**Modelling and Verification of Compensating Transactions using the Spin Tool**

Kaiyu Wan, Hemangee K. Kapoor, Shirshendu Das, B. Raju, Tomas Krilavičius, and Ka Lok Man

Abstract—Complex transactions are part of the most commonly used systems. Substantial part of such transactions are business transactions. Usually, they coordinate complex interaction among multiple systems, so called Long Running Transactions (LRT). Well known roll-back mechanism does not suffice to handle faults in LRTs, therefore compensation mechanisms are introduced. However, introduced structures are rather complex and hard to be understood and handled. Formal methods are well known tool for modelling, analysis and synthesis of complex systems. In this paper we introduce a work in progress, a technique that allows modelling LRTs using Compensating CSP, then translating them to Promela language and analysing using SPIN tool. We exemplify it using a Car Broker Service.

Index Terms—Long Running Transactions, Compensation Mechanisms, CSP, Promela, SPIN

I. INTRODUCTION

Business transactions typically involve coordination and interaction between multiple partners. These transactions involve hierarchies of activities and need to be orchestrated. Business transactions need to deal with faults that can arise in any stage of the transactions. In usual database transactions, a roll-back mechanism is used to handle faults in order to provide atomicity to a transaction. However, for transactions that require long periods of time to complete, also called Long Running Transactions (LRT), roll-back is not always possible. LRTs are usually interactive (communication with several agents). Handling faults where multiple partners are involved are both difficult and critical. Due to their interactive nature, it is not possible to checkpoint LRTs, e.g., a sent message cannot be unsent. In such cases, a separate mechanism is required to handle faults. A possible solution of the problem would be that the system designer can provide a mechanism to compensate the actions that cannot be undone automatically.

Compensation is defined as an action taken to recover from error in business transactions or cope with a change of plan [1]. Consider an example: a customer buys some items from an on-line store. The store debits the customer’s account for the payment of the items. Later the store realizes that one or more items are not available at that time. So, to compensate the customer, the store can credit the already debited amount and at the same time apologize to the customer, or the store can take alternate actions, such as, arranging items from an alternative source or asking the customer whether they want a later delivery, etc. The scenario shows that the concept of compensation is more general than traditional database roll-back. Compensations are very important for handling failures in long running transactions. Compensations are installed for every committed activity in a long-running transaction. If one sub-transaction fails, then compensations of the committed sub-transactions in the sequence are executed in reverse order.

Web services technology provides a platform on which we can develop distributed services. The interoperability among these services is achieved by the standard protocols (WSDL [2], SOAP [3]) that provide the ways to describe services, to look for particular services and to access services. With the emergence of web services, business transactions are conducted using these services [4]. Web services provided by various organizations can be inter-connected to implement business collaborations, leading to composite web services. Business collaborations require interactions driven by explicit process models. Web services are distributed, independent processes which communicate with each other through the exchange of messages. The coordination between business processes is particularly crucial as it includes the logic that makes a set of different software components become a whole system. Hence it is not surprising that these coordination models and languages have been the subject of thorough formal study, with the goal of precisely describing their semantics, proving their properties and deriving the development of correct and effective implementations.

Formal techniques proved their usefulness in quite a few areas, e.g., automotive industry [5], electronics [6], [7], industrial devices control [8], medical devices control [9], [10], [11]. Process calculi are models or languages for concurrent and distributed interactive systems. They have also been used for modelling interactions in latency insensitive SoC interconnects [12]. It has been advocated in [13], [14] that process algebras provide a complete and satisfactory assistance to the whole process of web services development. Being simple, abstract, and formally defined, process algebras make it easier to formally specify the message exchange between web services and to reason about the specified systems. Transactions and calculi have met in recent years both for formalizing protocols as well as adding transaction features to process calculi [15], [16], [17], [18].

Inspired by the growing interest in transaction processing using web services, in this paper we propose a technique for modelling business process in cCSP, then translating them into Promela language [19] and analysing the model in SPIN tool [20]. We introduce and informal translation from the cCSP to Promela, and leave formal description for the future
research. We exemplify the process with a business web service called Car Broker Web Service [21].

In section II we concisely overview existing results. Then we present cCSP (section III-A) and Promela/SPIN (section III-B). In section IV we discuss translation of cCSP model to Promela, then exemplify it in section V-A with the Car Broker Web Service. We finalize paper with conclusions (section VI).

II. RELATED WORK

Several research issues, both theoretical and practical, are raised by web services. Some of the issues are to specify web services by a formally defined expressive language, to compose them, and to ensure their correctness; formal methods provide an adequate support to address these issues [15].

Recently, many XML-based process modelling languages such as WSCI [22], BPML [23], WSFL [24], XLANG [25] have emerged that capture the logic of composite web services. These languages also provide primitives for the definition of business transactions.

Fu et al. [26] propose a method that uses the SPIN model-checking tool. The SPIN [20] tool takes PROMELA (Process or Protocol Meta Language)[19] as the input language and verifies its LTL (Linear Temporal Logic) [27] properties. Interactions of the peers (participating individual web services) of a composite web service are modeled as conversations and LTL is used for expressing the properties of these conversations.

Several proposals have been made in recent years to give a formal definition to compensable processes by using process calculi. These proposals can be roughly divided into two categories. In one category, suitable process algebras are designed from scratch in the spirit of orchestration languages, e.g., BPEL4WS. Some of them can be found in [28], [29], [30]. In another category, process calculi like the π-calculus [31], [32] and the join-calculus [33] are extended to describe the interaction patterns of the services where, each service declares the ways to be engaged in a larger process.

III. COMPENSATING CSP (cCSP) AND SPIN

A. Compensating CSP (cCSP)

Transaction processing and process algebra inspired the development of process algebra cCSP [34], [35], [30]. A subset of the original cCSP is considered in this paper, which includes most of the operators, as summarized in Table I. Similar to CSP, processes in cCSP can engage in atomic events and can be composed using sequential, choice and parallel composition operators. The processes are categorised into two types: (i) standard; and (ii) compensable: which have a separate set of actions to be executed upon failure of a transaction. Variables $P, Q, . . . \text{ are used for standard processes and } PP, QQ, . . \text{ are used for compensable processes.}$

Input on channel $a$ and output on channel $b$ can be described as $P!a$ and $Q?b$ respectively. The operators different from CSP are discussed below. In case of failures in long running transactions, we need support to raise interrupt and handle the interrupt. The THROW action is used to raise and interrupt and the $YIELD$ is used to handle it. For example, ($P; YIELD; Q$) is willing to yield to an interrupt in between the execution of $P$, and $Q$.

A compensable process is constructed using a pair $(P \div Q)$, where $P$ is the forward behaviour used to model normal execution, and $Q$ is the associated compensation designed to compensate actions executed in $P$. The sequential composition is defined in such a way, that actions done in $P$ are accumulated and will be executed in reverse order in case composition needs to be aborted and compensated. By enclosing a compensable process $PP$ inside a transaction block $[PP]$, we get a complete transaction, where the transaction block is also a standard process. Successful completion of $PP$ represents successful completion of the block. But, when the forward behavior of $PP$ throws an interrupt, the compensations are executed inside the block, and the interrupt is not observable from outside the block.

B. PROMELA and SPIN

PROMELA is the modelling language used in the Spin tool. It is used to model the required interaction behaviour and verify properties. The model consists of processes and channels. Processes are independent entities which need to be invoked using the run clause. Processes interact with each other over message channels and/or globally declared variables. Variables can be of types: bit, bool, byte, array etc. For details of all data types see the PROMELA manual in [19], [20].

The behavior of a process is defined by a proctype declaration and instantiated using the run command.

$$\text{proctype A() \{ byte state; state = 3; \}}$$

$$\text{init \{ run A(); \}}$$

The keyword atomic makes all the enclosed statements to be executed as one indivisible unit, non-interleaved with any other processes.

$$\text{atomic(statements;)}$$

Message channels are either input or output and carry data between processes. For example, the channel myout outputs value of variable $a$, whereas the channel myinput reads the incoming value in the variable $b$.

$$\text{chan myout = [2] of byte; \}$$

$$\text{chan myinput = [0] of byte; \}$$

$$\text{myout!a; \}$$

$$\text{myinput?b ; \}$$

The channel capacity can be given after its name. In the above example, channel myout has buffer capacity of 2 and the channel myinput having capacity zero is used for rendezvous communication. As cCSP uses rendezvous communication, we have used similar channels in our model.

For control flow, the if statement does a selection between a set of options. If multiple options are enabled, then any one is chosen at random. If none of the options are enabled, then the statement blocks until some statement becomes executable. In the following example, any one option will get executed, depending on the condition.

$$\text{if \{ \} unless \{ \} \}$$

The most important statement of PROMELA that we used in this paper is the unless statement

$$\text{( statements1 ) unless ( statements2 ) \}$$

It starts execution in statements1. Before every statement in statements1 is executed, it checks if the first
The first statement in statements2 can be executed. If yes, then the control transfers to statements2 else it continues execution of statements1. If statements1 terminates, statements2 is ignored.

IV. MODELLING COMPENSATION OF CcSP IN PROMELA

We propose a simple techniques for converting cCSP models to Promela and analysing them using SPIN. In this paper we just overview general principles of translation and leave formal treatment for the future research.

Translation of the simple cCSP process is rather straightforward and we do not discuss it here. Interesting part is translation of compensable processes. We divide them into two classes

1) single process in one transactional block,
2) multiple processes in one transactional block.

First, we will discuss a simpler case, and then we will build on it and show how more complex translation can be performed. Consider the following cCSP model of a compensable process:

\[
\begin{align*}
\text{Proc} & = [R] \\
R & = (((\text{channel1}?1;\text{SKIPP}) \\
\text{if (channel1}?0;\text{THROWW}) \div \text{Compensation Actions})
\end{align*}
\]

\(R\) receives input from \(\text{channel1}\) and if input is 0, it raises an interrupt to start Compensation Actions. The PROMELA implementation of \(R\) will be:

```c
byte doCompansate=0; // Global variable
PROC type R() {
    byte value; /\* local variable */
    {
        \/***********FORWARD SECTION***********/
        channel1?value; //Wait for input.
        if
            {value=1} -> printf("Success");
            {value=0} -> doCompansate=1; // interrupt
        fi
    }
    unless
        \/***********COMPENSATION SECTION***********/
        { doCompansate=1;
          ... //Compensation Actions. } }
```

The first statement in compensation section is a blocked statement and it has to be enabled first, to run Compensation Actions.

V. WEB SERVICE MODEL

Car Broker Model is used in [21], where the author used it as a case study for cCSP. Our Car Broker Model will be slightly different from his proposed model but most of the properties are common. The car broker web service negotiates car purchases for buyers and arranges loans for these. The car broker uses two separate web services: a Supplier to find a suitable quote for the requested car model and a Lender to arrange loans. Each web service can operate separately and can be used in other web services. In the following sections, we describe all three web services. We abstract several details from our description, e.g., how a supplier finds suitable quote for a car model, how a broker selects a quote from several available quotes, how a lender decides to select a loan request, the details a buyer request etc. The behavior of the web service is depicted in Figure 1.

A. Broker Web Service

We model the car broker using the process Broker. It provides online support to customers to negotiate car purchases and arranges loans for these. A buyer provides a need for a car model. The broker first uses its business partner Supplier to find the best possible quote for the requested model and then uses another business partner LoanStar to arrange a loan for the buyer for the selected quote. The buyer is also notified about the quote and the necessary arrangements for the loan. Both LoanStar and the Buyer can cause an interrupt to be invoked. A loan can be refused due to a failure in the loan assessment and a customer can reject the loan and quoted offer. In both cases, there is a need to run the compensation, where the car might have already been ordered, or the loan has already been offered.

The first step of the transaction is to receive an order from the buyer. The compensation mechanism in our model is little different than as mentioned in [21]. We will explain it after defining all the web service models. For now just consider that a compensation CompensateBroker, will be called if interrupt occurs. \(M\) is used to represent the finite set of car models ranged over by \(m\). After receiving the order (buyerOrder), it is then passed to the process ProcessOrder to perform the rest of the transaction. After receiving an order for a car from the Buyer, the Broker first requests the Supplier for available quotes (brokerRFQ) and then...
selects a quote from the received quotes (supQuote). We abstract away from the details of how decisions are made. The Broker then arranges a loan for the quoted car by requesting a loan from LoanStar. The amount of loan to be requested is decided from the selected quote and then passed to the process Loan. It requests loan from LoanStar and it can be either accepted or rejected. In the case where the loan is accepted, it is assumed that the loan provider starts its processing to arrange the loan. If the loan cannot be provided then an interrupt is thrown to cancel the actions that already took place. The buyer is also notified of the quote for the selected car (from SendQuote(c)). The Broker receives an acknowledgment (buyerAck) from the Buyer for either accepting or rejecting the quote. In case of rejection, an interruption is thrown to cancel the transaction and run the appropriate compensation. The processes SendQuote, Loan and SendOrder do not have any synchronization between them and they interfere with each other. An interrupt thrown from either the Buyer or the LoanStar can occur before or after ordering the car to the Supplier. In either case, the compensation mechanism takes care of it and the proper compensations will run.

B. Buyer Web Service

Buyer web service starts the whole process by expressing his need for a car to the Broker web service. Initially Buyer will send a request (buyerOrder) to the Broker and waits for a quote from the Broker. The Broker will collect the quote (the complete processing of Broker is already explained) and send it to the Buyer(brokerQuote). After receiving the Quote from Broker, Buyer can either accept it or reject it. In both situations Buyer must inform the broker about his decision (buyerAck). Buyer also has an associated compensation action called CompensateBuyer to cancel his order in case of exception occurred.

\[
\text{Buyer} \equiv ((\text{buyerOrder} \cdot m) : \text{M}; \text{ProcessOrder}(m)) \div \text{CompensateBuyer}
\]

\[
\text{CompensateBuyer} \equiv \text{SKIP}
\]

C. Lender Web Service

We assume a lender web service LoanStar, that offers loans to online customers. A customer submits a request for an amount to be loaned along with other required information. LoanStar first checks the loan amount and if the amount is 10,000 or more, then LoanStar asks its business partner Assessor to thoroughly assess the loan. After a detailed assessment of the loan, Assessor can either approve the loan or reject the loan. A full assessment is costly, so if the loan amount is less than 10,000, then we assume that the LoanStar will directly grant the loan.

\[
\text{LoanStar} \equiv (\text{brokerReqLoan} \cdot a) : \text{Amt}; \text{Process}(a)) \div \text{CancelLoan}
\]

\[
\text{Process}(a) \equiv \text{ChkAmt} : (\text{Below} \cdot a; \text{loanStarReply.accept}) \div (\text{Open} \cdot \text{Assessor}(a))
\]

\[
\text{Assessor}(a) \equiv \text{ChkRisk} : (\text{Loan} \cdot a; \text{loanStarReply.accept}) \div (\text{High} \cdot a; \text{loanStarReply.reject})
\]

\[
\text{CancelLoan} \equiv \text{SKIP}
\]

At the top level, the transaction is defined as a sequence of two processes. First, it receives a loan order (brokerReqLoan) from the Broker and then processes the loan. After the request is received from the Broker, the requested amount is passed to the process called Process to take the necessary steps before arranging the requested loan. It first checks the loan amount in order to determine the type of evaluation that it needs to perform before accepting the loan. We define a process ChkAmt which checks the loan amount in the order to determine whether the amount is over or below the given limit, which is in this case 10,000. Here, ChkAmt , Blow and Over abstract away the details of how the checking has been done. If the loan amount is less than 10,000, then Process will grant the loan by sending accept via loanStarReply. If the risk is high then control is passed to Assessor to perform a full assessment. On the other hand, if the amount is higher than or equal to 10,000, then Assessor will start its assessment immediately. After performing a full assessment and depending on the outcome, Assessor either accepts or rejects the requested loan. In the example, we abstract the details of the behavior of Assessor. It can be modeled as a separate web service or as a part of the lender web services. The associated compensation action here is called CompensateLoan and will be run if any exception occurred.

D. Supplier Web Service

It sends a set of quotes (supQuote) to the Broker after receiving a request for quotes (brokerRFQ) from Broker. We have not explained the detail quotes selection procedure here. We assumed that the Supplier will nondeterministically select one set of quotes (q) from the multi-set of quotes (FQ). After sending the quotes to the Broker it will wait for the Broker for ordering a car mentioned in the quotes. It has associated compensation action called CompensateSupplier to cancel the order in case of exception occurred.

\[
\text{Supplier} \equiv (\text{brokerRFQ} \cdot m) : \text{M}; \text{supQuote}? : \text{FQ}; \div \text{CompensateSupplier}
\]

\[
\text{CompensateSupplier} \equiv \text{SKIP}
\]

E. The whole car broker system

After defining each web services separately, now we are going to define the whole car broker web system. The cCSP description of this model is given below: We have enclosed
all the web service processes into a single transactional block (see section III-A). Hence all the processes can automatically yield the interrupt thrown by any of them to run the corresponding compensation actions.

F. Accomplishing Compensation

As mentioned in section V-A that the Broker will throw an exception if the Buyer rejects the quote or if the LoanStar rejects the loan. Since all the above cCSP processes (Broker, Buyer, Supplier and LoanStar) are running in parallel and comes under one single transactional block, the interrupt thrown by Broker can be automatically yielded by the other processes. So, whenever Broker throws some interrupt it will be yielded by all the processes (including Broker) and starts their individual associated compensation action. The purpose of compensation sub-process in each process is to revert back all of its updates. We are assuming that each web service maintain enough log records, so that they can revert their own updates in case of transaction failure. Since the compensation actions of each processes are all internal actions we are ignoring them by just assuming a SKIP.

G. PROMELA Model for Car Broker Web Service

Our complete system is enclosed within one transactional block as explained in section V-E. The detailed explanation about how to convert such type of cCSP model into PROMELA is already discussed in section IV. Success of a transaction is dependent on two conditions: (i) Buyer accepts the quote and (ii) LoanStar grants the loan. Otherwise transaction cannot succeed and compensation must run.

In our implementation we use two global variables sqSuccess and lSuccess to represent the status of buyer acceptance and loan grant respectively. Initially both assigned to 0. Two subprocess of Broker will decide the success of the transaction by setting the appropriate variables. If any condition is not satisfied, the corresponding compensation variable is set, in particular variables sqCompensate=1 and lCompensate=1. If any one of the compensate variables is 1, then the "unless" part of all the compensation processes will become enabled and hence starts compensation. So for each compositional process in our model, we define success as: (lSuccess==1 && sqSuccess==1) and run compensation if: (lCompensate==1 || sqCompensate==1)

Note that in our model every transaction will eventually either succeed or fail. Hence, either of the above state will eventually be true.

The sub-process Loan of the Broker process is shown below. It sets the appropriate variables depending on the decision of the loan agent LoanStar.

```
proctype Loan(int qL)
{
  brokerReqLoan=qL;
  loanStarReply?lresult;
  if
    :: lresult==1 -> // loan accepted
      lSuccess=1;
    :: lresult==0 -> // loan rejected
      lCompensate=1;
  fi;
  if // Wait for success indication
    :: (lSuccess==1 && sqSuccess==1)
    -> loanSuccessful=1;
  fi
}
```

unless {
  (lCompensate==1 || sqCompensate==1)
  // run compensation
  -> loanCancelled=1;
}

The SendQuote sub-process of the Broker process deals with the Buyer and sets the appropriate variables depending upon the Buyer’s decision.

```
proctype SendQuote(int qSQ)
{
  brokerQuote=qSQ;
  buyerAck?sqresult;
  if
    :: sqresult==1 -> // Buyer accepts quote
      sqSuccess=1;
    :: sqresult==0 -> // Buyer reject quote
      sqCompensate=1;
    fi;
    ...
  unless {
    (lCompensate==1 || sqCompensate==1)
    // Run compensation
    ...
  }
}
```

1) Verification: We verified safety and liveness properties for the model. The verification output for deadlock freedom is given below.

- Non-progress cycles: 27937663 states, matched
- Invalid end states: 3774707 states, stored
- Valid end states: 3791795 states, stored
- 11 atomic steps

The verification output for liveness is given below. Liveness is checked by showing the absence of acceptance cycles and also non-progress cycles. The model was free of acceptance cycles:

```
1) Verification: We verified safety and liveness properties for the model. The verification output for deadlock freedom is given below.

2) If loan is rejected by LoanStar then all processes are compensated.
3) If loan is rejected by LoanStar then all processes are compensated.
```

The list of LTL properties satisfied by the model is given below.

1) If quote is rejected by the buyer then all processes are compensated.
2) If loan is rejected by LoanStar then all processes are compensated.

The verification output for liveness is given below. Liveness is checked by showing the absence of acceptance cycles and also non-progress cycles. The model was free of acceptance cycles:

```
1) Verification: We verified safety and liveness properties for the model. The verification output for deadlock freedom is given below.

2) If loan is rejected by LoanStar then all processes are compensated.
3) If loan is rejected by LoanStar then all processes are compensated.
```

The list of LTL properties satisfied by the model is given below.

1) If quote is rejected by the buyer then all processes are compensated.
2) If loan is rejected by LoanStar then all processes are compensated.
ltl1( [] (lCompensate==1) -> (supplierCancelled==1) && (brokerCancelled==1) && (sendOrderCancelled==1) && (sendQuoteCancelled==1) && (loanCancelled==1) && (loanStarCancelled==1) && (assessorCancelled==1) && (processCancelled==1) ) )

3) If quote is accepted by buyer and the loan is sanctioned, then the processes are successful.

4) If no compensation request comes due to the buyer rejecting the quote and for the loan amount if there is low risk involved or the loan amount is less, then the loan is sanctioned.

ltl1( [] ((lowRisk) && (lessLoanAmount) && sqsCompensate==0) -> (lSuccess==1))

5) If there is no compensation request from LoanStar and if the buyer accepts the quote, then the quote accepted variable is set.

ltl1( [] (buyerAccepts && lCompensate==0) -> (sqsSuccess==1))

VI. CONCLUSIONS

Modelling, analysis and implementation of complex business transactions is not a trivial task. In this paper we present a technique that could be helpful in solving this problem. We propose to use cCSP for modelling of business transactions, then to translate cCSP model to Promela and to analyse it using SPIN. In such a way a language designed for compensations in flow composition languages, then to translate cCSP model to Promela and to analyse it using SPIN. We have defined a procedure for translating cCSP model to Promela language and exemplified using realistic Car Broker example. The results seem very promising. However, these results are just work in progress, because it is necessary to define formal translation from cCSP to Promela to be able to show how analysis results translate back to cCSP model.

ACKNOWLEDGMENT

This paper is partially supported by Research Grant from National Natural Science Foundation of China (61103029) and the project VIZIT-1-TYR-003 (VP1-3.1-ŠMM-01-V-001), Research Council of Lithuania.

REFERENCES