Design and Implementation of a BPMN to PROMELA Translator

Wenzhong Sun (Jim)

MSc in Advanced Computer Science,
School of Computing Science, Newcastle University, U. K.
August 2012
w.sun2@ncl.ac.uk

Abstract. An Electronic Contract (E-Contract), is the electronic version of a traditional paper based contract. Electronic Contracts are widely used in the business world today. How to convert from a traditional contract into an E-contract is a considerable challenge for system architects. Service Choreography tools (e.g. BPMN) can be employed to help designers with modelling business processes described within business contracts. Verifying such a choreography conforms to the contract requirements is an extremely important yet challenging task. Manual verification is extremely difficult because of the potential complexity of the choreography model. Therefore in this dissertation we propose using model checking tools (specifically the SPIN model checker) in order to be able to analyse choreography models systematically. We propose that a BPMN choreography can be converted into PROMELA, which is the input language of the SPIN model checker. To automate the conversion process from BPMN to PROMELA. I have built a BPMN to PROMELA translator, which I describe in this dissertation.

Declaration: I declare that this dissertation represents my own work except where otherwise explicitly stated.

1 Introduction

Business to Business Interactions (B2Bi) conducted over the Internet are regulated by legal business contracts signed between two or more participants; for example, between a buyer and seller that have agreed to conduct business with each other.

A business contract contains a list of normative statements or equivalently, as a list of rights, obligations and prohibitions that the signatory parties are expected to observe.

The clauses of the a business contract precisely stipulate what activity (also called operations or actions) the parties are expected to execute, when and in which order to honour their rights, obligations and prohibitions.

A hypothetical example of a business contract (or just a contract) between a buyer and store, with typical activities (buy request, buy confirmation, buy rejection, pay and cancel) is shown below:

1. *The buyer has the right to place a* ***buy request*** *with the store to buy an item.*
2. *The store is obliged to respond with either* ***buy confirmation*** *or* ***buy rejection*** *within 3 days of receiving the buy request.*
3. *The buyer can use its discretion to either* ***pay*** *for or* ***cancel*** *the buy request within 7 days of receiving a confirmation.*

The execution of each activity involves the participation of the two business partners which interact in a peer–to–peer relationship as against the more traditional client-server relationship.

The implementation of contracts results in intricate interactions that are conveniently expressed as a cross–organisational business process executed between the business partners. Realistic contracts normally result in cross–organizational business processes of considerably complexity that are likely to suffer from logical errors. For this reason, contracts are normally represented at different levels of abstractions that serve different purposes. Abstraction levels range from informal models aimed at humans to formal ones aimed at computers and amenable to systematic manipulation and reasoning with the intention of uncovering potential errors.

For instance, widely used models are choreographies which allow systematic reasoning at design time but without accounting yet for the particularities of implementation technologies. We believe that contracts should be represented as choreographies and scrutinise for potential logical flaws before modelling (or implementing) them as distributed systems. As elaborated below, choreographies represent a B2Bi at message level. Once the designer is satisfied about certain correctness requirements verified on the choreography, the latter is used for producing (either mechanically or manually) the actual public processes (one for each participant) that implement the cross–organizational business process. This idea is illustrated in Fig. 1.

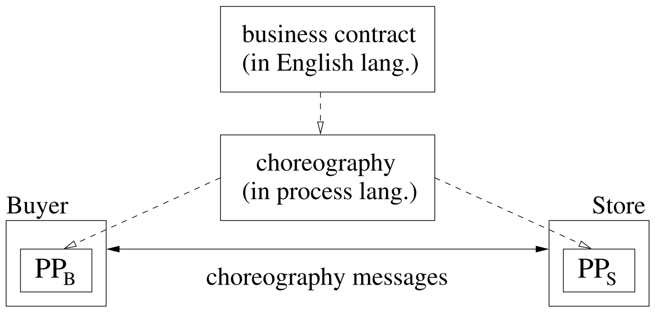


Fig. 1. A contract, its choreography and public processes.

In the figure, Buyer and Store represent two arbitrary contracting parties that have agreed on the terms and conditions included in the business contract expressed in English language. The dashed lines represent production, so the business contract is used for producing the choreography which is expressed in a process language. Similarly, the choreography is used for producing PP_B and PP_S. PP_B and PP_S stand for the buyers and store's public processes respectively. The solid line represent a communication channel used by the Buyer and Store for sending the choreography messages involved in the execution of each activity included in the contract.

Choreographies offer a global view of a message–based interaction and are specified in choreography languages such as [1]. In this order, a choreography of a bilateral interaction would include the sequences of messages exchanged between the two business partners as observed by a third party with a global view of the interaction.

Choreographies are not directly executable. They are only abstract models of a cross–organisational process. They can be used for several purposes. As shown in Fig. 1, they can be used to derive the public processes of the business partners. More importantly within the context of our discussion, choreographies can be used for reasoning at design time about different aspects of the behaviour of the contract that they represent. In this order, choreography models can help determine whether the business process satisfies certain safety and liveness properties, such as absence of deadlocks, causality (message order) and non–progress cycles. For this to be possible, choreographies need to be expressed in notations that are amenable to systematic analysis with the assistance of software tools such as model checkers.

Several choreography languages have been suggested by both industry and academic institutions. A categorization of choreography languages is presented in [2] and re–examined in [3]. A language that has achieved wide acceptance and the focus of interest of this dissertation is BPMN. BPMN is relatively new: version 1.1 was approved in 2006 whereas its latest version 2.0 was released in 2011 ([1]) and still under very active scrutiny. Consequently, mechanical tools for reasoning about of BPMN process are still the subject of research activities. To help cover the gap, in this dissertation we design, implement and discuss a translator that can mechanically produce PROMELA code from choreography diagrams expressed in BPMN, precisely in the RosettaNet version [4] of BPMN 2.0. We will refer to it as the BPMN2PROMELA translator. As explained at large in Section 2, PROMELA is the input language of SPIN, a mature, widely available and well documented model checker. The PROMELA model produced by the translator can be used for uncovering potential logical errors included in the BPMN choreography diagram. The idea is illustrated in Fig. 2.

In Fig. 2, BPMN editor represents a tool for generating BPMN2 choreography diagrams. BPMN choreography is a diagram created with the BPMN editor. Actually, a BPMN choreography is an XML files. BPMN2PROMELA is the BPMN choreography to PROMELA translator that we have implemented in this dissertation. It is a Java application which can be configured and run on any platform. We will discuss it in detail in Section 3. As pointed out in Fig. 2, a valid BPMN choreography is presented as input to the BPMN2PROMELA

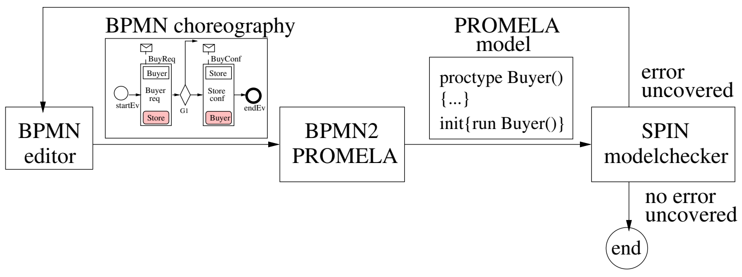


Fig. 2. BPMN2PROMELA translator used to uncover choreography errors.

translator. Then, the corresponding PROMELA model is obtained as the output of BPMN2PROMELA translator. The PROMELA model is presented to the SPIN model checker which is a general tool for verifying the correctness of distributed software models in an automated way to uncover potential errors. If no error is uncovered by SPIN, then whole process ends. Otherwise, it will loop back to the very beginning to inspect behaviours described in the original BPMN choreography.

The PROMELA model can be used for other purposes as well. For instance, and as explained in Section 5, it can be used for mechanically producing the message sequences encoded in the original BPMN choreography.

To show the practicality of the concepts presented in this dissertation, we discuss them within the context of SAVARA Testable Architecture [5] (an on going Jboss project aimed at creating tools for designing business processes) framework. However, our concepts and ideas, including those behind the BPMN2PROMELA translator, are rather generic and can be used in other frameworks.

1.1 Proposed Aim and Objectives

The primary aim of the project is to *design and implement a BPMN to PROMELA translator* by using Java.

The following points have been identified as main objectives of the project:

– To understand why we need a BPMN to PROMELA translator.
  We would like to verify a BPMN choreography with SPIN. However, SPIN only takes PROMELA as its input language. Therefore, the BPMN choreography needs to be converted into PROMELA model. But currently, there are no translators for translating from BPMN choreographies into PROMELA models. So we need to build a translator to bridge the gap between them.
– To investigate related fields that associated with this translator.
  In this dissertation, we want to translate a high level language into another high level language. Therefore, full understanding about BPMN constructs and PROMELA syntax is a very important objective in the dissertation.

– To design and implement the translator.
  To design and implement a BPMN2PROMELA translator is the main objective of this dissertation. In this project we want to build a XML based Java application that translates BPMN choreographies into PROMELA models.
– Symbolic execution based sequence generation of BPMN choreographies.
  Sequences in BPMN diagram are very important for contract experts and system designers. How to generate message sequences automatically from BPMN choreographies is an issue for us.
– To validate PROMELA models which generated by the translator in SPIN and analyse results of typical scenarios.
  Analyse some typical scenarios, in order to get a idea of how the translator helps us in design level.

1.2 Structure of Dissertation

This dissertation is organised as following:

The background in Section 2 will primarily introduce what is E-Contract, choreography and orchestration, Jboss Savara Testable Architecture, PROMELA, SPIN and Linear Temporal Logics.

Section 3 introduces different alternatives of translation. Then this section shows the structure of BPMN2PROMELA translator. After that, the overall design and implementation of the translator will be discussed at the end.

We will analyse some typical BPMN choreographies in Section 4 in order to get a clear idea about how the translator helps us in design level.

Section 5 discusses basic concepts about events in distributed systems. Then we will briefly introduce methods of sequence generation from BPMN choreographies.

Section 6 introduces the testing strategy used for testing core components of the translator. Then we evaluate the aim and each objective of the dissertation. At the end of this section we will evaluate the BPMN2PROMELA translator.

Some previous work in this area is discussed in Section 7.

The conclusions and future work in Section 8 ends up the dissertation by summarizing the work which has been done and points out some further work that can be done about this project.

2 Background

2.1 Electronic Contracts

A business contract stipulates what activities the contracting parties are expected to execute. In our execution model, the execution of each activity always involves the participation of the two parties that interact with each other in a loosely coupled manner. In principle, the execution of activities can involve more than two parties at the price of more complexity; this possibility fall outside the scope of this dissertation. The execution model that we follow is discussed in

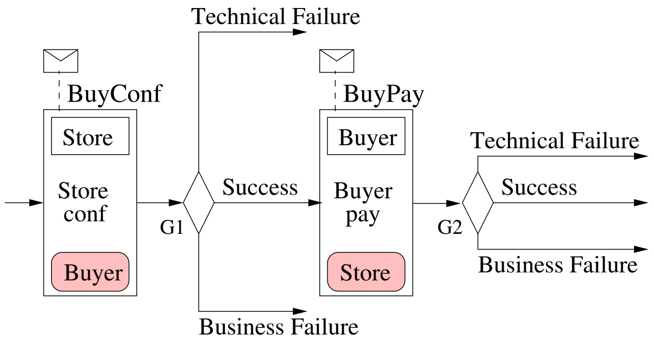


Fig. 3. Execution model of activities.

depth in [6], we present here only a brief summary. We assume that each execution involves an initiator and a responder. The initiator is the party that initiates the execution, the responder is the party that responds to the execution. We assume that once an execution is initiated, it always terminates and produces an outcome. The execution of each activity within each party always and **independently** produces one out of three possible outcomes: *success*, *business failure*, or *technical failure*. *Success* (*es*) represents the desirable (normal) outcome; whereas *business failure* (*bf*) and *technical failure* (*tf*) represent undesirable outcomes due to, respectively, business and technical reasons. By *independently* we mean that for the execution of a given activity the two parties can potentially produce conflicting outcomes (for example, one of them declares *s*, whereas its counterpart declares *bf*). To keep the two participants aligned, synchronization (both declare *s*, *bf* or *tf*) mechanisms are used (see for example [7]). The use of synchronization mechanisms is assumed in the RosettaNet version [4] of BPMN. Since the BPMN diagrams shown in this dissertation follow this specification, they also assume the existence of underlaying synchronization mechanisms. The execution model of each activity that we assume in this dissertation is shown in Fig. 3.

Observe that the outcome of each activity (*Store conf* and *Buyer pay*) is guarded by exclusive gateways (*G1* and *G2*, respectively) with three alternative execution paths. The target of *Technical Failure* or *Business Failure* outcomes is not shown in the figure. The assumption is that these paths lead to activities that handle the undesirable situation, such as repetition of the failed execution.

For the sake of simplicity, in some of our examples of choreographies (in Fig. 4 for instance) we show only the normal (primary) execution paths.

2.2 Choreography and Orchestration

The concepts of choreography and orchestration as defined in firstly introduced in [8], are used to refer to different views of a business process. Another way of looking at these terms is that they represent (model) the same business process at different levels of abstractions, presumably, with the intention of capturing different aspects of its behaviour.

A choreography refers the description of how multiple participants (or roles) interact to achieve a goal. It is used to capture all interaction obligations and constraints from a global perspective. On the other hand, an Orchestration represents executable business process from participant's perspective. In Fig. 1 for instance, the choreography specifies interactions between Buyer and Store in an abstract process model. Whereas the orchestration specifies public business processes for each parties which projected by the choreography.

Since they represent the same business process, orchestration can be generated automatic projection from choreography.

2.3 Jboss Savara Testable Architecture

Testable Architecture is a new methodology designed and developed by Red Hat and Cognizant Technology Solutions in 2009. It can be regarded as a framework with the purpose of supporting designers and architects in designing and developing distributed business applications. This methodology mainly focuses on message and service oriented system.

Savara is a on going project established and maintained by JBoss [5]. Its aim is to build a framework that based on the Testable Architecture methodology which can help us building distributed and service oriented systems. Different from other architecture tool kits, the goal of this project is to ensure that all artifacts created throughout the life cycle of a software development project are verifiable against other previously defined artifacts. It means that the purpose of the framework is to make sure the verifiable of all parts of a software project. In another word, it assists us to ensure that delivered system will still conform to the original software design (therefore business requirements) for the whole lifecycle of a project. Under the guidance of such a theory and employ tools that have been and will be developed in Savara, the final system has the possibility to fully apply the business requirements.

Savara offers several tools for supporting designers at different stages. We will focus our discussion only on those that are relevant to this dissertation, namely: the BPMN diagram editor, scenario editor and the scenario simulator.

The BPMN Diagram editor BPMN diagram editor here refers to BPMN2 choreography diagram generation tool which has been installed on eclipse platform as a plug-in. It corresponds to the BPMN editor of Fig. 2. As a part of SAVARA tool suits, this editor enable us generate BPMN choreography conveniently by simply dragging and dropping BMPM elements from the icon menu to the canvas. Actually, these BPMN diagrams are encoded as conventional and standard XML files that can be used and manipulated by XML supportive tools like Dom4j (A third party library for working with XML on Java platform). Diagrams which created by using the editor can be employed for several purposes. For example, they can be used to verify scenarios which are generated by

SAVARA scenario editor (we will discuss it later). As mentioned in Section 1 in this dissertation we use them as input files for creating PROMELA models. Basically, it is a convenient graphical user interface (GUI) that allows developers to create choreography documentation by working with images rather than typing texts. Currently, it supports some basic BPMN elements such as: Connectors, Tasks, Gateways, Events, Data Items and other elements. The information on how to build a choreography can be found in the SAVARA User Guide.

The scenario editor SAVARA also offers a scenario editor to help developers create scenarios by using a GUI tool. A model is a highly abstract manifestations of a system of interest for designers. A model is competent for a given analysis if it contains sufficient detail to permit that analysis. A simulation is a symbolic execution [9] of a model. And what is a scenario? "The inputs and stimuli provided to a model during a simulation run is termed a scenario [10]." The scenario editor in combination with the simulator and the BPMN diagrams helps designers perform a simulation action easily by choosing the model that need to be checked and a type of simulator. Note that these scenarios are also encoded as conventional and valid XML files and suffixed as "scn".

The scenario simulator A scenario simulator in SAVARA is a software tool which can perform certain type of simulation action of a scenario specification against a choreography model. We have known what a model is. And a model which can be simulated is called an executable model. In this case, a choreography model is an executable model because it can be put into a simulation environment and produces corresponding results. Actually it performs a symbolic execution of the BPMN choreography diagrams against the scenario provided as input. A symbolic execution refers to the analysis of programs by tracking symbolic rather than actual values. When a simulation is acting, message events (associated with the specific roles) in the scenario are simulated against the models specified in a option dialog. If the message event is valid, its node will be displayed in green. If however the event is unexpected, then it will be shown in red instead. However, how does a simulator work is totally transparent to users.

Simulator stands at abstract level and employs symbolic execution method to analyse programs by tracking symbolic values.

2.4 PROMELA and SPIN

PROMELA (Process Meta Language) is a language for specifying business models and the input language accepted by the SPIN model checker. As thoroughly discussed in [11], PROMELA has some syntax to represent key elements such as processes, channels, messages etc. which are abstracted from message based distributed systems. Basically, it is a language for specifying verification models [12]. By verification models we mean abstract design which only includes properties that we want to verify. It is usually used in abstracting distributed systems, especially focus on the exchange of messages between business parties.

We will describe here only the constructs that we use in our BPMN2PROMELA translator in Section 3.

SPIN is a software tool for validating models of distributed system (e.g. communications protocols [13]) written in PROMELA. It can simulate the execution of a PROMELA model and reveal designers whether their designs conform to the original requirements. As a valuable design tool it can be utilized at any level of abstraction. In general, SPIN is a model checker that employs state exploration to validate properties which defined in PROMELA model[14]. Model checking [15] is the main work which done by SPIN. Model checking apply some algorithm to verify concurrent systems. These systems are described by using PROMELA and specified with some properties. SPIN exhaustively verifies the system and returns the result of correctness.

There are two ways of initializing the execution of a PROMELA process. First, put it into a "init" block. The "init" block is used to declare the behaviour of a process that is active in the initial system state. The obvious advantage of such approach is that, we can do some complex initialization work (e.g. define channels, pass parameters into predefined process prototype) before we run it. The second way is that a "active" prefix can be added before the declaration of a proctype. It defines processes that are required to be executed in the initial system state. In this dissertation, some figures have been shown as first approach and some code examples is represented in the second way. This two approaches is identical in the paper.

2.5 Linear Temporal Logics

Assertion is an important language construct in PROMELA. It produces some results(e.g. error) during verifications with SPIN in accordance with parameters it holds. Correctness properties of concurrent programs that written in PROMELA can be verified and specified by using assertions. But assertions cannot express all of these properties. It means that for majority correctness properties of PROMELA models, assertions are not enough.

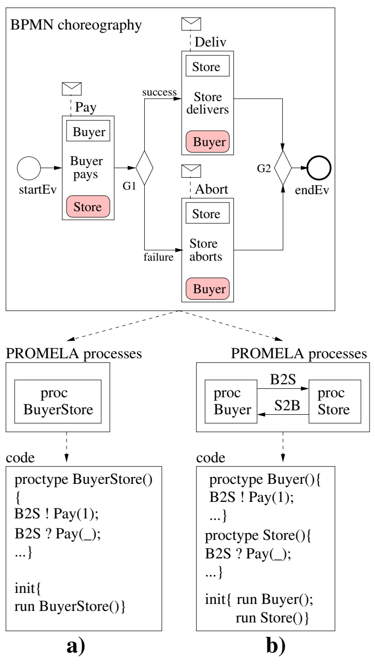
Linear temporal logic (LTL) is the formal logic used for expressing correctness requirements expected to be observed by abstract models like those written in PROMELA and validated with SPIN[16]. LTL are used for two different purposes. First, they are used for expressing correctness properties. For example, a choreography designer can use them to express some requirements that his or her choreography should meet such as "message A is always followed by either message B or message C". LTL are also used for expressing trap properties in model checking-based testing. The aim here is not to uncover logical errors by to generate sequences automatically from a validated PROMELA model [17]. The syntax and semantics of LTL are not within the range of this dissertation. Therefore, we do not discuss them here.

Elementary and practical introductions to LTL with focus on SPIN are presented in [13, 17].

3 BPMN to PROMELA Translation

3.1 Choreographed vs Orchestrated Translation

A given BPMN choreography can be translated into different alternatives: choreographed or orchestrated. The idea is illustrated in Fig.4. At the top of the figure, there is a BPMN choreography which involves two parties (Buyer and Store). Choreographed translation is illustrated as alternative a) in the figure. With this approach only one process is created during the translation regardless of the number of participants included in the BPMN choreography. Communication activities (e.g. sending and receiving messages by each party) are within this process which is shown as process BuyerStore in the PROMELA process box and as proctype BuyerStore in code box. The orchestrated translation is illustrated as alternative b) in the figure. With this approach, the PROMELA model created has as many processes as participants are in the choreography model plus the init process to initiate them. The two boxes on the right bottom show the situation. Process Buyer and Store (proctype Buyer and proctype Store in code box) represent the two participants of the BPMN choreography. However, it is worth observing that communication operations of orchestrated translation is explicitly modelled. Communication messages are transferred from one process to another via channels (e.g. B2S and S2B).



```
proctype BuyerStore()
{
B2S ! Pay(1);
B2S ? Pay(_);
...}

init{
run BuyerStore()}
```

```
proctype Buyer(){
B2S ! Pay(1);
...}
proctype Store(){
B2S ? Pay(_);
...}
init{ run Buyer();
      run Store()}
```

Fig. 4. Choreographed and orchestrated translations.

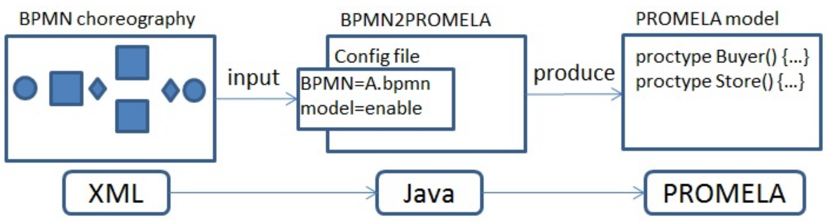


Fig. 5. The structure of BPMN2PROMELA translator.

3.2 Structure of BPMN2PROMELA Translator

The BPMN2PROMELA translator is meant to produce PROMELA models from BPMN choreography diagrams mechanically. It was implemented in Java within Eclipse framework. Fig. 5 shows the structure of the translator. BPMN choreography in the figure refers to a BPMN diagram created by using SAVARA BPMN editor. A BPMN choreography is actually organized and stored as an XML document. Therefore, it can be processed programmatically. *Config file* is a configuration file for the translator. For example, it allows the designer to specify the size of channel buffers, the storage path of PROMELA model files, the type of messages and the BPMN file to be translated. More information about configuration can be found in Appendix A.3. The translator follows the configuration file when execution and produces a corresponding PROMELA model automatically.

3.3 Basic BPMN Constructs

Our translator supports only some of the constructs specified in the BPMN 2.0 standard. Precisely, it follows the RosettaNet version of BPMN which is a subset of BPMN 2.0. Before discussing the translation from BPMN choreography to PROMELA model. We will discuss the BPMN elements that we support in our translation. We will use a BPMN diagram (the BPMN choreography of Fig.4) as example to support our explanation.

Events: An event represents something that can be notified during the process of a workflow. Events in a flow always generate some results. There are three types of Events defined in BPMN: Start, End and Intermediate events.

Currently, this project supports only Start and End events. In BPMN, circles are used for representing events, thus in Fig. 4, *startEv* and *endEv* represent, respectively, the start and end events of the process.

Activities: An activity represents a task that can be achieved by business partners during the process.

The activities are represented by rectangle boxes that indicate the names of the activity, involved parties and messages. Fig. 4 includes three activities called

Buyer pays, Store delivers and *Store aborts.* They represent the Buyer's payment request, the Store's delivery notification and the Store's cancellation of payment request, respectively. The involved parties (participants) are named inside bands (top and bottom of an active box) of different colours. The sender's in a white band and the receiver in a shaded band.

Messages: A message represents an abstract information which can be transferred from one party to others.

The RosettaNet version of BPMN stipulates that there are only two participants in each activity which are a sender and a receiver. There are three messages in Fig.4, they are *Pay, Deliv* and *Abort* which represent *PaymentRequest, DeliveryNotification* and *AbortNotification*, respectively. For example, in the *Buyer pays* activity, the *Buyer* sends the *Pay* message to the *Store* and the *Store* finally receives it.

Gateways: A gateway is used to control the flow of sequences in a choreography. They are represented by diamonds. BPMN 2.0 supports different types of gateways(e.g. Exclusive, Inclusive, Parallel, Event-Based, and Complex). Gateways control the splitting and merging of sequence flows.

Currently, only exclusive gateways (split and merge) are supported in this version of the translator. Fig. 4 includes one exclusive split gateway (*G1* and corresponding exclusive merge gateway *G2*).

Sequence Flows: A sequence flow is used to express the order between events, activities and gateways.

It is specified by arrowed line that connect events, activities and gateways. Fig. 4, for example shows a sequence flow from start event to Buyer Pays activity.

3.4 Basic PROMELA Constructs

To help the reader understand how the translator works, we will explain here some of the central PROMELA constructs.

Global Variables: In PROMELA, variables can be divided into several types. Our translator supports two of them: First, Enumerated types are used to define the messages in a choreography such as *mtype* = {*Abort, Pay, Deliv*}. The syntax in PROMELA is expressed as *mtype (=)?* {*msg (,msg)**}. Second, LTL variable types: they are used to include LTL correctness properties into PROMELA model, this kind of definition will be placed at the beginning of the content. For example, *#define p (ture).* The corresponding PROMELA syntax is *#define variable (bool).* The current implementation of the translator does not support basic data types or Structures.

Global Channels: In PROMELA, channels can be declared either global or local. Channel declaration specifies buffer size and data type. Only the type of declared messages can be transferred through the channel. For example, *mtype* = {*BuyConf, BuyPay*} defines all messages involved in a choreography. So *chan Store2Buyer = [0] of* {*mtype, byte*} is the declaration of a channel. It can accept messages of mtype and byte and its buffer size is zero. Therefore, *BuyConf and BuyPay* are all allowed to be communicated by using *Store2Buyer*. In this translator we use only global channels. The syntax in PROMELA can be described as *chan channelName = [INT] of* {*typename(, typename)**}.

Processes: Currently, as mentioned in Section 2, there are two ways to define processes for a BPMN choreography. In choreographed translation there is only one process in PROMELA model whereas an orchestrated translation will create as many processes as participants are defined in the choreography. The body of a process includes a number of statements or blocks that can be one of atomic block, message activities (sending and receiving), selection construction (if block), GOTO statement, labelled statement, SKIP statement and run statement. Moreover, there are two means to initialize processes. The "active" keyword can be used to initialize each process or alternatively the designer can initialize them in an "init" block. Both approaches are supported in this translator by configuration. The syntax in PROMELA should be *(active [N]) proctype processName (args)* {*statements*}.

Message Communications: The syntax for message sending and receiving in PROMELA is represented as $chan!msg(1)$ and $chan?msg()$ respectively. For each task in the choreography, results in a sending statement and corresponding receiving statement. The format in the translator for message communications can be described as $participantA2participantB?(!)msg$. *participantA* and *participantB* represent the sender and the receiver in a task, and *msg* is the message of the task. For example, suppose that in a task Buyer sends BuyReq to Store, then the corresponding message communications are $Buyer2Store!BuyReq(1)$ and $Buyer2Store?BuyReq()$.

Selection Constructions: The syntax of selection constructions can be described as *if (::(condition)? statements)+ fi*. *Condition* is a bool value, the option can be executed if it is true. Basically, a split gateway can be translated into an if-fi block. However, there are some exceptional circumstances during the implementation work. We will discuss them in following sections.

GOTO and Labelled Statements: Within a process, a label must be uniquely identified. They are usually associated to GOTO statements. In this translator, these statements only appear when loop paths were involved in the BPMN choreography. The translation syntax is as follow: *label: statements ... GOTO label.*

Table 1. Mapping from BPMN constructs to PROMELA syntax.

BPMN constructs	PROMELA syntax	explanation
start event	global variables, global channels, blank processes, LTL properties	initialize a framework for PROMELA model. If non-active processes, then init block and RUN statements
end event	nothing	indicates the end of translation
activity	a channel, a message sending statement, corresponding message receiving statement (according to sender and receiver)	if loop paths, then GOTO statements and labelled statements; if message activities are the first statements of a if-fi blocks, then colon blocks
participant sender	a process	repeated sender will be ignored when creating processes
participant receiver	a process	repeated receiver will be ignored when creating processes
exclusive split gateway	selection constructions (if-fi blocks)	an if-fi block will be deleted under some exceptional circumstances [18]
exclusive merge gateway	nothing	a signal for the end of if-fi block; a pointer for message activities; if it follows a split gateway immediately, then SKIP statement
message	a message definition, construct message communications	define a message as global variable
sequence flow	nothing	used for going through all paths in the choreography

Run Statements: As mentioned before, there are two ways to initialize models. Run operator within init block is an option for this translator. The translation syntax in PROMELA is *atomic* {*run processA (args); run processB (args); ...*}.

3.5 Mapping from BPMN Constructs to PROMELA Syntax

In this section, we will discuss the mapping from BPMN constructs to PROMELA syntax. We will focus on the orchestrated translation of Fig.4. TABLE 1 indicates the relationship between BPMN constructs and PROMELA syntax. Choreographed translation is slightly different from orchestrated one. We will discuss it later.

Basically, the procedure of translation can be described as shown in Fig.6. We can see that the translation can be divided into two stages: first, *parsing stage*, in this stage, a BPMN choreography diagram (XML file) is parsed into a set of interrelated Java objects. These objects represent BPMN constructs

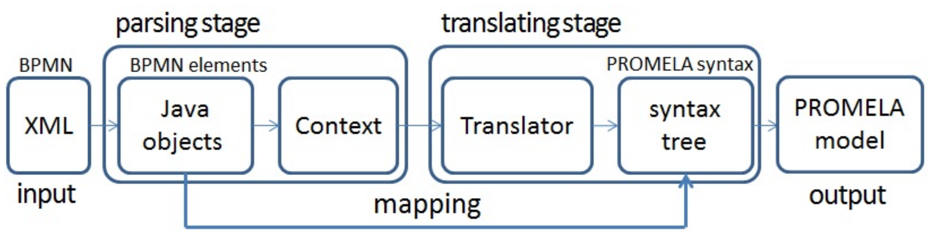


Fig. 6. Stages in translation.

in this domain. They are stored within a *Context*, which in this project has been represented by a Java class. Information such as messages, participants, gateways, and tasks are stored (after parsing them) in the *Context*. These objects are retrieved from the *Context* as needed during the translation. Then at the second stage, namely, *translating stage*, a *Translator*, which is a key class in this stage, reads the *Context* and produces a syntax tree in memory by following the mapping explained in TABLE 1. Finally, the translator work through the syntax tree [19] and transforms it into PROMELA model which is stored in a plain ascii file.

A start event can be translated into a framework (all messages, channels and processes) for a PROMELA model. However, in fact, start event does not have those necessary information, it only indicates the beginning of a translating. When the translator recognizes that there is a start event, it will get these information from *context*. An end event does not translate into any BPMN construct, it is just a sign for telling the translator that it is the end of the translation. An activity is translated into a channel in the parsing stage and message activities (sending and receiving) in translating stage respectively. Note that if these message activities are the first statements of a selection construction (if-fi block), a colon block will be created as a child of the if-fi block. If the diagram includes loop paths, then labelled statements and corresponding GOTO statements would be added, because an activity will be passed more than once. Each participant represents a process in orchestrated translation. An exclusive split gateway can be translated into selection constructions and placed into each process. However, under some exceptional circumstances it will be removed from the syntax tree. First, if a selection construction has only one option, then it will be removed and all its children will be appended to its parent node. Second, if a selection construction has zero options, it will be removed directly. An exclusive merge gateway can be translated into a SKIP statement only if it follows a split gateway immediately (see Fig.9). In addition, it is also an important signal point. A message (whose type is configurable) in a task will be translated into a global variable.

3.6 Translation from BPMN Choreography to PROMELA Model

This section we will discuss some key aspects of the design and implementation of the translator.

Abstract and Concrete Syntax Tree An abstract syntax tree (AST) is a tree-like data structure which is widely used in language related tools such as interpreters, code generators or compilers. Typically, it is a ordered and rooted tree that represents structures in certain type of programming language. Basically, there are two types of nodes (components) in the construction of a tree: branch nodes and leaf nodes. A branch node is a node that can contain other nodes. A leaf node cannot contain other nodes, it only can be embodied into a branch node as its child. Each node actually represents a construct of code. In this case, branch nodes include PROMELA syntax like *proctype definition block or if-fi block*. Leaf nodes include syntax such as *variable declaration, channel declaration or message activities (sending and receiving)*. The translation involves the construction of PROMELA syntax trees for BPMN choreography diagrams. Precisely, the PROMELA syntax will be organized in corresponding Java objects [20, 21].

Different from abstract syntax tree, a concrete syntax tree always reflect syntactic structures concretely. It usually represents strings according to the grammar of the language. In this project for example, *chan!msg, chan?msg or* $mtype = \{Message\}$ are nodes from syntax tree. These two types of trees can be used within the same domain, but they stand at different levels.

Fig. 7 gives us an idea about how to parse a BPMN choreography into a syntax tree; then map it to PROMELA code fragment. b) shows the syntax tree with labels (1 to A). The corresponding PROMELA code is shown in c). The branch nodes have been circled by smooth square with a label on its edge. In general, each node has a place within the PROMELA code document after translation.

Design Patterns of Java in this dissertation Design patterns are some reusable solutions that help designers solve problems which recurring during the development of software systems [22]. They can be divided into three types: creational patterns, structural patterns and behavioural patterns. In this project, some design patterns were used during the design work. The most important design pattern in the project is the *Composite Pattern* [22, 23]. This project is language related, so tree structure is used here to organize and represent PROMELA syntax. In our project, *Composite Pattern* enables us to treat PROMELA syntax objects in the same way. It helps us construct the syntax tree and treat the nodes uniformly. We use block and inline represent branch and leaf respectively. Fig. 8 shows the structure of this pattern.

Block nodes represent syntax like process, if-fi etc. Inline nodes represent message sending, receiving, GOTO statement etc. For example, a process block (label 7 in c) of Fig. 7) can contain other components (block or inline), but a

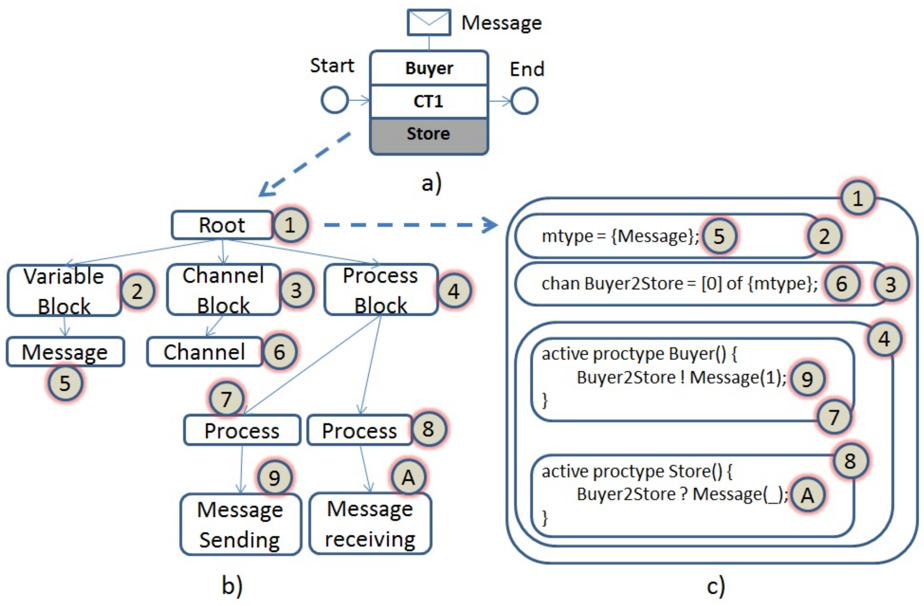


Fig. 7. The example of syntax tree.

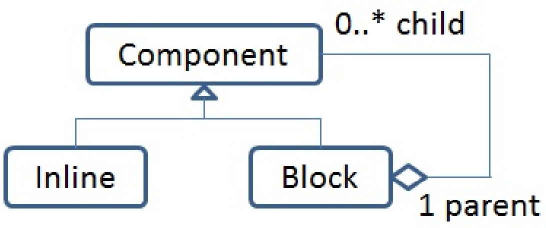


Fig. 8. The structure of composite pattern.

message sending statement (label 9 in c) of Fig. 7) only can be included into a block (e.g. process block or if-fi block).

4 Verification of Translated BPMN Choreographies

To show the benefit and proof the test the soundness of the BPMN2PROMELA translator, we will discuss in this section some of the most representative translations we have conducted. We will also show some logical errors that SPIN has uncovered from the PROMELA models produced by the BPMN2PROMELA translator.

4.1 Contract Example

Imagine that a choreography designer is commissioned to design a BPMN choreography for the contract example shown in Section 1 and that he or she produces the choreography shown in Fig. 9. To verify the logical soundness of the choreography, we convert it into PROMELA code with the assistance of our BPMN2PROMELA translator. The PROMELA code generated by our translator with partial editing is shown below after some minor alterations to improve readability. LTL (never block) is shown below.

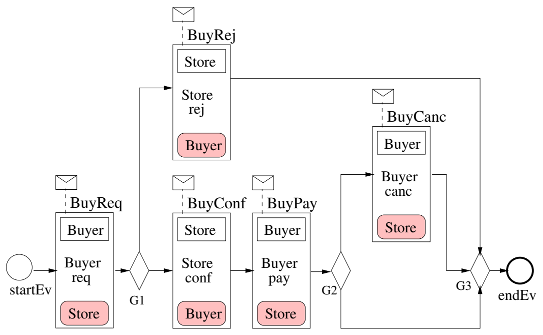


Fig. 9. Incorrect version of choreography of contract example.

```
#define TRUE  1
#define FALSE 0

#define c (conf==TRUE)
#define r (rej==TRUE)
#define p (pay==TRUE)
#define n (canc==TRUE)

mtype = {BuyConf, BuyPay, BuyRej, BuyCanc, BuyReq};

chan Store2Buyer = [0] of {mtype, byte};
chan Buyer2Store = [0] of {mtype, byte};

bool conf= FALSE;
bool rej= FALSE;
bool pay= FALSE;
bool canc= FALSE;

active proctype Buyer() {
Buyer2Store ! BuyReq(1);
if
:: atomic{Store2Buyer ? BuyRej(_); rej=TRUE};
:: atomic{Store2Buyer ? BuyConf(_); conf=TRUE};
Buyer2Store ! BuyPay(1);
if
:: Buyer2Store ! BuyCanc(1);
:: skip
fi
fi
}

active proctype Store() {
Buyer2Store ? BuyReq(_);
if
:: Store2Buyer ! BuyRej(1);
:: Store2Buyer ! BuyConf(1);
atomic{Buyer2Store ? BuyPay(_);  pay=TRUE}
if
:: atomic{Buyer2Store ? BuyCanc(_); canc=TRUE}
:: skip
fi
fi
}

never {    /* !([](c -><>((p && !n) || (n && !p) ) )) */
T0_init:
        if
        :: (((! ((n)) && ! ((p)) && (c)) || ((c) && (n) && (p)))) ->
            goto accept_S35
        :: (1) -> goto T0_init
        fi;
accept_S35:
        if
        :: (((! ((n)) && ! ((p))) || ((n) && (p)))) -> goto accept_S35
        fi;
}
```

After verification of SPIN, we can see a error is generated finally. Details have been shown blow it.

```
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 32 byte, depth reached 18, errors: 1
proc  0 (Buyer) Recv BuyConf, ... (Store2Buyer)
proc  1 (Store) Recv BuyPay, ... (Buyer2Store)
proc  1 (Store) Recv BuyCanc, ... (Buyer2Store)
```

As we can see from the *never* block, the correctness property described in LTL can be expressed in English as *After receiving confirmation, the buyer is* *expected to either pay or cancel.* More technically and in LTL terminology, this property can be expressed as: *always confirmation (c) is eventually followed either by payment (p) or cancellation (n) but not by both.* Then, according to the expressing designer creates a formula. The formula is transformed into a *never* block by using tools such *SPIN*. However, as shown in the code, there is a error after verification. Because SPIN produced a counter-example indicating that the LTL is not satisfied. An exploration of the counter-example shows the following message sequence $BuyReq \rightarrow BuyConf \rightarrow BuyPay \rightarrow BuyCanc$ which is not correct. Therefore, this LTL is violated in this choreography.

Imagine now that after taking into account the pertinent corrections, the choreography designer produces the BPMN choreography shown in Fig. 10. Again, we use our BPMN2PROMELA translator to convert the choreography into PROMELA code which is shown in Appendix B.1.

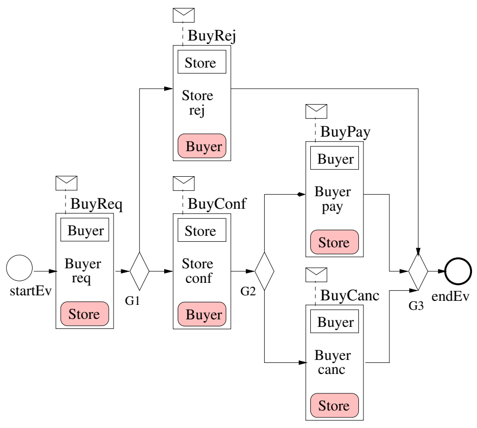


Fig. 10. Correct version of choreography of contract example.

The Result from SPIN validation is shown below.

```
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 32 byte, depth reached 14, errors: 0
```

If we present the SPIN validator with the two pieces of code that derived from Fig. 9 and Fig. 10 into SPIN without inclusion of LTL and verify them, we will find that none of them has error. Is that means all of them meet the original requirement? Let us take an example to explain it. A clause in the textual contract could be described like "after confirmation always follows either payment or cancellation". This statement can be abstracted as a series of message sequences: BuyConf → BuyPay and BuyConf → BuyCanc. From Fig. 10, we can analyse and obtain: BuyConf → BuyPay and BuyConf → BuyCanc. They conform to the design. However, from Fig. 9 we can get the following message sequences: BuyConf → BuyPay and BuyConf → BuyPay → BuyCanc. Apparently, the second one does not conform to the requirement. But this kind of hidden error cannot be detected only by using SPIN. Therefore, LTL is introduced here to help designers uncover these type of problem. The clause can be converted into correctness requirement that can be represented in LTL. These LTL formulas can be included in to PORMELA model and then be used to uncover logical errors like this. Fig. 11 illustrates the idea about how to validate a PROMELA model by introducing correctness properties in LTL. BPMN choreography is the diagram which has been created by using SAVARA tool. PROMELA model is generated by using BPMN2PROMELA convertor. Here, note that correctness properties in LTL is included into PROMELA model and exposed together to the SPIN model checker to test the correctness of the model against the LTL. SPIN produces counter-examples if the LTL is not satisfied by the PROMELA model.

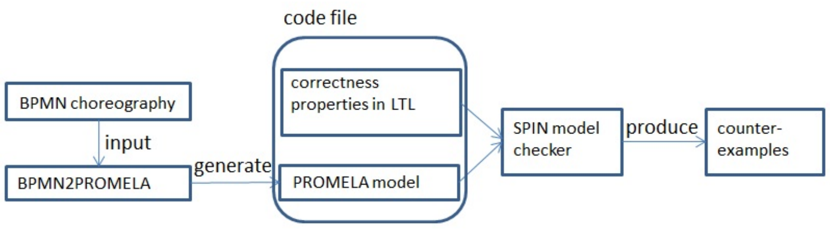


Fig. 11. Validating logical consistency of a BPMN choreography using LTL.

As we anticipated, a validation with SPIN reassured us that the BPMN diagram is logically consistent against conventional safety and liveness properties (deadlocks, unexpected messages, non–progress cycles, invalid end states, etc.) and against a list of LTL properties that we chose to verify such as *buy confirmation is eventually followed by either buy payment or buy cancellation.*

4.2 Purchase Goods Example

Fig. 12 is the BPMN diagram of SAVARA PurchaseGoods example.

The code in Appendix B.2 is its corresponding PROMELA code produced by our BPMN2PROMELA translator.

A validation of the PROMELA code with SPIN against conventional safety and liveness properties reveals that the PurchaseGoods choreography of Fig. 12 suffer from *end state* errors.

```
pan: invalid end state (at depth 10)
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 52 byte, depth reached 16, errors: 1
proc 3 (CreditAgency) terminates
proc 2 (Store) terminates
```

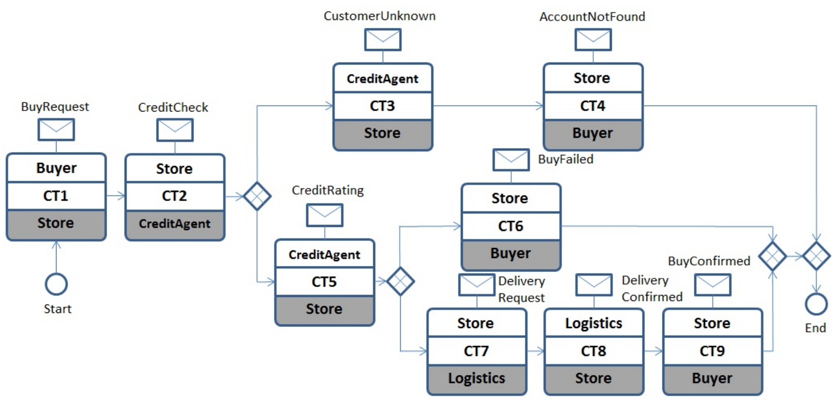


Fig. 12. Purchase Goods SAVARA example

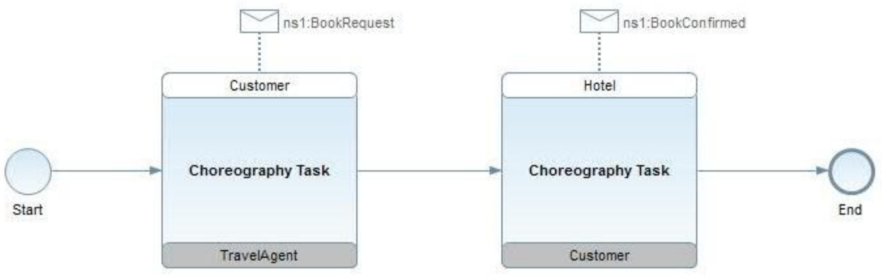


Fig. 13. Unrealizable choreography (example one).

```
proc 1 (Buyer) terminates
proc 0 (Logistics) ... (state 1)
```

Examination of the counter example produced by SPIN reveals that the uncovered error is located within the *Logistics* process. A close examination of the BPMN diagram reveals that SPIN uncovers that the *Logistics* process is left in an invalid end state when the execution does not include the *Good Rating* branch of the *Evaluate Credit Rating* split gateway. One can fairly argue that this is not a serious error. In the PROMELA code it can be quickly fixed by the inclusion of and *end–state* label in the *Logistics* process.

4.3 Unrealizable Choreography Examples

A fundamental question that arises from a choreography specification (expressed in BPMN for example) is whether its is realizable or not. In other words, is the choreography implementable as a sound distributed application composed out of two or more participants? Fig. 13 illustrates the point.

The BPMN diagram includes three participants (*Customer, TravelAgent, Hotel*) and specifies a single message sequence: $\{BookRequest \rightarrow BookConfirmed\}$. At first glance, the choreography seems to be sound, however, a systematic examination with SPIN would reveal that it is realizable only under synchronous communication. Under asynchronous communication, there is no guarantee that the precedence of the two messages will be observed.

To uncover the problem, we translated the BPMN diagram into PROMELA using our BPMN2PROMELA translator. Next we presented SPIN with the PROMELA code and the following LTL property to verify message precedence: *BookConfirmed is always preceded by BookRequest.* The PROMELA code produced from the translator argumented with the LTL which was included manually is shown below.

```
#define TRUE  1
#define FALSE 0

#define c (CreditRcv==TRUE)
#define i (InvRcv==TRUE)

bool CreditRcv=FALSE;
bool InvRcv=FALSE;

mtype = {BookRequest, BuyConfirmed};

/*all channels in the diagram*/
chan Customer2TravelAgent = [1] of {mtype, bool};
chan Hotel2Customer =       [1] of {mtype, bool};

/*all parties involved in the choreography*/
active proctype Customer() {
Customer2TravelAgent ! BookRequest(TRUE);
Hotel2Customer ? BuyConfirmed(InvRcv);
}

active proctype Hotel() {
Hotel2Customer ! BuyConfirmed(TRUE);
}

active proctype TravelAgent() {
Customer2TravelAgent ? BookRequest(CreditRcv);
}

/* LTL expressing precedence BookRequest->BookConfirmed */
never {    /* !(!i U c) */
accept_init:
T0_init:
        if
        :: (! ((c))) -> goto T0_init
        :: (! ((c)) && (i)) -> goto accept_all
        fi;
accept_all:
        skip
}
```

A SPIN verification run signals the violation of the LTL property. Details are shown below:

```
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 44 byte, depth reached 9, errors: 1
proc  0 (Customer) Recv BuyConfirmed, ... (Hotel2Customer)
proc  2 (TravelAgent) Recv BookRequest, ... (Customer2TravelAgent)
```

In addition to the above errors, SPIN produces a counter-example. An examination of the counter-example generated by SPIN and the PROMELA code shows that the PROMELA model can actually accept two message sequences: $BookRequest \rightarrow BookConfirmed$ and $BookConfirmed \rightarrow BookRequest$. The first one is correct, the second one is not.

Let us analyse the realizability of the choreography in Fig. 13. In this example, the buffer size is a very important factor for this property. The choreography is realizable only if the size of channel buffer equals to zero (buff=0). This means that communications between business parties are synchronized. The diagram expresses a design requirement, which can be described as *BookConfirmed is always followed by BookRequest.* From the PROMELA code we can see that these three processes are running simultaneously, this scenario can be described as Fig 14 when buffer size is zero. There are three steps according to the figure:

1. *Customer tries to send BookRequest to TravelAgent and blocks until TravelAgent receives the message.*
2. *When BookRequest is received by TravelAgent, customer unblocks.*
3. *Customer can now receive BuyConfirmed sent by Hotel.*

Under synchronous circumstance (i.e. buff=0), this choreography is realizable because Customer can receive BuyConfirmed from Hotel only after unblocking, this guarantees the sequence of these two messages.

However, the choreography will not be realizable if the size of channel buffer greater than zero (buff>0). This means that communications between business parties are unsynchronized. Consider the situation described above. This time, both messages can be stored into channel buffer (buff>0), which means that as a receiver in the second task, Customer has the possibility to receive BookConfirmed before TravelAgent receives BookRequest. In this situation, BookRequest will followed by BookConfirmed which totally against the original requirement.

Another unrealizable choreography example is illustrated in Fig. 15. This diagram includes four participants (*Buyer, Store, Warehouse, Logistics*) and specifies single message sequence: $\{BuyRq \rightarrow DelivRq\}$. Like Fig. 13, we translated

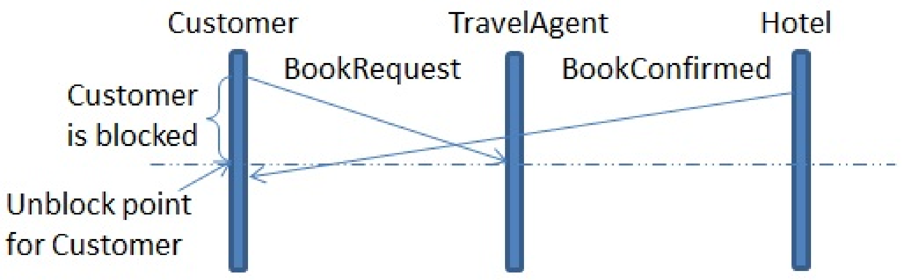


Fig. 14. The scenario when buffer size is zero (example one).

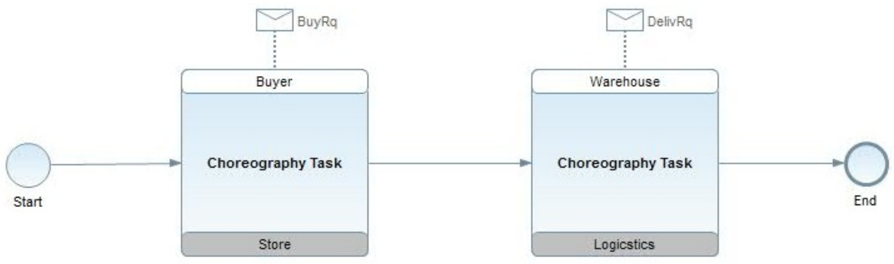


Fig. 15. Unrealizable choreography (example two).

it into PROMELA and then we presented SPIN with the code including the following LTL property to verify message precedence: *DelivRq is always preceded by BuyRq*. The code is shown below.

```
#define TRUE  1
#define FALSE 0

#define b (BuyRqRcv==TRUE)
#define d (DelivRqRcv==TRUE)

mtype= {BuyRq, DelivRq};

chan Bu2Sto=[0] of {mtype,bool};
chan Wh2Log=[0] of {mtype,bool};

bool BuyRqRcv=FALSE;
bool DelivRqRcv=FALSE;

active proctype Buyer() {
Bu2Sto ! BuyRq(TRUE)
}
```

```
active proctype Store() {
atomic{Bu2Sto ? BuyRq(_); BuyRqRcv=TRUE};
}

active proctype Werehouse() {
Wh2Log ! DelivRq(TRUE)
}

active proctype Logistics() {
atomic{Wh2Log ? DelivRq(_); DelivRqRcv=TRUE}
}

never {    /* !(!d U b) */
accept_init:
T0_init:
        if
        :: (! ((b))) -> goto T0_init
        :: (! ((b)) && (d)) -> goto accept_all
        fi;
accept_all:
        skip
}
```

A SPIN verification run signals the violation of the LTL property. Details are shown below:

```
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 48 byte, depth reached 11, errors: 1
proc  3 (Logistics) Recv DelivRq, ... (Wh2Log)
proc  1 (Store) Recv BuyRq, ... (Bu2Sto)
```

Obviously, the choreography in Fig. 15 is unrealizable, because it is impossible for Warehouse to know that Buyer has sent BuyRq to Store before it sends DelivRq to Logistics. It is unrealizable regardless of the type of communication (synchronous or asynchronous). This is a more general case than example one and it can be corrected only with the inclusion of an extra message. Fig. 16 briefly indicates the procedure of correctness. a) is the choreography which represents the choreography in Fig. 15. Possible scenarios (B, S, W and L represent Buyer, Store, Warehouse and logistics respectively, BR and DR refer to BuyRq and DelivRq) of it which can cause the appearance of wrong sequences ($\{DelivRq \rightarrow BuyRq\}$) are shown as b). We can see from these charts that there are gaps between Store and Warehouse. Therefore, these two tasks are completely independent and these messages are occurred concurrently. We can see from these charts that Logistics will receive DelivRq from Warehouse before Store receives BuyRq from Buyer regardless the sequence of message sending. So in these scenarios, the choreography is unrealizable.

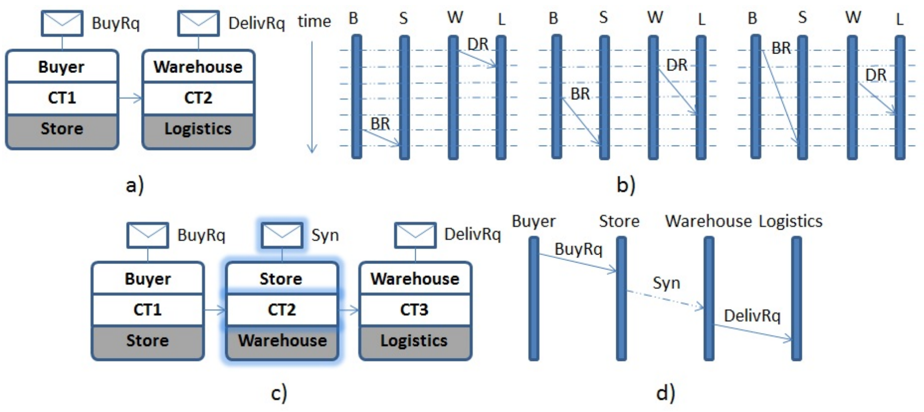


Fig. 16. Correct choreography with the inclusion of an extra message (example two).

As mentioned, we can correct it by including an extra message (namely Syn). Compare with two choreographies (a) and c)) in Fig. 16, we can see that an extra task (with blur edges) has been included between original tasks. After receiving BuyRq from Buyer, Store notices Warehouse with a message in task two. d) indicates that this message (with dashed line) has bridged the gap between Store and Warehouse. Under such circumstance, BuyRq and DelivRq are forced to observe the precedence needed. There is a relation between them now. According to principle three in Section 5.1, the causal precedence between them is $BuyRq < DelivRq$. It conforms to the design requirement. The code of c) is shown below.

```
#define TRUE  1
#define FALSE 0

#define b (BuyRqRcv==TRUE)
#define d (DelivRqRcv==TRUE)

mtype= {BuyRq, DelivRq, Syn};

chan Bu2Sto=[0] of {mtype,bool};
chan Wh2Log=[0] of {mtype,bool};
chan Sto2Wh=[0] of {mtype,bool};

bool BuyRqRcv=FALSE;
bool DelivRqRcv=FALSE;

active proctype Buyer() {
  Bu2Sto ! BuyRq(TRUE)
}
```

```
active proctype Store() {
  atomic{Bu2Sto ? BuyRq(_); BuyRqRcv=TRUE};
  Sto2Wh ! Syn(1)
}

active proctype Werehouse() {
  Sto2Wh ? Syn(_);
  Wh2Log ! DelivRq(TRUE)
}

active proctype Logistics() {
  atomic{Wh2Log ? DelivRq(_); DelivRqRcv=TRUE}
}

never {    /* !(!d U b) */
accept_init:
T0_init:
        if
        :: (! ((b))) -> goto T0_init
        :: (! ((b)) && (d)) -> goto accept_all
        fi;
accept_all:
        skip
}
```

A SPIN verification run signal the violation of the LTL property:

```
(Spin Version 4.3.0 -- 22 June 2007)
State-vector 60 byte, depth reached 8, errors: 0
```

We can see the result from SPIN validation, problem has been fixed by introducing an extra message.

It is worth mentioning that same problem for Fig. 13 can be solved by using similar method, too. Fig. 17 shows the procedure and the basic concept is the same as Fig. 15. Due to space limitations we will not discuss this issue further.

5 Generation of Message Sequences from BPMN Choreographies

The behaviour encoded in a choreography diagram can be regarded as the message sequence that such choreography can accept. It follows that the availability of these sequences is crucial as they can be used for several purposes. For instance, given a message sequences that is known to be correct, one can present it as input to its correspondent BPMN diagram, simulate it and verify if the sequence is accepted or rejected. In this section we discuss two potential alternatives for generating the message sequences encoded in BPMN choreographies.

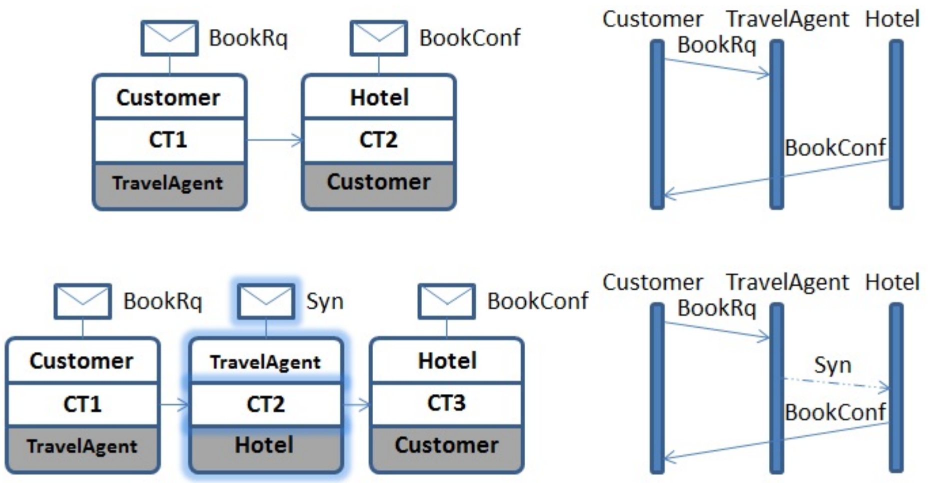


Fig. 17. Correct choreography with the inclusion of an extra message (example one).

5.1 Symbolic Execution Based Sequence Generation

The generation of sequences automatically is important for designers. As told above, an alternative is BPMN $\rightarrow$ Promela code $\rightarrow$ Promela model $\rightarrow$ SPIN $\rightarrow$ sequences. The problem with this approach is that it requires knowledge of model checking and LTL. Are there other alternatives? It is worth exploring. As a potential alternative we have implemented a simple framework which can generate message set and sequence set automatically from a given BPMN. It is only a preliminary implementation but can be used for experimental purposes. Its functionality can be described by the following steps. BPMN $\rightarrow$ framework $\rightarrow$ sequences.Currently, this framework can support simple choreography diagram including loop. Let us take an example to explain how it works.

There are three parties in the example (Buyer, Store and Factory). They agree about the buying and selling of goods. The global model was drawn as Fig.18.

From the figure, we see that there are some messages sent from one process to another. They are all driven by sending events. Here we did not include receiving events because we assume that the message will eventually delivered. So there exist a set $V_{events} = \{BuyRequest(BR), NotEnoughGood(NEG), OrderRequest(OR), OrderConfirmed(OC), BuyFailed(BF), BuyConfirmed (BC), CustomerNotFound(CNF)\}$ which represents valid messages (events) in the model. For any other message, $CheckBalance \notin V_{events}$. According to the description of preceding paragraphs, we can conclude that there must be a $V_{seq} = \{BR \rightarrow CNF, BR \rightarrow OR \rightarrow NEG \rightarrow BF, BR \rightarrow OR \rightarrow OC \rightarrow BC\}$ represents permitted message flows of the model. Actually, each entry in this set indicate a possible path in this diagram. Thus a message flow is correct if and only if it is a subset of any entry in V_{seq}. According to this, we can sum-

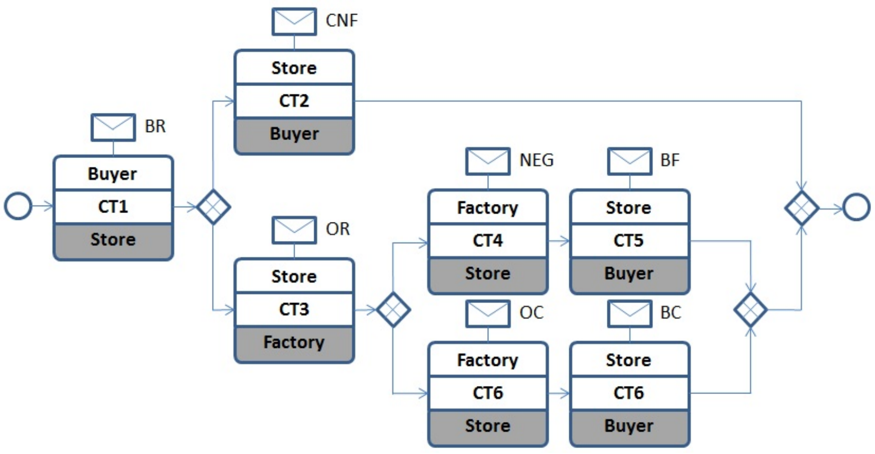
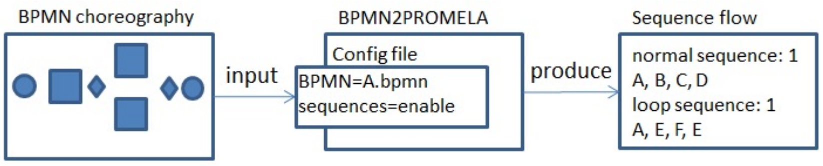


Fig. 18. Global model of choreography example.

mary that there are two types of wrong message flow. First, wrong message. For example, $BR \rightarrow CheckBalance \notin V_{seq}$ because $CheckBalance \notin V_{events}$. Second, wrong sequence. For instance, $CNF \rightarrow BR \notin V_{seq}$ because although $CNF, BR \in V_{events}$, $CNF \rightarrow BR$ is not a part of V_{seq}. So for example, if OR has occurred in the system in one business conversation, then the following message should be either NEG or OC, all others are not valid except these two.

Currently, this sub-function (symbolic execution based sequence generation) has been integrated into the Eclipse project. The sequence generator or the BPMN2PROMELA can be selected by means of the Config file. Users can choose the type of sequence flow file either in plain text or XML format. The sequence flow box on the right of Fig.19 represents the outcome.



```
BPMN=A.bpmn
sequences=enable
```

```
normal sequence: 1
A, B, C, D
loop sequence: 1
A, E, F, E
```

Fig. 19. The structure of sequence generator.

5.2 Model–Checking Based Sequence Generation

As we know, PROMELA code can be used as input language to SPIN for detecting errors. It also can be used for generating message sequences as well [24]. In order to obtain a test sequence, a designer needs a validated PROMELA model.

Second, trap properties have to be defined and be included into this model. A trap property is actually an LTL included into a PROMELA model to produce the execution test sequences automatically. Sequences appear as counter-examples produced by SPIN in response to the examination at the PROMELA model against the LTL trap properties. Fig. 20 briefly expresses the idea of how to produce counter-examples by working with trap properties, it represents the work that has been done so far. The meaning of "correctness properties in LTL" and "PROMELA model" has been explained in Fig. 11. Here trap properties are included into PROMELA model which has been checked previously by using correctness properties. SPIN produces counter-examples that contains information from where the sequences can be extracted after some filtering of irrelevant information.

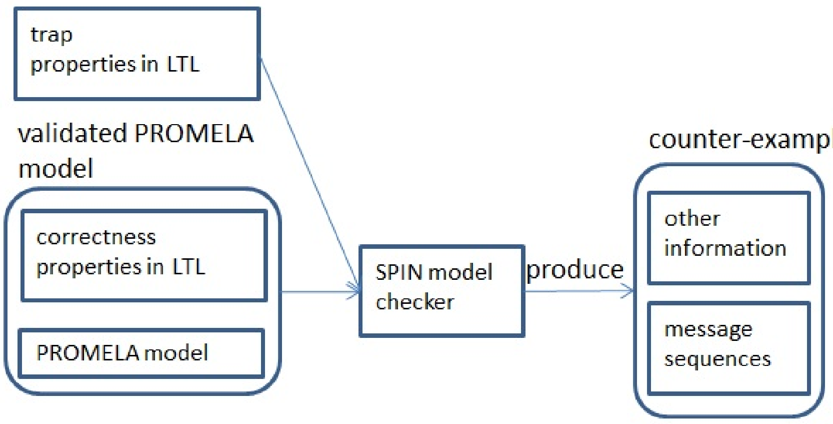


Fig. 20. Generating execution sequences by employing trap properties.

Compare with the symbolic execution based sequence generation, Model–Checking Based Sequence Generation has an advantage: these message sequences are produced from a validated PROMELA model (in symbolic execution based sequence generation, sequences are produced from a BPMN choreography that might be flawed). This is a very important property for the reliability of the sequences. This approach has been used also in [25].

6 Testing and Evaluation

This section describes the testing of the BPMN2PROMELA translator. It includes the testing of PROMELA model generation, LTL formula and definition generation, message sequences generation and basic functionalities. Outcomes after testing are compared with the predefined expected outcome. After that, we will evaluate aim, objectives and the translator.

6.1 Overall Testing Strategy

Generally speaking, this project is done by Test-Driven Development (TDD), TDD is a type of software development which employs the concepts of test-first programming of extreme programming [26]. It especially suits for those short term, small or medium-sized projects. During the development, developers repeat a short development cycle [27]. First, in a test-driven development (e.g. this project), writing a test before the implementation of a new feature is essential. For example in this project, I begin with a very simple BPMN choreography which has only one activity. Then test this case, we found that it fails because we have not implemented it yet. Second, write some code to implement the new feature. The aim of the code is to pass the test. If it is successful, write another test like the step at very beginning. For instance, this time the new feature is exclusive gateway, I repaint the BPMN choreography: add a gateway after the activity. Then implement it and run all tests and see whether the new one fails. Repeat the work until we meet all requirements. It is very important that new code should not damage any existing functionality which has passed the test. Fig. 21 illustrates the testing strategy of this project. It is worth mentioning that the final outcomes (PROMELA models) should also pass the SPIN syntax checker. Currently, we perform this checking work manually. But in the future, it should be integrated into the whole testing structure and be tested mechanically.

6.2 Testing Performed

Three forms of testing were performed:

- Unit Testing
  JUnit [28] is a easy to use unit testing framework for Java. It has been integrated in Eclipse. We use it to test basic functionalities such as loading the configuration file, check properties in BPMN diagram like valid path size, loop path size, messages, tasks. Basically, it follows the principle of TDD.
- Integration Testing
  The aim of integration testing is to detect errors between integrated components [29]. Basically, parsing and translating stages are developed independently. This test in the project aims to see if a BPMN element can be transformed into correct PROMELA syntax without losing meaning.
- System Testing
  We followed some basic rules of system testing mentioned in [30]. In our dissertation as we mentioned, PROMELA models which generated by the translator should pass the SPIN syntax checker. The goal of this dissertation is to create PROMELA models with valid syntax. Therefore, for each PROMELA model, we should check it within SPIN. The version of SPIN we used in the project is 4.3.0. This aims to test whether the final system meets all original requirements.

The choice of test cases and the test result can be found in Appendix C.

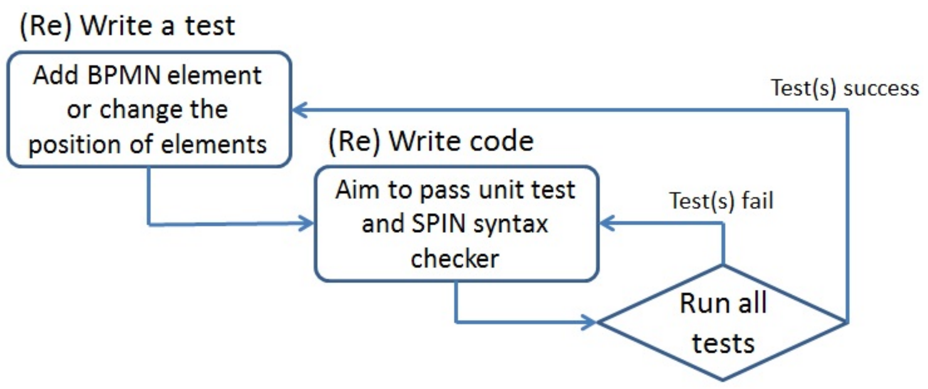

(Re) Write a test
Add BPMN element or change the position of elements
Test(s) success
(Re) Write code
Aim to pass unit test and SPIN syntax checker
Test(s) fail
Run all tests

Fig. 21. Test strategy in this project.

6.3 Evaluation Against Aim and Objectives

In this section, we will evaluate the proposed aims and objectives which have been set out in Introduction. Following this we would like to evaluate the translator which is the main contribution of this dissertation.

Aim Evaluation The primary aim of the dissertation is to design and implement a tool that translates BPMN choreographies into PROMELA models. The translator consumes BPMN choreography diagrams which is generated by using SAVARA BPMN diagram editor and produces corresponding executable PROMELA code.

Objectives Evaluation The first objective we set out to achieve was to do some research work and get a clear idea about why we need the translator. For this point, we employ Jboss Savara Testable Architecture to explain the procedure of the development of distributed systems. Here we mainly focus on the design level. Now we have known that the translator aims to bridge the gap between BPMN choreography and SPIN model checker.

The second objective was to investigate related fields that associated with this translator. In this project the translator is responsible for the translating from a high level language (BPMN that represented in XML) into another high level language (PROMELA). Therefore, before coding, have a clear idea of BPMN constructs, SPIN model checker, PROMELA syntax and LTL formula is essential. Moreover, mapping BPMN elements into PROMELA syntax is the key of the translator.

The third objective was to design and implement the translator. It is the primary objective of this project. We will discuss it thoroughly in next part.

Next objective was to generate message sequences from BPMN choreographies automatically. Currently, this part has been merged into BPMN2PROMELA translator as a sub-function. Two types of files can be generated after execution for the purpose of flexibility.

The final objective was to validate PROMELA models which generated by the translator in SPIN and analyse results of typical scenarios. In this dissertation, we take some example to briefly explain how the translator helps us.

6.4 Evaluation of BPMN2PROMELA Translator

As stated before, this is still an on-going project. Some features have not been added into it. the following lists shows the work that has been done and some future work. Currently, following functionalities has been implemented in the translator:

- Supporting a set of BPMN elements such as events, activities, messages, exclusive gateways and sequence flows which can be translated into PROMELA syntax like variable definitions, message definitions, channel definitions, process definitions, if blocks, atomic blocks, init block, message activities, run statements, GOTO statements, label points.
- It supports configuration of buffer size and message type.
- It supports simple loop back paths from activity to activity or from split exclusive gateways to activity.
- It supports choreographed and orchestrated translation of PROMELA models.
- It supports two types of initializing processes (init and active).
- It supports LTL generation (it supports a single LTL formula in current version).
- It supports message sequences generation from BPMN choreographies into two types (plain text and XML format).

Some future work that can be done is listed below:

- A GUI is needed for users.
- Supporting more elements in BPMN standard.
- Supporting multiple LTL formula definition and configuration.
- Supporting complex loop back paths in BPMN diagrams.
- Testing PROMELA model in SPIN automatically. Currently, syntax checking work is done manually. In the future, we want to integrate this work into this project.
- A static checker for BPMN diagrams.

7 Related Work

A lot of work has been done to encode BPMN choreographies into another form. The results has been presented in paper [31]. In their work, they primarily focus on realizability issue with choreographies. So a realizability checking apporach for BPMN 2.0 choreographies has been proposed. Then in [31], BPMN

2.0 choreographies is translated into LOTOS NT, which ”is an improved version of the LOTOS ISO standard that combines the best features of imperative programming languages and value-passing process algebras”. They have listed several reasons about why LOTOS NT was chosen to be the target language: first, there are some similar points between BPMN constructs and LOTOS NT syntax such as choice, sequence, etc. Second, LOTOS NT can be analysed by state-of-the-art verification toolbox (CADP). They consider both synchronous and asynchronous communication when discussing realizability of BPMN choreography. As what we did, they also focus on a significative subset of BPMN elements. A counter-example will be generated if realizability cannot be ensured and it helps designer in their modelling work.

In [32], in order to solve the communication problems between developers and domain experts. They present a user friendly ECLIPSE plug-in for the verification of requirements over a Business Process model. Standard BPMN notation is used for the specification of Business Process in the tool. This tool implements a converting from BPMN specification to the CSP formal language and supports CSP verification via model checking. Their work can be described into three steps: first, mapping BPMN model into CSP model. Second, set up goals according to the quality requirements. Third, verification of CSP model with these goals. Then analyse counter-example if necessary. The idea behind it is similar as the relationship between PROMELA model, LTL formulas and SPIN model checker in our dissertation. The goal of their work is to enable designers to do verification work with graphical notation that they are already familiar with.

A tool has been implemented in [33] that is used for the translation from XML serialization of BPMN models to the Petri Net Markup Language (PNML) for static analysis. This paper focuses on the control-flow perspective of BPMN. They also mentioned how to use Petri net analysis toolset for analysing. Basically, they did the same job as what we did but translated BPMN into petriNet. For example, they have a mapping table from BPMN Object to Petri-net Module, just like the BPMN element to PROMELA syntax mapping table in this dissertation. Also, they have some issues which have not been addressed with the limitation of Petri nets: first, parallel activities, which is a problem in our project too. Second, exception handling of sub-processes. Third, complex gateways. From the point of view of our design, we considered that PROMELA and SPIN are a more mature model checking tool than petri nets and better documented. We do not know what properties are better expressed in petri net than in PROMELA and the other way around. It could be one of our future works.

The translation from BPMN to Event-B has been presented in [33] in order to improve the quality of business process models within a software design process by using formal methods. They also talk about the impact of the analysis results on software design by examining their latest work by using Event-B and Rodin. They focus on readability, provability and analyzability of the translation. They also have a mapping table like what we did. The translation is done automated and covers a large set of BPMN constructs. Moreover, both control flow and data flow have been considered.

However, few works focused on the encoding of BPMN choreographies into PROMELA models. [34] has the similar idea with this paper. In [34], they introduce a verification procedure which has an automatic translation algorithm for producing readable PROMELA code from BPMN specification. However, this translation algorithm remains in the theoretical stage, they have not published the translator yet. In this dissertation, we have a preliminary implementation of a BPMN2PROMELA convertor that can mechanically produce PROMELA code from SAVARA BPMN choreography diagrams.

8 Conclusions and Future Work

In this dissertation, we have discussed why we need a BPMN to PROMELA translator. We investigated related concepts such as BPMN choreography, SPIN, PROMELA and LTL. Then we design and implement a BPMN2PROMELA translator by using Java. After that, some examples were given and analysed to explain how the translator can be used at design stage. Next, we discussed sequence generation of BPMN choreographies. Finally, we evaluated what have been done in this project. Basically, the main contribution of this dissertation is the implementation of the BPMN2PROMELA translator. It acts as an bridge between BPMN choreographies and SPIN model checker. However, it is a on going project. New features needed to be added to complete it in the future.

As we mentioned before, counter-examples have been produced. However, it is not enough for the process of sequence generation. Because counter-example is a plain textual file with some excess information. It cannot be processed mechanically and be used as scenarios in the SAVARA platform. To address this limitation we describe here some future work that can be undertaken in the future (see Fig. 22). The first step is to create a filter which can help us filter out information from counter-examples. The second step is to design and implement a tool which can be used to transform sequences from textual form to XML format. These XML files then can be used as scenarios for automated processing. Surely before that, we should define XML tags for this domain and predefine them in a XML schema. The tool is named "Sequence2XML" in the figure.

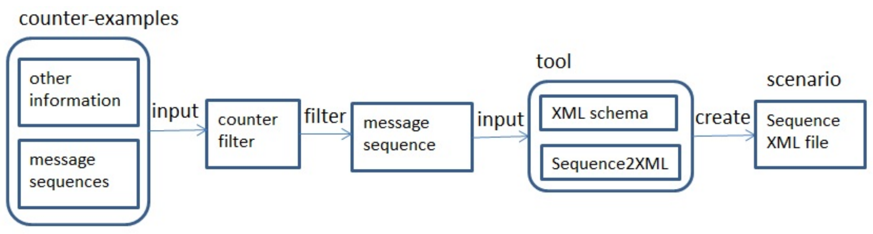


Fig. 22. Future work on model-checking based sequence generation.

Another potential enhancement of this work would be the automatic inclusion of LTL properties into PROMELA code. We know that now PROMELA model can be generated automatically. However, correctness properties in LTL still have to be defined manually. Include them mechanically is highly desirable. Fig. 23 shows how to include LTL into PROMELA model currently. Basically, there are four steps.

1. *The designer think of a correct property in English such as* message A is always preceded by message B.
2. *Express it in LTL notation such as* $!(!a \bigcup b)$.
3. *Convert the LTL into PROMELA code (within a never block) using the automatic converter provided by SPIN.*
4. *Copy and paste the resulting PROMELA code of the LTL into PROMELA model.*
5. *Run SPIN which will verify the PROMELA model against the LTL.*

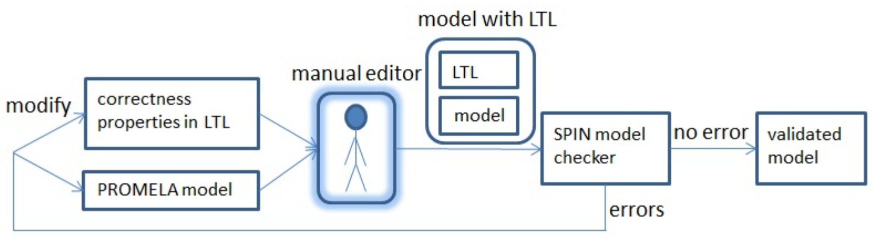


Fig. 23. Future work on the automatic inclusion of LTL.

We can see now, LTL properties have to be added manually that is editing (as shown in the blur square with a human). Integrate model and LTL mechanically means that this step can be accomplished without the interference of human.

As mentioned before, scenarios for choreography can be generated manually by using SAVARA tool. It is feasible for those choreographies that have few paths. However, it is improper or even impossible for designers to create them one by one if the number of paths is large (e.g. more than 10). Therefore, we should have a method to create these scenarios mechanically. As we can see from Fig. 22, sequence files which can be regarded as scenarios will be generated from a PROMELA model with trap properties. They are XML files which have the capability to be processed by machines.

Acknowledgements I would like to thank my project supervisors Dr. Ellis Solaiman and Dr. Carlos Molina-jimenez for the patience, time and effort they have spent on me during the work of this dissertation. This work was partially supported by them. I was entitled to use some figures and segments of SPIN validations and LTL formulas from the research papers of Dr. Carlos Molina-jimenez. This enable me to focus on the main aim of my dissertation: The design, implementation and testing of the BPMN2PROMELA translator. Their guidance helps me bring project to its successful completion.

I would also like to express my gratitude to my family and friends for their continuing support. Their help has been greatly valued and appreciated from start to end during this year.

References

1. OMG: Documents associated with business process model and notation (bpmn) version 2.0 (Jan 2011)
2. Decker, G., Kopp, O., Barros, A.: An introduction to service choreographies. Information Technology **50**(2) (2008) 122–127
3. Schönberger, A.: Do we need a refined choreography notion? In: Proc. 3rd Central–European Workshop on Services and their Composition, Services (ZEUS'11). Volume 705., CEUR-WS.org (2011)
4. RosettaNet: Rosettanet methodology for creating choreographies (27 July 2011 2012) Version Identifier: R11.00.00A.
5. Jboss: Savara and testable architecture (2012)
6. Molina-Jimenez, C., Shrivastava, S., Strano, M.: A model for checking contractual compliance of business interactions. IEEE Trans. on Service Computing **5**(2) (2012) 276–289
7. Molina-Jimenez, C., Shrivastava, S.: Maintaining Consistency between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors. In: Proc. 4th IEEE European Conf. on Web Services (ECOWS'06), IEEE CS (2006) 148–160
8. Peltz, C.: Web services orchestration and choreography. Computer **36**(10) (October 2003) 46–523
9. Coward, P.D.: Symbolic execution and testing. In: Proc. IEE Colloquium on Software Testing for Critical Systems. (1990)
10. Fitzgerald, J.S., Larsen, P.G., Pierce, K.G., Verhoef, M.H.G.: A formal approach to collaborative modelling and co–simulation for embedded systems. Technical Report CS–TR–1264, School of Computing Science, Newcastle University, UK (2011)
11. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall (1991)
12. Zhao, W.: Yet another model checker for promela: – the transformation approach. In: Proc. 4th IEEE Int'l Conf. on Secure Soft. Integration and Reliability Improvement Companion (SSIRI–C'10),. (2010) 137–142
13. Ben-Ari, M.: Principles of the Spin Model Checker. Springer (2008)
14. Bharadwaj, R., Heitmayer, C.L.: Model checking complete requirements specifications using abstraction. Automated Software Engineering **6**(1) (1999) 37–68
15. Staunton, J., Clark, J.A.: Finding short counterexamples in promela models using estimation of distribution algorithms. In: Proc. 13th annual Conf. on Genetic and evolutionary computation (GECCO'11). (2011) 1923–1930
16. University, K.S.: Spec patterns (Aug 2012)
17. Holzmann, G.J.: The Spin model checker: primer and reference manual. Addison–Wesley Professional (2003)

18. Overbey, J.L., Johnson, R.E.: Generating rewritable abstract syntax trees (Oct 2008)
19. Howarth, N.: Abstract syntax tree design. Technical report (1995)
20. Jones, J.: Abstract syntax tree implementation idioms. (2003)
21. Bokowski, B., Spiegel, A.: Barat a front-end for java. Technical report (1998)
22. Copper, J.W.: Java Design Patterns: A Tutorial. 2nd edn. Addison-Wesley (April 2000)
23. Erich Gamma, Richard Helm, R.J., Vlissides, J.: Design patterns: elements of reusable object-oriented software. 37th edn. Addison-Wesley (March 2009)
24. Rayadurgam, S., Heimdahl, M.P.: Test–sequence generation from formal requirement models. In: Proc. of the 6th IEEE Int'l Symposium on High Assurance Systems Engineering (HASE01), IEEE CS (2001) 23–31
25. Abdelsadiq, A., Molina-Jimenez, C., Shrivastava, S.: On model checker based testing of electronic contracting systems. In: 12th IEEE Int'l Conf. on Commerce and Enterprise Computing(CEC'10). (2010) 88–95
26. BENJI KOLTAI, JEFFREY WARNICK, R.A., NILAN, S.: Test-driven development (Dec 2011)
27. George, B., Williams, L.: A structured experiment of test-drivendevelopment. Volume 46. (April 2004)
28. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley (2000)
29. Beizer, B.: Software testing techniques. 2nd edn. Dreamtech (2002)
30. Petschenik, N.H.: Practical priorities in system testing. Technical report, Bell Communications Research (1985)
31. Poizat, P., Sala´un, G.: Checking the realizability of bpmn 2.0 choreographies. In: Proc. 27th Annual ACM Symposium on Applied Computing (SAC'12). (2012) 1927–1934
32. Flavio, C., Alberto, P., Barbara, R., Damiano, F.: An eclipse plug-in for formal verification of bpmn processes. In: Third Int'l Conf. on Communication Theory, Reliability, and Quality of Service (CTRQ'2010). (2010) 144–149
33. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and automated analysis of BPMN process models. available at: http://eprints.qut.edu.au/6859/ (April 2007)
34. Aguilar, J.C.P., Hasebe, K., Mazzara, M., Kato, K.: Model checking bpmn models for reconfigurable workflows. Technical Report CS–TR–1274, School of Computing Science, Newcastle University, UK (2011)

A BPMN2Promela Translator

A.1 Technologies and Tools Used in Implementation

Eclipse Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java. It can be used to develop applications in Java and, by means of various plug-ins, other mainstream programming languages including C, C++, Perl, PHP, Python, Ruby. It is a free tools platform for almost all kinds of developers. As we mentioned before, it has a extensible plug-in system which makes it very easy to be extended. Another important aspect of eclipse is that compilation is executed automatically whenever we save a file. It always builds for users and the procedure is completely transparent for developers. This essentially means the code is always compiled.

The version that is used here is ”Indigo Service Release 1”. It can be downloaded in http://www.eclipse.org/indigo/.

SAVARA Savara is a project established and maintained by JBoss (still in development so far). It gives us a framework that based on the Testable Architecture methodology which can help us building distributed and service oriented systems. It includes several tools to assist the designer of the business application at different stages of his design. It used for design purpose.

Main functionalities of the tool so far:

1 Definition of choreography
2 Creation of scenarios
3 Generation of service (BPEL, BPMN2 Process, SCA Java)
4 Generation of contracts (WSDL)
5 Simulation of scenarios against the choreography

The focus here is the definition of choreography. In this project, we were using SAVARA to define choreography diagrams. The information on how to build a choreography can be found in the SAVARA User Guide.

SAVARA 2.0.x tools is used here. They have been downloaded and deployed to eclipse on Win 7.

Extensible Markup Language(XML) XML is a markup language that defines a set of rules for encoding documents in a format that can be both read by human and executed by machine. It is produced by W3C and follows XML 1.0 Specification currently. Although the design aim of XML is to transport and store data, it is widely used for the representation of data structures. In this case, choreography diagrams that created by using SAVARA are actually XML documents. Therefore, it has the capability to be processed automatically by procedures in any languages on any platform. XML documents may begin by declaring some information about themselves, as in the following example:

<?xml version=”1.0” encoding=”UTF-8”?>

In the declaration, "version" attribute indicates that it is defined in the XML 1.0 Specification and "encoding" attribute means the textual format data in the document is encoded in UTF-8 (It is a variable-width encoding that can represent every character in the Unicode character set).

Understand XML syntax and features is essential for the project. Here we use an example to briefly explain some main features of XML in our research scope. Figure 24 shows how to open the file. Right click the choreography file, mouse over "Open With" menu, when the sub-menu appears at its right, click "Text Editor" button to open it in a textual perspective.

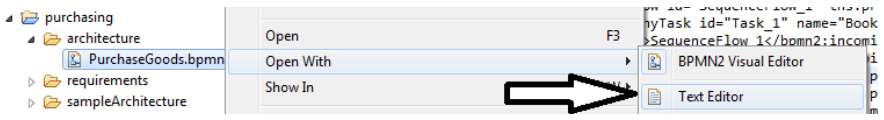
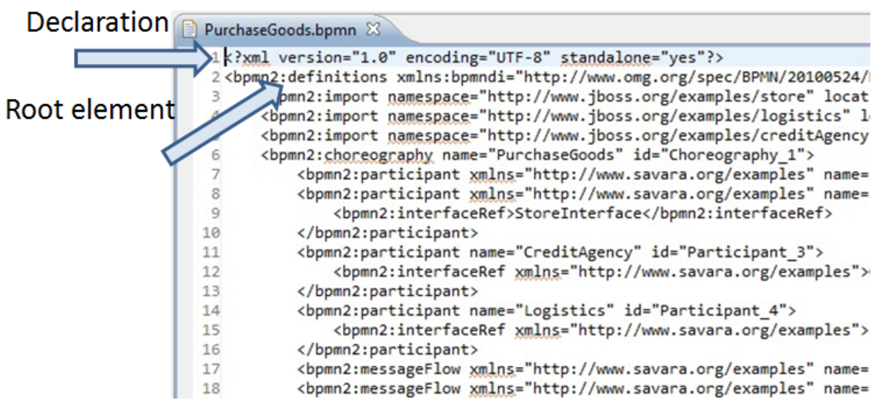


Fig. 24. Open a choreography in text editor.

This is a official example given by SAVARA. The download page is http://downloads.jboss.org/savara/examples/savara2-example-purchasing.zip. The information on how to import this project into eclipse can be found in SAVARA Getting Started Guide. This file is in the "architecture" folder which has been shown in the figure above. Figure 25 shows part of the document.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bpmn2:definitions xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/
     pmn2:import namespace="http://www.jboss.org/examples/store" locat
    <bpmn2:import namespace="http://www.jboss.org/examples/logistics" l
    <bpmn2:import namespace="http://www.jboss.org/examples/creditAgency
    <bpmn2:choreography name="PurchaseGoods" id="Choreography_1">
        <bpmn2:participant xmlns="http://www.savara.org/examples" name=
        <bpmn2:participant xmlns="http://www.savara.org/examples" name=
            <bpmn2:interfaceRef>StoreInterface</bpmn2:interfaceRef>
        </bpmn2:participant>
        <bpmn2:participant name="CreditAgency" id="Participant_3">
            <bpmn2:interfaceRef xmlns="http://www.savara.org/examples">
        </bpmn2:participant>
        <bpmn2:participant name="Logistics" id="Participant_4">
            <bpmn2:interfaceRef xmlns="http://www.savara.org/examples">
        </bpmn2:participant>
        <bpmn2:messageFlow xmlns="http://www.savara.org/examples" name=
        <bpmn2:messageFlow xmlns="http://www.savara.org/examples" name=
```

Fig. 25. The content of choreography file.

As mentioned above, XML has some features and syntax. When we create a valid XML file, we should always follow the predefined syntax. Fortunately, SAVARA choreography diagram generation tool help us creating such documents by simply dragging and dropping icons from the menu to the canvas. XML documents which generated by using this tool apply with following syntax:

1 All XML elements must have a closing tag
2 XML tags are case sensitive
3 XML elements must be properly nested
4 XML documents must have a root element
5 XML attribute values must be quoted

Therefore, the document which is created in this way is a compliant file. This means that no need for us to care about the grammatical structure of a document. We can put our efforts on business requirements and information.

It is important to know that all tags in figure 25 are not defined in any XML standard. They are defined by the creator in a schema file. Because XML emphasize simplicity, generality, and usability. So XML language has no predefined tags. We can invent specific tags in any area. So basically, an XML schema is a description of a certain type of XML document. In our case, all BPMN XML files are constrained by BPMN2.0 schema. This schema can be found in http://www.omg.org/spec/BPMN/2.0/.

Process Meta Language (PROMELA) and SPIN PROMELA is a process verification modeling language for both communication protocols and concurrent programming. The language allows for the dynamic creation of concurrent processes to model (e.g. distributed systems). The goal of the language is to test the logic of concurrent execution processes. It focus on following procedure rules: first, it ignores format of messages, encoding, storage of data, transmission, etc. Second, it specifies a validation model. In PROMELA models, communication via message channels can be defined to be synchronous or asynchronous. PROMELA models can be analysed with the SPIN model checker, to verify that the modeled system produces the desired behavior. PROMELA programs consist of three key elements: processes (global), finite message channels and variables (global or local). We will discuss the syntax of this language thoroughly in next section.

We have mentioned SPIN above, it is a general tool for testing the correctness of distributed software models in a automated way. Models to be verified are described in PROMELA.

Currently, we use Xspin to check the syntax of the PROMELA code. Verification and simulation are also run on this platform. The SPIN version is 4.3.0 (22 Jun 2007) and the Xspin version is 5.2.0 (8 May 2009).

Java Java is a object-oriented, class-based, platform independent programming language. Its design principle is to make application developers WORA (write once, run anywhere). It means that Java code that runs on one platform does not need to be recompiled to run on another. Because Java applications are typically compiled to class file (byte code) that can run on any Java Virtual Machine (JVM) regardless of platform. We have told about eclipse before, eclipse does compilation for developers automatically and transparently. Therefore, develop application in eclipse is a good choose for Java developer.

It is noteworthy that there are plenty of third-party Java Libraries available for developers. It means that we can build our own application on the top of some mature technologies (e.g. in this project, XML parsing techniques). No need for us to develop them again, we just reuse them. Actually, it is one of Java design purposes.

JavaSE-1.6 or higher version can be employed in the project.

In addition, some knowledge of design patterns is required when constructing an expandable application.

Third party Jars Dom4j is an open source, easy to use library for working with XML on the Java platform. It is a highly-performance, high flexible, and memory-efficient implementations of XML framework. So it is used here to help us resolving XML issues, by using this library a standard XML file can be easily parsed and employed within any Java application. The list below is some key features of Dom4j:

1 Dom4j fully supports Java Collection Framework.

2 It has full support for JAXP, SAX, DOM and XSLT standards.

3 As mentioned before, it is a memory-efficient implementations. When dealing with relatively large documents, it produces little memory overhead.

The latest stable version of dom4j, 1.6.1, was released on May 16, 2005. This version is steady and completely satisfies the requirement of this project. Parsing XML by using Dom4j simplifies the hardships of the development of this project(especially, this is an XML-based Java application). dom4j-1.6.1.jar and jaxen-1.1.1.jar are imported into this project.

A.2 Deployment

This section describes the installation procedure for the BPMN2Promela tool.

Preparatory Work SAVARA tools should be downloaded and installed as plug-in into eclipse before deployment. The information on how to install SAVARA into eclipse can be found in the SAVARA Getting Started Guide.

Importing Project into Eclipse BPMN2PromelaV1 is an project which can be used to translate choreography diagram(generated by using SAVARA) into PROMELA executable code. To get the source code, please contact Dr. Ellis Solaiman (ellis.solaiman@newcastle.ac.uk) or Dr. Carlos Molina-jimenez (carlos.molina@newcastle.ac.uk). Once the SAVARA Eclipse Tool distribution has been correctly installed, and the BPMN2PromelaV1.zip file is available, then use the following steps to import the project:

Open eclipse and right click on the blank area under "Package Explorer" view, mouse over "New" menu and click "Java Project" on sub-menu when it pops up.

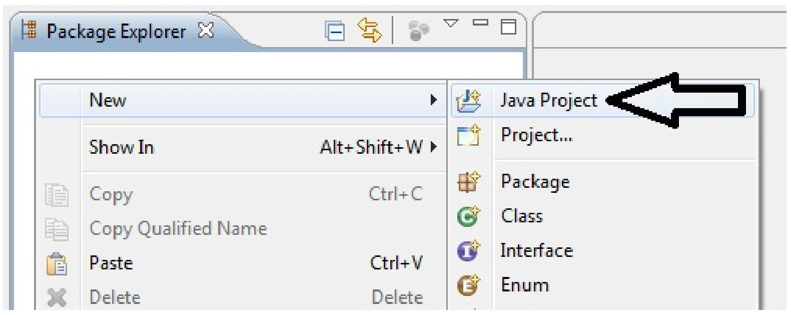
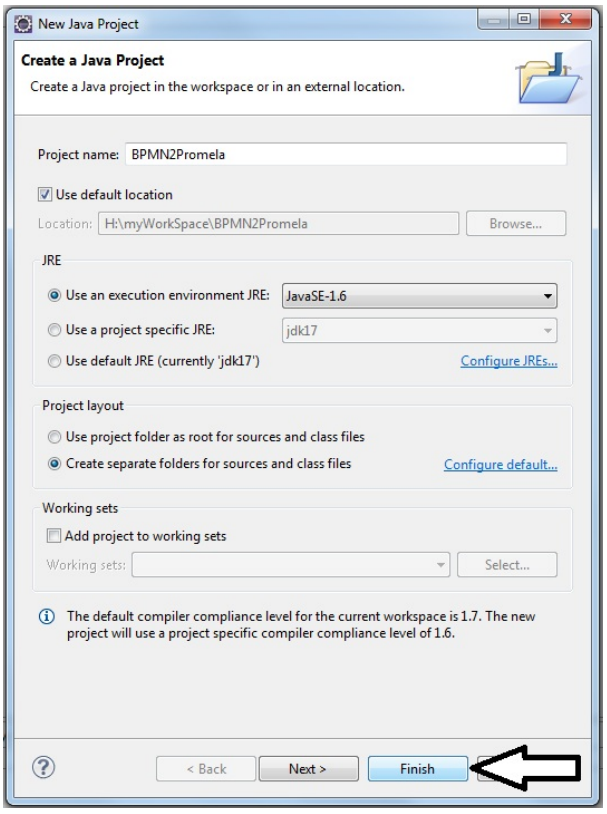


Fig. 26. Create a new java project.

When the "New Java Project" dialog appears, name the project(e.g. BPMN2Promela), then select javaSE-1.6 or higher version to be its execution environment JRE, press "finish" button to create a blank project.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: BPMN2Promela

☑ Use default location

Location: H:\myWorkSpace\BPMN2Promela Browse...

JRE

● Use an execution environment JRE: JavaSE-1.6

○ Use a project specific JRE: jdk17

○ Use default JRE (currently 'jdk17') Configure JREs...

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files Configure default...

Working sets

☐ Add project to working sets

Working sets: Select...

The default compiler compliance level for the current workspace is 1.7. The new project will use a project specific compiler compliance level of 1.6.

< Back Next > Finish

Fig. 27. Name a project and create it.

Right click the blank project, select "import" on the menu.

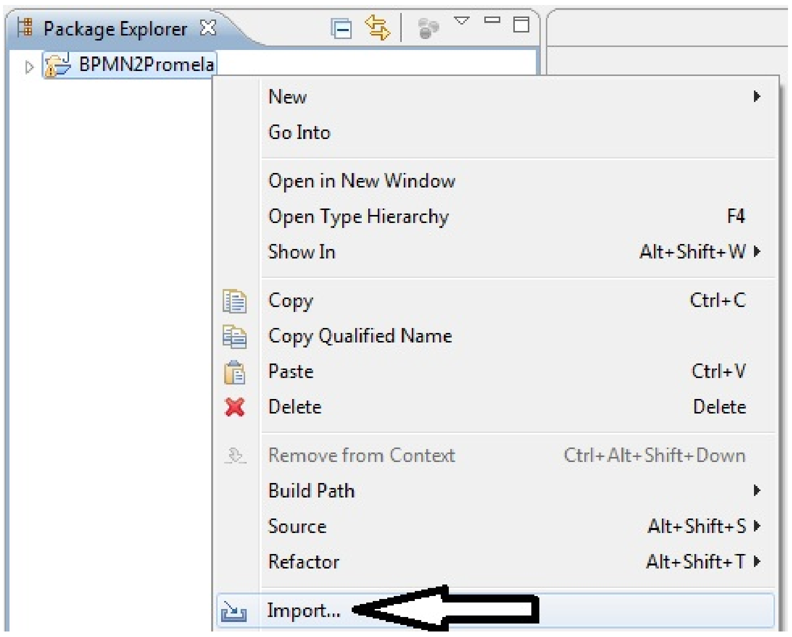


Fig. 28. Import the project.

When the "Import" dialog shows up, select the "General->Archive File" option and press the 'Next' button.

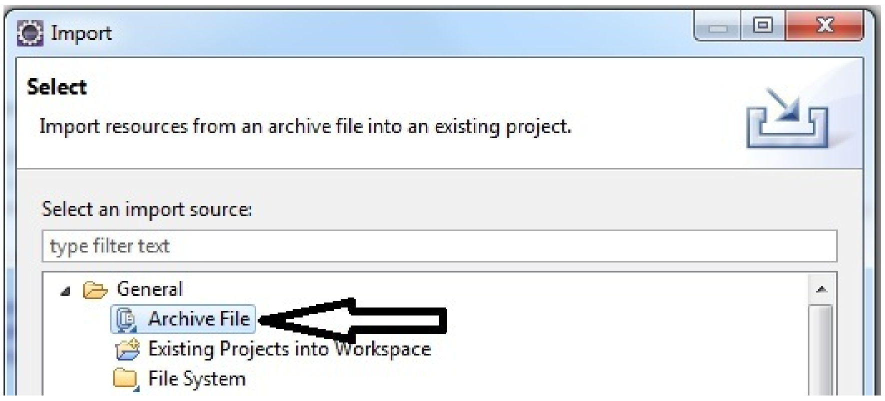

Import
Select
Import resources from an archive file into an existing project.
Select an import source:
type filter text
General
Archive File
Existing Projects into Workspace
File System

Fig. 29. Select archive file and press next.

Press the ’Browse’ button and locate the BPMN2PromelaV1 zip. Press the ’Finish’ button to import the project.

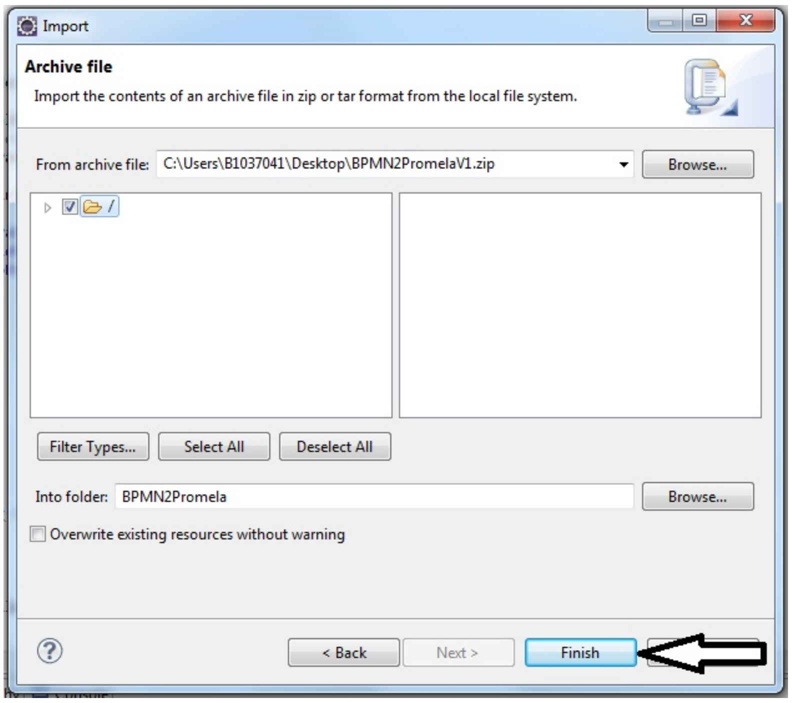

Import
Archive file
Import the contents of an archive file in zip or tar format from the local file system.
From archive file: C:\Users\B1037041\Desktop\BPMN2PromelaV1.zip
Browse...
/
Filter Types...
Select All
Deselect All
Into folder: BPMN2Promela
Browse...
Overwrite existing resources without warning
< Back
Next >
Finish

Fig. 30. Import the zip into the blank project.

Once imported, the Eclipse navigator will list the content of the project. But there are some errors, it is because we have not imported related jar files.

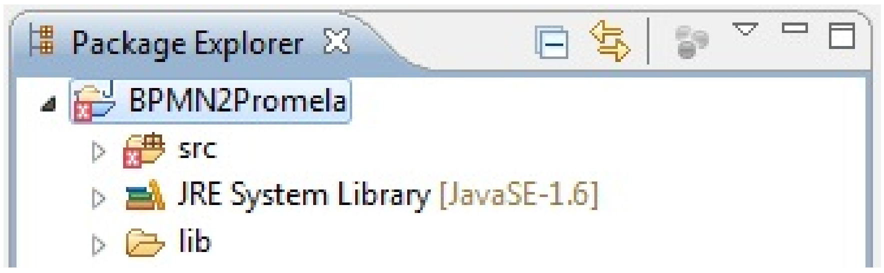


Fig. 31. The content of the project.

Then right click the project, mouse over "Build Path" menu and press "Configure Build Path" on the submenu.

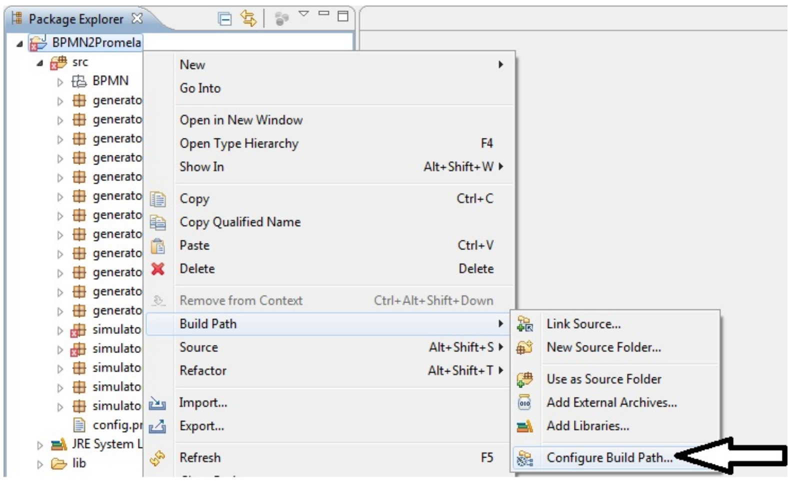


Fig. 32. Build class path for the project.

When the "Properties" dialog appears, press "Add Jars" button to select jar files.

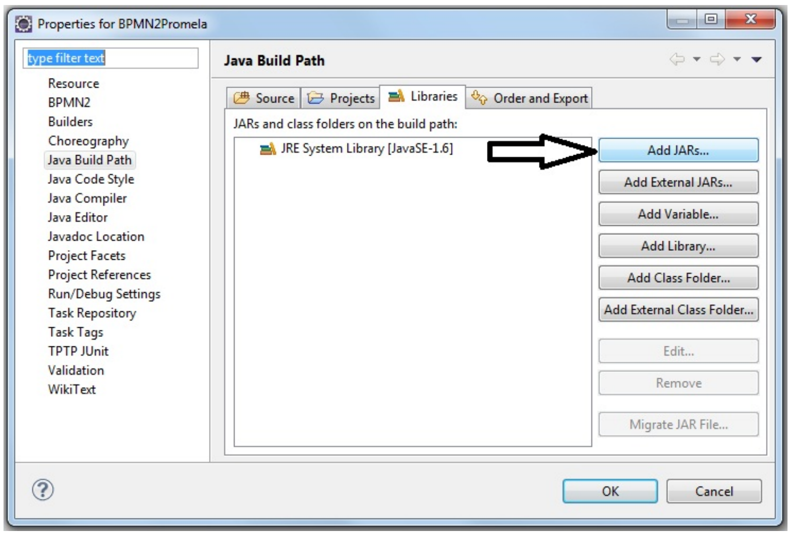


Fig. 33. Add Dependent jars to the project.

In ”Jar Selection” diagram, select all jar files in lib folder and press ”OK” button. Then click ”OK” button on ”Properties” dialog to finish this step.

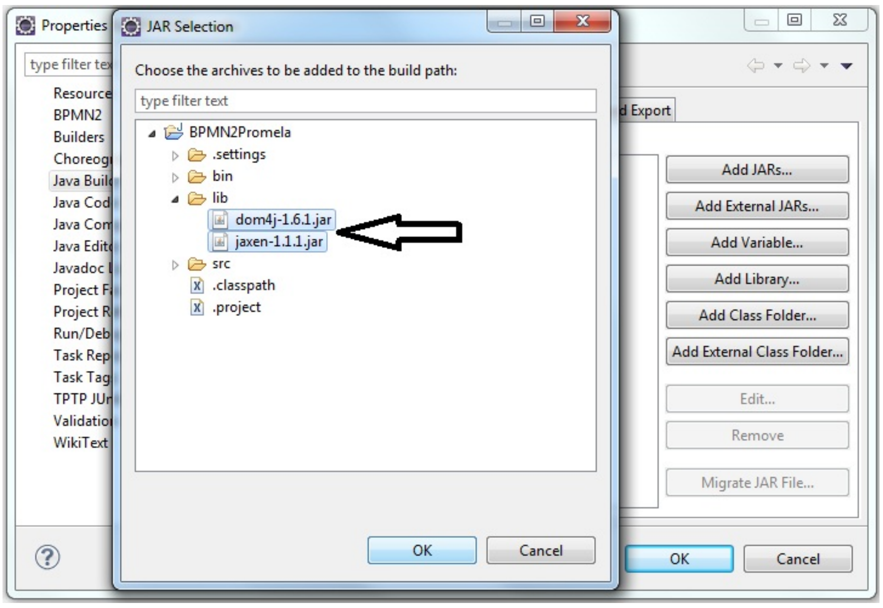


Fig. 34. Select Dependent jars.

”Referenced Libraries” should be added into the project and all errors should be disappeared now. If these errors still exist, right click the project and refresh it.

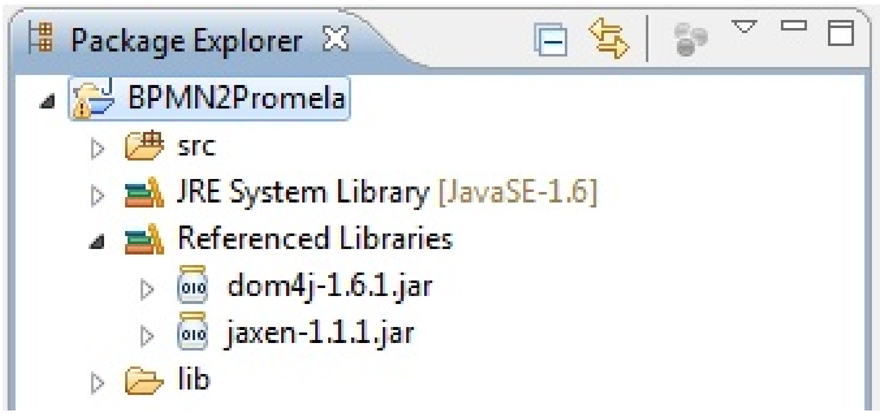
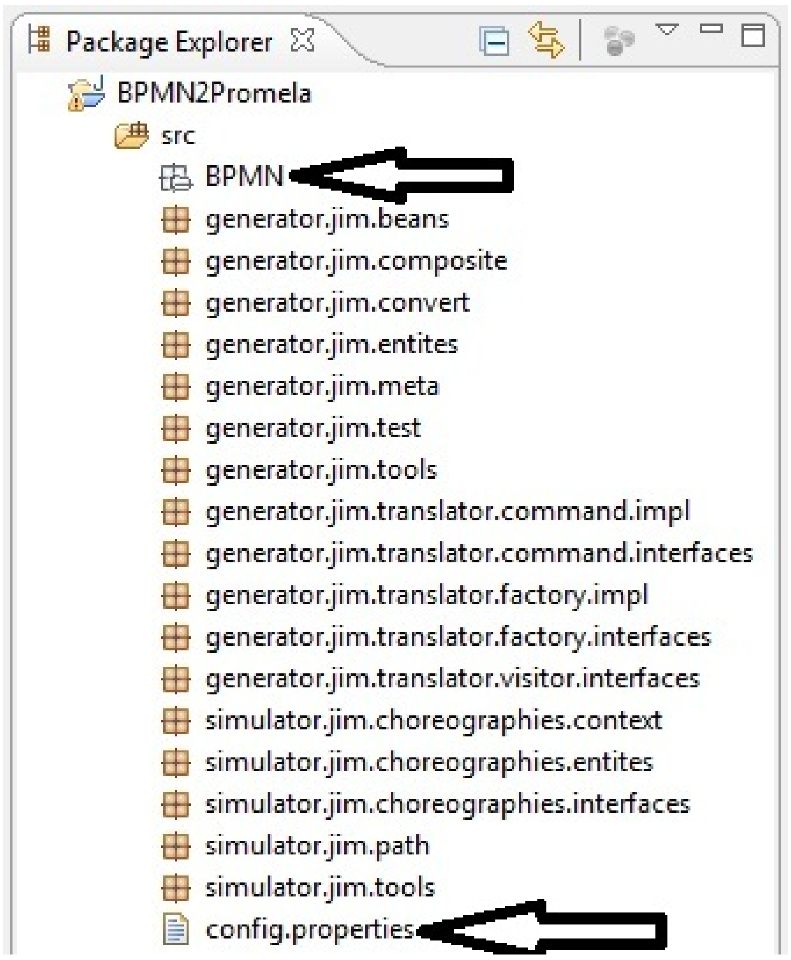


Fig. 35. Project has been imported successfully.

A.3 Configuration and Execution

This section describes how to configure and run the project.

Preparatory Work Make sure that BPMN2PromelaV1 project has been successfully imported into eclipse and all third party jar files is added into class path. Figure 36 shows the content of the project: some packages which contain source code. A BPMN folder which includes all choreography diagrams in it, they are inputs for this program. A config.properties file that controls the behavior of the program.



```
Package Explorer
BPMN2Promela
  src
    BPMN
    generator.jim.beans
    generator.jim.composite
    generator.jim.convert
    generator.jim.entites
    generator.jim.meta
    generator.jim.test
    generator.jim.tools
    generator.jim.translator.command.impl
    generator.jim.translator.command.interfaces
    generator.jim.translator.factory.impl
    generator.jim.translator.factory.interfaces
    generator.jim.translator.visitor.interfaces
    simulator.jim.choreographies.context
    simulator.jim.choreographies.entites
    simulator.jim.choreographies.interfaces
    simulator.jim.path
    simulator.jim.tools
    config.properties
```

Fig. 36. Content of the project.

Configuration Config.properties file is under src folder. Open it, we can see some configuration options.

1 buffer.size: This option defines the size of channel buffer. Its value should be greater or equals than 0.

2 message.type: This option defines the type of predefined messages which can be stored in channels. There are five basic types in PROMELA: bit, bool, byte, short, int. The value should be chosen from one of them.

3 file.location: This option specifies where to store these automatically generated files. e.g. D://promelaFile.

4 file.suffix: It specities the suffix of these PROMELA files.

5 comment.*: Some comment in the content of PROMELA file.

6 tranlator.type: This option specifies the form of generated PROMELA code. Currently, it supports ”decentralized” and ”centralized” forms. We will discuss it in detail in next section.

7 BPMNName: There is a block which named Examples in the file. It lists all choreography diagrams in BPMN folder. All there diagrams are created by SAVARA tool and they are the source of the project. We can choose which diagram will be used by simply marking out it in the configure file.

8 init.type: Let us choose the method of initialization.

9 generation.model: The PROMELA model can be generated optional.

10 generation.sequences: The sequences can be generated optional.

11 generation.sequences.format: We can choose either plain text or XML format of the sequences.

12 generation.LTL: The never block can be generated optional.

13 generation.LTL.formula: The formula we used to verify a PROMELA model.

Execution After configuration is complete, open the main Java file (AutomaticGenerator.java under package generator.jim.test). Right click at the content of it, mouse over ”Run As” menu then press ”Java Application” option to run it. Figure 37 indicates the procedure. After running it, some useful information will be displayed in console and PROMELA code file will be created and stored in the given folder which has been configured in config.properties file.

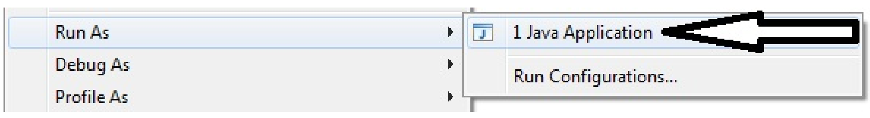


Fig. 37. Execute the project.

Here is a configuration and the corresponding example code. It briefly indicates the relationship between them.

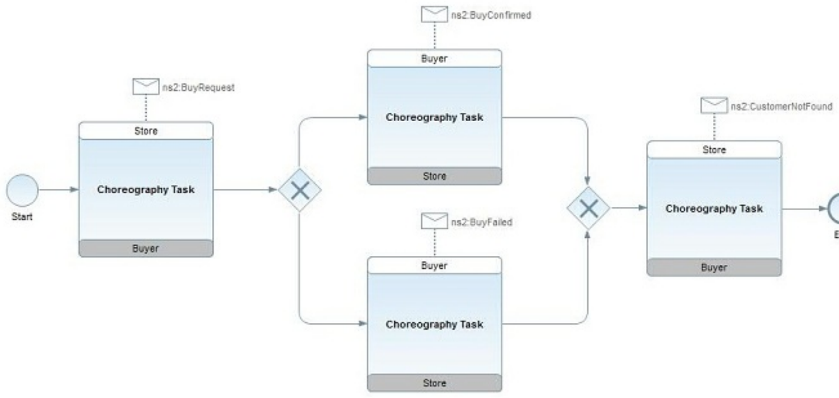
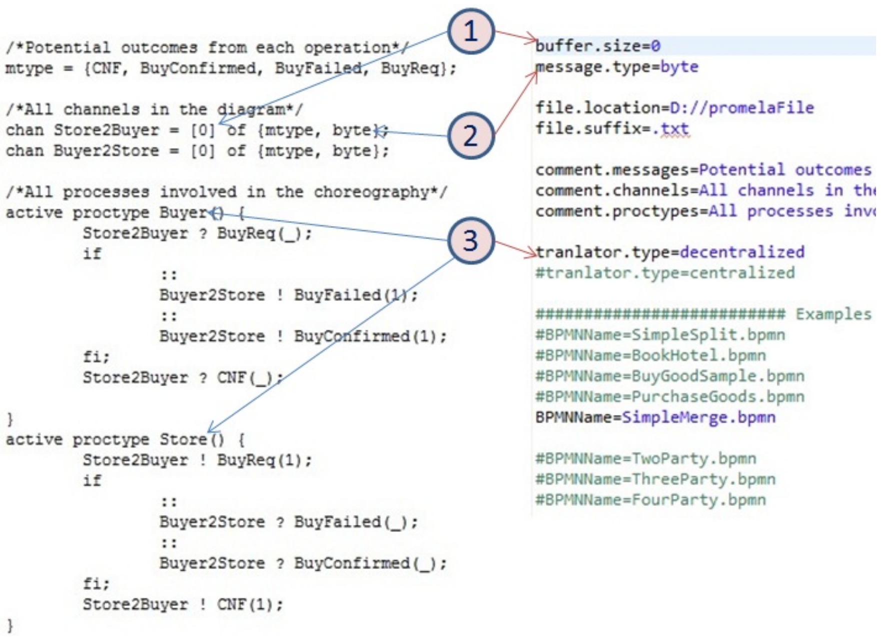


Fig. 38. Diagram of the example.



```
/*Potential outcomes from each operation*/
mtype = {CNF, BuyConfirmed, BuyFailed, BuyReq};

/*All channels in the diagram*/
chan Store2Buyer = [0] of {mtype, byte};
chan Buyer2Store = [0] of {mtype, byte};

/*All processes involved in the choreography*/
active proctype Buyer() {
        Store2Buyer ? BuyReq(_);
        if
                ::
                Buyer2Store ! BuyFailed(1);
                ::
                Buyer2Store ! BuyConfirmed(1);
        fi;
        Store2Buyer ? CNF(_);

}
active proctype Store() {
        Store2Buyer ! BuyReq(1);
        if
                ::
                Buyer2Store ? BuyFailed(_);
                ::
                Buyer2Store ? BuyConfirmed(_);
        fi;
        Store2Buyer ! CNF(1);
}
```

```
buffer.size=0
message.type=byte

file.location=D://promelaFile
file.suffix=.txt

comment.messages=Potential outcomes
comment.channels=All channels in th
comment.proctypes=All processes inv

tranlator.type=decentralized
#tranlator.type=centralized

########################## Examples
#BPMNName=SimpleSplit.bpmn
#BPMNName=BookHotel.bpmn
#BPMNName=BuyGoodSample.bpmn
#BPMNName=PurchaseGoods.bpmn
BPMNName=SimpleMerge.bpmn

#BPMNName=TwoParty.bpmn
#BPMNName=ThreeParty.bpmn
#BPMNName=FourParty.bpmn
```

Fig. 39. Configuration and corresponding code.

A.4 Structure and Restriction of the Project

This section briefly pictures the structure of the project and describes how to modify the code so that it can meet new requirements.

Structure of the Project The structure of the project can be divided into three parts.

1 BPMN elements model part.

In the part, we use following diagram (Figure 40) to help us understand some key concepts.

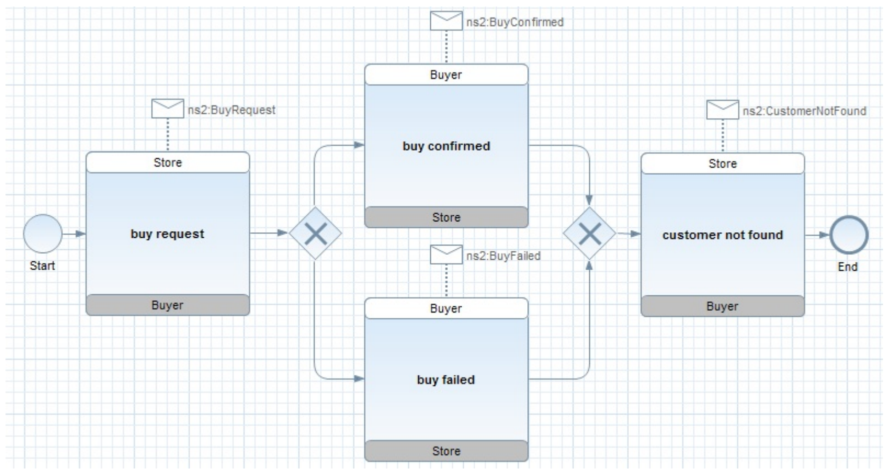


Fig. 40. An example to explain the structure of the project.

Key interfaces:

InterfaceTranslatable: Elements in choreography diagram that implement this interface can be translated into some certain PROMELA code.

Examples of translatable elements in Figure 40 are all events, tasks and gateways.

InterfaceFlowNode: This interface represents those elements which can be connected by sequence flow.

Examples of elements which can be connected in Figure 40 are all events, tasks and gateways.

SequenceFlow: The connector between elements.

Examples of sequence flow in Figure 40 are those strings with a arrow linking between events, tasks or gateways.

InterfaceLogicalParent: Logically, a element can be regarded as the parent of other elements.

InterfaceLogicalChild: Logically, a element can be regarded as a child of other elements.

For example in Figure 40, "buy request" task is a child of "start" event. Similarly, "buy confirmed" task is a child of the split gateway. The split gateway is a child of "start" event and at the same time, it is the parent of "buy confirmed" and "buy failed" tasks. It is noteworthy that "customer not found" task after the merged gateway is a child of ”start” event. In the process of identifying logical parent, some data structure can be used (e.g. stack). The situation becomes more complicated when a choreography contains loop paths in it. Figure 41 shows this kind of situation, when ”buy request” task acts as a member of normal path in this case ([start, buy request, split gateway, buy confirmed, end]), its parent is ”start” event. However, if it appears in loop path instead ([start, buy request, split gateway, buy failed, buy request]), its parent, at this time, is the split gateway. It is very important for us to get the right parent of an element. Because when we generate the code, we have to know where the statement or syntax block should be placed.

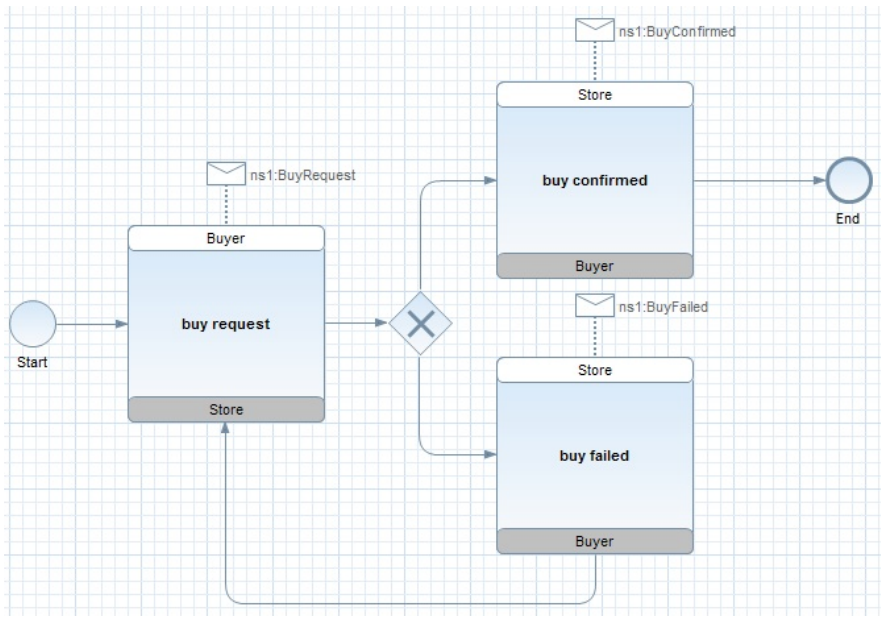


Fig. 41. Choreography with loop path from a Choreography Task to another Choreography Task.

Figure 42 shows the main structure of BPMN elements. InterfaceTranslatable stands at the top of the hierarchy, InterfaceFlowNode extends it. Because according to the requirement, BPMN element who implements InterfaceFlowNode should be translated into some certain PROMELA code.

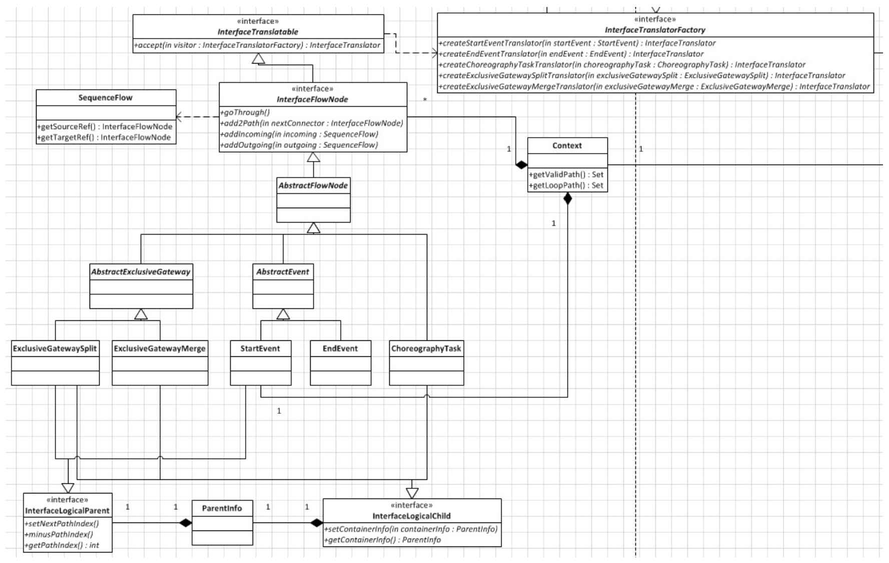

«interface»
InterfaceTranslatable
+accept(in visitor : InterfaceTranslatorFactory) : InterfaceTranslator
«interface»
InterfaceTranslatorFactory
+createStartEventTranslator(in startEvent : StartEvent) : InterfaceTranslator
+createEndEventTranslator(in endEvent : EndEvent) : InterfaceTranslator
+createChoreographyTaskTranslator(in choreographyTask : ChoreographyTask) : InterfaceTranslator
+createExclusiveGatewaySplitTranslator(in exclusiveGatewaySplit : ExclusiveGatewaySplit) : InterfaceTranslator
+createExclusiveGatewayMergeTranslator(in exclusiveGatewayMerge : ExclusiveGatewayMerge) : InterfaceTranslator
SequenceFlow
+getSourceRef() : InterfaceFlowNode
+getTargetRef() : InterfaceFlowNode
«interface»
InterfaceFlowNode
+goThrough()
+add2Path(in nextConnector : InterfaceFlowNode)
+addIncoming(in incoming : SequenceFlow)
+addOutgoing(in outgoing : SequenceFlow)
*
1
Context
+getValidPath() : Set
+getLoopPath() : Set
1
1
AbstractFlowNode
AbstractExclusiveGateway
AbstractEvent
ExclusiveGatewaySplit
ExclusiveGatewayMerge
StartEvent
EndEvent
ChoreographyTask
1
«interface»
InterfaceLogicalParent
+setNextPathIndex()
+minusPathIndex()
+getPathIndex() : int
1
1
ParentInfo
1
1
«interface»
InterfaceLogicalChild
+setContainerInfo(in containerInfo : ParentInfo)
+getContainerInfo() : ParentInfo

Fig. 42. The structure of transforming XML file into BPMN elements in Java objects.

2 Translator generation part.

Key interfaces:

InterfaceConvertor: Class who implements this interface has the ability to translate a given context into a piece of PROMELA code.

InterfaceTranslatorFactory: It defines a series of functions which can create translators for all kinds of BPMN elements.

InterfaceTranslator: It is generated by InterfaceTranslatorFactory. Class who implements this interface can translate a BPMN element in diagram into its corresponding PROMELA code and insert the code fragment into the right place of the document.

Figure 43 shows the main structure of translator factory and its products – translators for BPMN elements. InterfaceConvertor has a InterfaceTranslator-Factory and the converter uses this factory to create translator for each BPMN element.

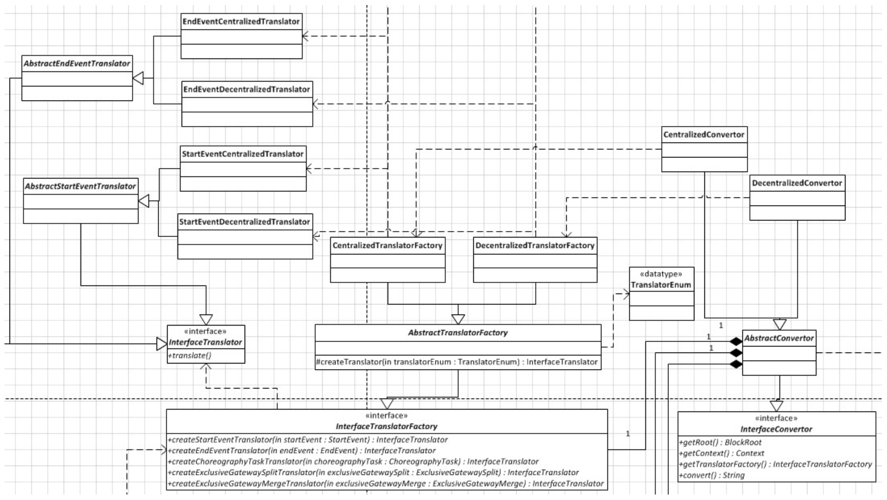
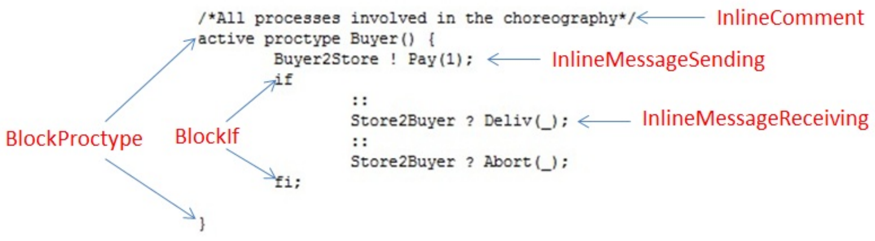

EndEventCentralizedTranslator
AbstractEndEventTranslator
EndEventDecentralizedTranslator
StartEventCentralizedTranslator
AbstractStartEventTranslator
StartEventDecentralizedTranslator
CentralizedConvertor
DecentralizedConvertor
CentralizedTranslatorFactory
DecentralizedTranslatorFactory
«datatype» TranslatorEnum
«interface» InterfaceTranslator
+translate()
AbstractTranslatorFactory
#createTranslator(in translatorEnum : TranslatorEnum) : InterfaceTranslator
AbstractConvertor
«interface» InterfaceTranslatorFactory
+createStartEventTranslator(in startEvent : StartEvent) : InterfaceTranslator
+createEndEventTranslator(in endEvent : EndEvent) : InterfaceTranslator
+createChoreographyTaskTranslator(in choreographyTask : ChoreographyTask) : InterfaceTranslator
+createExclusiveGatewaySplitTranslator(in exclusiveGatewaySplit : ExclusiveGatewaySplit) : InterfaceTranslator
+createExclusiveGatewayMergeTranslator(in exclusiveGatewayMerge : ExclusiveGatewayMerge) : InterfaceTranslator
«interface» InterfaceConvertor
+getRoot() : BlockRoot
+getContext() : Context
+getTranslatorFactory() : InterfaceTranslatorFactory
+convert() : String

Fig. 43. The structure of translator factory for BPMN elements.

3 PROMELA syntax model part

In the part, we use following code fragment (Figure 44) to help us understand some key concepts.



```
/*All processes involved in the choreography*/
active proctype Buyer() {
          Buyer2Store ! Pay(1);
          if
                    ::
                    Store2Buyer ? Deliv(_);
                    ::
                    Store2Buyer ? Abort(_);
          fi;
}
```

Fig. 44. An example to explain the structure of PROMELA syntax model.

Key interfaces:

InterfaceComponent: It defines all behaviours for PROMELA syntax objects.

All statements and blocks in Figure 44 are regarded as components in a file.

AbstractBlock: A block element contains other syntax component.

Examples of block elements in Figure 44 are components like BlockIf or BlockProctype which have been marked in the picture.

AbstractInline: A inline element can only be contained into a block element.

Examples of inline elements in Figure 44 are components like InlineComment, InlineMessageSending or InlineMessageReceiving.

Figure 45 shows the relationship between PROMELA syntax objects. Basically, the whole content of PROMELA file is a syntax tree. we will explain it in next section.

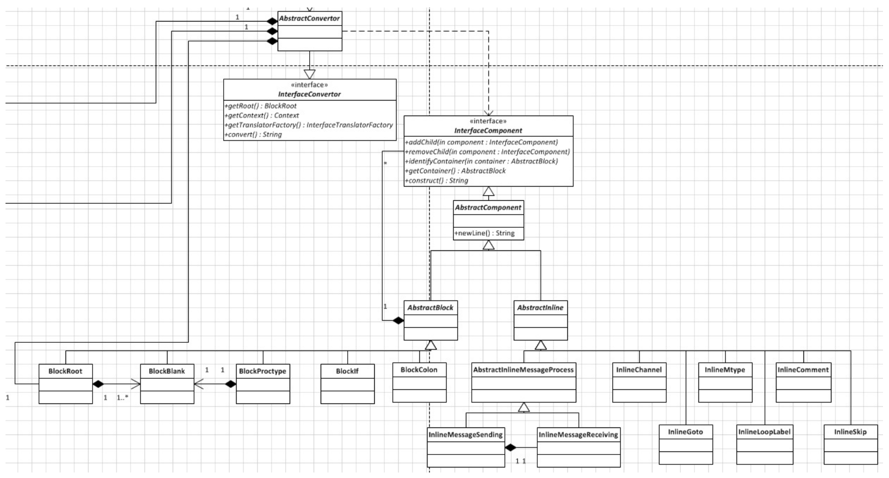


Fig. 45. The structure of PROMELA syntax classes.

Restriction of the Project However, currently, there are some restriction when generating PROMELA code by using this application.

1 The supportive type of PBMN elements: Start Event, End Event, Exclusive Gateway, Choreography Task, Sequence Flow and Message are supported presently. Figure 40 can give us a clear idea of this, it includes all valid elements for this version. We will discuss how to expand it in next section.

2 Loop back path in diagram: Looping back is very common and useful in choreography. Therefore, support for this feature is necessary. Basically, there are two types of Looping back are supported in this version. Loop back from a Choreography Task to another Choreography Task. Figure 41 shows this kind of situation. And another one is from an Split Exclusive Gateway to a Choreography Task. Figure 46 is an example for this situation. Additionally, this version supports one loop back. It means that for each Choreography Task there can exist only one loop back path. It should be modified to support multi loop path in the future.

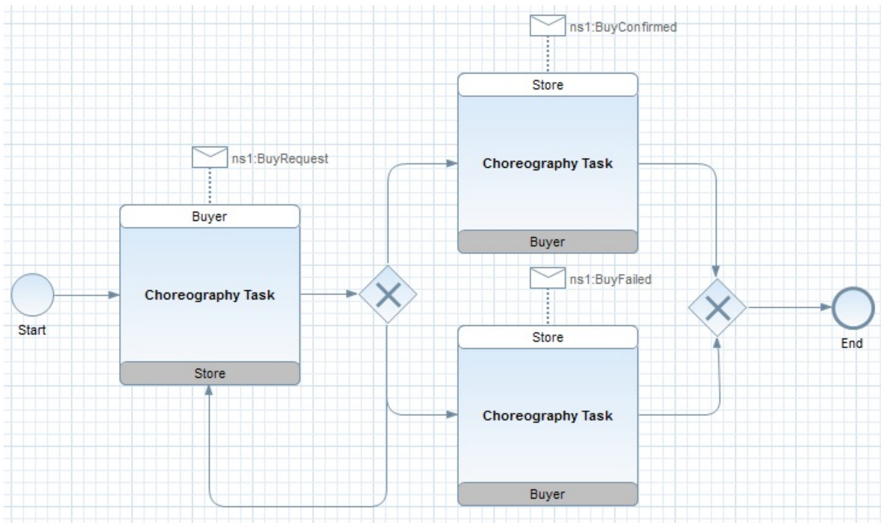


Fig. 46. Choreography with loop back from an Exclusive Gateway to a Choreography Task.

3 Gateway pair problem: Choreography diagrams in figure 47 and figure 48 are trying to express the same meaning. However, the matching of gateways is not corrent in figure 47. Choreography can be processed correctly only if the gateways are matched properly.

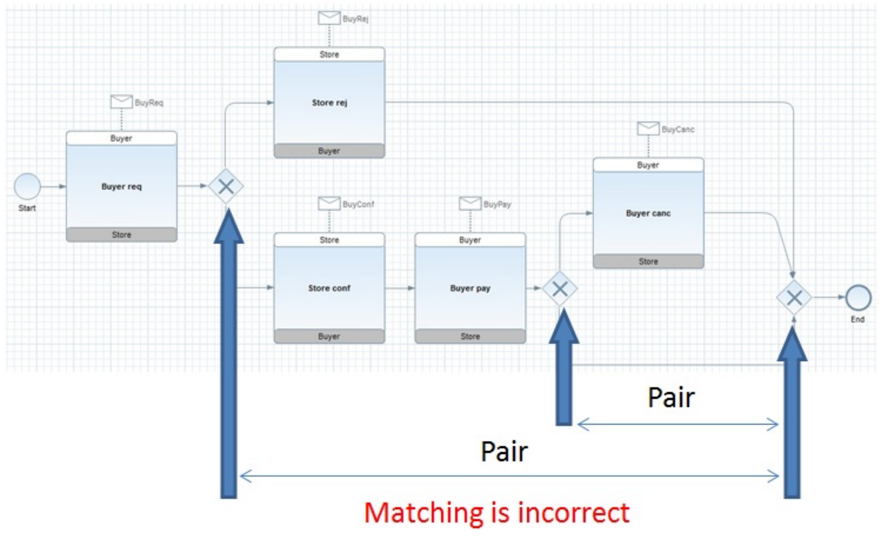


Fig. 47. Incorrect matching of gateways.

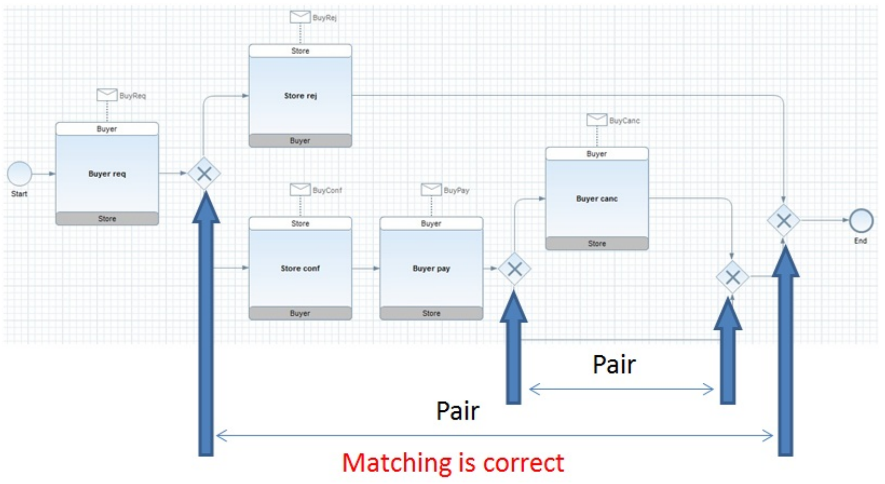


Fig. 48. Correct matching of gateways.

B Code examples

B.1 Code 1

```
#define TRUE  1
#define FALSE 0

#define c (conf==TRUE)
#define r (rej==TRUE)
#define p (pay==TRUE)
#define n (canc==TRUE)

mtype = {BuyConf, BuyPay, BuyRej, BuyCanc, BuyReq};

chan Buyer2Store = [0] of {mtype, byte};
chan Store2Buyer = [0] of {mtype, byte};

bool conf= FALSE;
bool rej= FALSE;
bool pay= FALSE;
bool canc= FALSE;

active proctype Buyer() {
Buyer2Store ! BuyReq(1);
if
:: atomic{Store2Buyer ? BuyConf(_); conf=TRUE};
if
:: Buyer2Store ! BuyPay(1);
:: Buyer2Store ! BuyCanc(1);
fi;
:: atomic{Store2Buyer ? BuyRej(_); rej=TRUE};
fi;
}

active proctype Store() {
Buyer2Store ? BuyReq(_);
if
:: Store2Buyer ! BuyConf(1);
if
:: atomic{Buyer2Store ? BuyPay(_);  pay=TRUE}
:: atomic{Buyer2Store ? BuyCanc(_); canc=TRUE}
fi;
:: Store2Buyer ! BuyRej(1);
fi;
}

never { /* !([](c -><>((p && !n) || (n && !p) ) )) */
T0_init:
        if
        :: (((! ((n)) && ! ((p)) && (c)) || ((c) && (n) && (p)))) ->
            goto accept_S35
        :: (1) -> goto T0_init
        fi;
accept_S35:
        if
        :: (((! ((n)) && ! ((p))) || ((n) && (p)))) -> goto accept_S35
        fi;
}
```

B.2 Code 2

```
mtype = {AccountNotFound, BuyRequest, DeliveryRequest, BuyConfirmed,
CreditCheck, BuyFailed, CustomerUnknown, CreditRating, DeliveryConfirmed};

chan Store2CreditAgency = [0] of {mtype, byte};
chan Store2Logistics = [0] of {mtype, byte};
chan CreditAgency2Store = [0] of {mtype, byte};
chan Store2Buyer = [0] of {mtype, byte};
chan Logistics2Store = [0] of {mtype, byte};
chan Buyer2Store = [0] of {mtype, byte};

active proctype Logistics() {
Store2Logistics ? DeliveryRequest(_);
```

```
Logistics2Store ! DeliveryConfirmed(1);
}

active proctype Buyer() {
Buyer2Store ! BuyRequest(1);
if
::
if
::
Store2Buyer ? BuyConfirmed(_);
::
Store2Buyer ? BuyFailed(_);
fi;
::
Store2Buyer ? AccountNotFound(_);
fi;
}

active proctype Store() {
Buyer2Store ? BuyRequest(_);
Store2CreditAgency ! CreditCheck(1);
if
::
CreditAgency2Store ? CreditRating(_);
if
::
Store2Logistics ! DeliveryRequest(1);
Logistics2Store ? DeliveryConfirmed(_);
Store2Buyer ! BuyConfirmed(1);
::
Store2Buyer ! BuyFailed(1);
fi;
::
CreditAgency2Store ? CustomerUnknown(_);
Store2Buyer ! AccountNotFound(1);
fi;
}

active proctype CreditAgency() {
Store2CreditAgency ? CreditCheck(_);
if
::
CreditAgency2Store ! CreditRating(1);
::
CreditAgency2Store ! CustomerUnknown(1);
```

Table 2. Result of test cases.

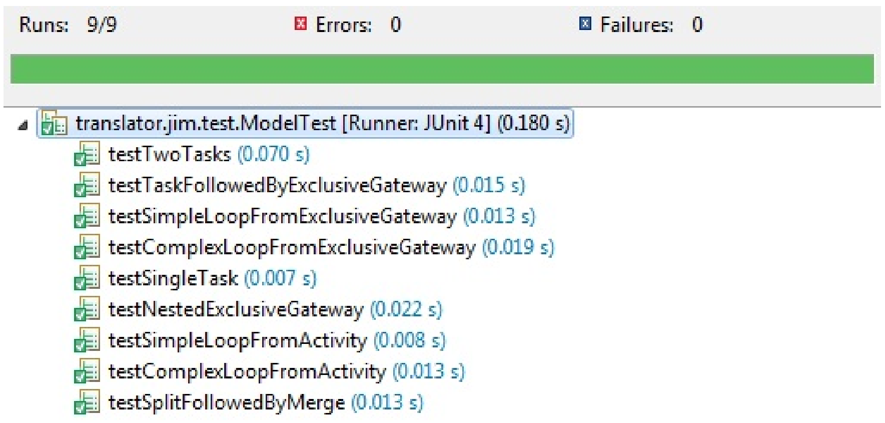
test case description	expect outcome	actual outcome	SPIN syntax checking
single task	common path: 1; loop path: 0; task: 1; message: 1; participant: 2	PASS	NO ERROR
a task follow by another task	common path: 1; loop path: 0; task: 2; message: 2; participant: 2	PASS	NO ERROR
a task followed by a exclusive gateway; the gateway followed by two tasks in different path	common path: 2; loop path: 0; task: 3; message: 3; participant: 2	PASS	NO ERROR
nested exclusive gateways: a gateway is on the branch of another gateway	common path: 3; loop path: 0; task: 9; message: 9; participant: 4	PASS	NO ERROR
simple loop from an activity to another activity (with exclusive gateway)	common path: 1; loop path: 1; task: 3; message: 3; participant: 2	PASS	NO ERROR
complex loop from an activity to another activity (with nested exclusive gateway)	common path: 2; loop path: 1; task: 6; message: 6; participant: 2	PASS	NO ERROR
simple loop from an exclusive gateway to an activity	common path: 2; loop path: 1; task: 3; message: 3; participant: 2	PASS	NO ERROR
complex loop from a nested exclusive gateway to an activity	common path: 3; loop path: 1; task: 5; message: 5; participant: 2	PASS	NO ERROR
a split exclusive gateway followed immediately by a merge gateway	common path: 3; loop path: 0; task: 5; message: 5; participant: 2	PASS	NO ERROR

```
fi;
}
```

C Test Cases

TABLE 2 shows the typical test cases and the result from unit test and SPIN syntax checker.

Fig. 49 shows the result of these test cases.



```
Runs:   9/9                Errors:  0                Failures:  0

translator.jim.test.ModelTest [Runner: JUnit 4] (0.180 s)
    testTwoTasks (0.070 s)
    testTaskFollowedByExclusiveGateway (0.015 s)
    testSimpleLoopFromExclusiveGateway (0.013 s)
    testComplexLoopFromExclusiveGateway (0.019 s)
    testSingleTask (0.007 s)
    testNestedExclusiveGateway (0.022 s)
    testSimpleLoopFromActivity (0.008 s)
    testComplexLoopFromActivity (0.013 s)
    testSplitFollowedByMerge (0.013 s)
```

Fig. 49. The result of junit test.