are also omitted such as when user's balance is not enough to afford or commodity is out of stock.

In this example, remote process and transaction is involved. We need to note some key points.

1. Post-conditions of remote processes are added by user, thus we cannot find the detailed implementation of online payment in post-condition of '*Online_Payment*'. We focus on their input and output variables to ensure the reasonability and correctness of the whole workflow.

2. We have not defined the compensating transaction for '*Purchase_Trans*', thus taking snapshot is used by default.

3. Data store '*account_info*' belongs to our system, thus it appears in CDFD, But user's data about his balance is transparent to our workflow, so it should not appear in CDFD.

4. If we try to use account information from other systems, '*account_info*' can be remote data store. This requirement is also common if several systems try to work together and share their users. The CDFD of this workflow is shown is figure 4.

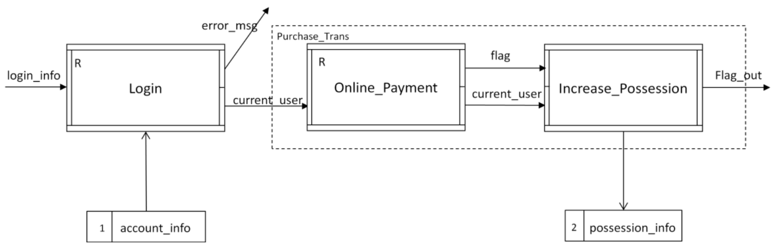

Fig. 4. CDFD of Purchase

Algorithm 4. SOFL Specification of Purchase

```
module Purchase;
type
  Login_Info = composed of
    user_name: string
    password: string
  end;
  Account_Info = composed of
    user_id: nat
    email: string
  end;
  LoginAccountFile = map Login_Info to Account_Info;
  PossessionAccountFile = map Account_Info to nat0;
var
  ext #account_info: LoginAccountFile;
  ext #possession_info: PossessionAccountFile;
inv
  forall[x: Account_Info] | not exists [y: Account_Info] | x.user_id = y.user_id;
transaction Purchase_Trans;
behav Purchase_CDFD;
remote process Login(login_info: Login_Info) current_user: Account_Info |
error_msg: string
ext rd account_info
post if login_info inset dom(account_info)
  then current_user = account_info(login_info)
  else error_msg = "Your password or user name is incorrect."
end_process;
remote process Online_Payment(current_user: Account_Info)current_user_out:
Account_Info, flag: bool
transaction Purchase_Trans;
post current_user_out = current_user
   and flag = true
end_process;
process Increase_Possession(current_user: Account_Info, flag: bool)flag_out:
bool
transaction Purchase_Trans;
ext wr possession_info
pre flag = true
post possession_info = override(~possession_info, map:current_user − >
~possession_info(current_user) + 1)
  and flag_out = true
end_process;
end_module;
```

5 Related Work

Workflow technology has attracted attentions of researches for a period of times. In workflow researching field, a representative language is BPEL[13]. Because of its ambiguities and lack of formal semantics, many researches have been performed to formalize BPEL by introducing process algebgra, Petri nets, automata, etc.

Process algebras is also used to form workflow. It contains ACP (stands for Algebra of Communicating Processes), CSP (stands for Communication Sequential Processes), CCS (Calculus of Communicating Systems), etc[14]. Researches on modeling business logic by process algebra is a popular topic these years. Salaun has presented a method for verifying business processes based on process algebras which mainly focus on their interactions[15]. The shortage of using process algebra to model business process mainly lies in its lack of support on dynamic process instantiation and correlation set. Process algebra also does not support dynamic structure alteration, which is important in business aspect.

Petri net is a strict and mathematical describing language. It can also used to verify workflow in an dynamic way. Using Petri net to model a business is an ideal and reliable method, especially after the introducing of high-order Petri net. Many researchers have tried their way to translate BPEL to Petri net[16][17][18]. But Petri net is based on graphical unit, thus its complexity boosts when modeling a large scaled and practical workflow. Moreover, data types in Petri net is also limited. Workflow involving rich data types is difficult to be described by Petri net.

According to the definition, automata is a public and base model of formal specification for systems which contains a set of stats, actions and transitions between states[19]. It is convenient to describe workflow because corresponding definitions can also be found in workflow. Diaz has researched a set of methods converting business processes writtin in BPEL-WSCDL to timed automata[20]. Fu has developed a tool which translating BPEL to guarded automata[21]. But limited by the feature of automata, it is also not suitable for describing large-scaled system because of the complicated structures and loss of accuracy.

Comparing to these relatively mature researches, SOFL starts later. The initial development of SOFL was made at the University of Manchester in the United Kingdom in 1989[22]. After that, SOFL had developed gradually with contribution of many researchers. Shaoying Liu has formalized its grammar and introduced to people in his papers and books.[5] A remarkable advantage of SOFL against other software formal language is its support of automatic verifying and validating. Several tools have also been developed to finish these[23][24]. Although its researches have not attract much people yet, its potential power and solid foundation is still convinced that it will keep going on.

6 Conclusion

As the rapid growth of software, both developers and users keeps changing. Because the number of software users boost, software need to face greater challenges not only on its functional requirements, but also its performance, availability, quality, etc. Being different from classic testing method, modeling software workflow using formal language is another attractive way to ensure software quality. Besides, it is more convincible to people because its mathematical base.

In this paper, we have performed several extensions on grammar of SOFL in order to make it able to keep pace with the developing software world. These extension points are listed as follows.

1. Extensions on remote elements. This can further be divided into remote processes and remote data stores. We use remote processes to model invoking remote services in workflow, which is popular in SOA. Remote data stores are used to model invoking remote storage services in workflow. It is similar to remote processes. Remote elements are different from normal elements in several points. Its grammar on SOFL Specification mainly highlights in keyword **remote**.

2. Extensions on transactions. Transactions are important to software product because of its “all-or-nothing” feature. After extension, we are able to declare transaction name in module declaring section and add transaction declaring to process definition section to indicate which transaction(s) it belongs to. Transactions can be nested. One process can be included by multiple transactions. Its compensating transaction can be written by user, or using taking snapshots by default. All these two extra declaring section starts with keyword **transaction**.

We have also given out an simple example dealing with purchasing commodities to show the using scenarios and methods of our extensions. The changes we made to SOFL is not very large, and it does not conflict with normal SOFL grammar either because we use new keywords. These extensions can also be illustrated in CDFD by introducing different mark to distinguish.

SOFL is a new comer comparing with some elder and mature formal language members such as automata, Petri net and process algebra. But its potential power can not be regardless. As more researches have been done in SOFL, it is believed that SOFL will attracts increasingly more people including both researchers and users, thus ensure software quality in a more convincible way.

References

1. Marciniak, J.J.: Encyclopedia of Software Engineering, 2nd edn. Wiley Publications (1994)
2. Cai, L., Yang, G.: Software Quality Assurance Testing and Evaluating. Tsing Hua University Publications (2007)
3. Aked, M.: RUP in brief. In: Risk Reduction with the RUP Phase Plan, pp. 1–10. IBM (November 2003)
4. Pressman, R.S.: Software Engineering, a Practitioner’s Approach. McGraw-Hill Science/Engineering/Math. (2009)
5. Liu, S.: Formal Engineering for Industrial Software Development. Springer (2008)
6. Liu, S.: A property-based approach to reviewing formal specifications for consistency. In: Proc. of 16th International Conference on Software Systems Engineering and Their Applications, pp. 1–6 (2003)

7. Liu, S.: An automated rigorous review method for verifying and validating formal specifications. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 15–19. Springer, Heidelberg (2004)
8. Sun, C., Zhang, X., Zheng, L.: The research of the component-based software engineering. In: Sixth International Conference on Information Technology: New Generations, ITNG 2009, pp. 1590–1591 (2009)
9. Bell, M.: Introduction to Service-Oriented Modeling. Wiley and Sons (2008)
10. Raicu, I., Lu, S., Foster, I., Zhao, Y.: Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, GCE, pp. 1–10 (August 2008)
11. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups: The new generation of web applications
12. Korth, H.F., Silberschatz, A.: Database System Concepts, 4th edn. McGraw-Hill Education (2006)
13. Morimoto, S.: A survey of formal verification for business process modeling. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part II. LNCS, vol. 5102, pp. 514–522. Springer, Heidelberg (2008)
14. Sipei, L., Jin, W., Lei, W., Park, S.: Description logic rule, matching process algebra based OWL-S modeling, and composition
15. Schaerf, M., Salaun, G., Bordeaux, L.: Describing and reasoning on web services using process algebra. In: Proceedings of the IEEE International Conference on Web Services, pp. 43–50 (2004)
16. Verbeek: Analyzing bpel processes using petri nets
17. Van der Aalst: Verification of workflow nets
18. Dumas, M., Van der Aalst, Verbeek, H.M.W.: An approach based on bpel and petri nets (extended version)
19. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley (2006)
20. Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V., Cuartero, F.: Automatic translation of WS-CDL choreographies to timed automata. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 230–242. Springer, Heidelberg (2005)
21. Fu, X., Bultan, T., Su, J.: Analysis of interacting bpel web services. In: Proc. of 13th International Conference on the World Wide Web, pp. 621–630 (2004)
22. Sun, Y., Liu, S.: Structured methodology+object-oriented methodology+formal methods: methodology of sofl
23. Miyamoto, K., Liu, S., Fukuzaki, T.: A gui and testing tool for sofl
24. Wang, Y., Zheng, Q., Chen, H.: Soflipse: Tool for automatic modelling and reviewing sofl workflows. International Journal of Computing Technology and Information Security 1, 88–98 (2011)