Formal Verification of Workflow Patterns with SPIN

Cátia Vaz Carla Ferreira

Abstract

In this paper it is proposed the use of model-checking software technology for the verification of the business processes, namely the SPIN model checker. Since the business processes can be decomposed into patterns, it is proposed a translation of a well known collection of workflow patterns into PROMELA, the input specification language of SPIN. The use of this translation is illustrated with two business process scenarios, which demonstrates how their translation to PROMELA models can be useful in business process specification and verification.

Keywords: Workflow Patterns, Formal Verification, Business Analysis, SPIN, BPMN.

1 Introduction

The aim of establishing workflow patterns is to identify generic and recurring constructs of workflow systems. A pattern based approach allows to express the workflow systems core characteristics in a way that is sufficiently generic for its application to a wide variety of technology offerings. So, using workflow patterns offers a language-independent and technology-independent means of expressing the core characteristics of a business process. In [1] is presented a specification of the workflow patterns from the control flow perspective. Namely, it describes activities and their execution ordering through different constructs, which allow flow of execution control such as sequence, choice, parallelism and synchronization.

Since workflow patterns are widely used in business process modeling, which is a domain that requires a certain degree of confidence, it is necessary to obtain a formal semantics for each workflow pattern. The application of formal methods to workflow patterns will allow formal verification of workflow systems. The formal verification can assure if the workflow system has or has not certain properties. Most of commercial workflow products does not have support for verification of workflow models which may lead to models with errors and to undesirable executions of some or all instances of a workflow model. Some formal methods provide semantic analysis tools such as SPIN model checker [2]. With this tool, systems to be verified are described in PROMELA and the properties to be verified are expressed as Linear Temporal Logic (LTL) formulas.

In this paper it is proposed the use of software model-checking technology for the verification of the business processes, namely the SPIN model checker [2]. Since the business processes can be decomposed into patterns, it is proposed a

translation of workflow patterns into PROMELA, the input specification language of SPIN. Namely, we have translated each workflow pattern belonging to the original set of twenty patterns, considering their reviewed definition, described in [1]. The use of this translation is illustrated with two business process scenarios, which demonstrates how their translation to PROMELA models can be useful in business process specification and verification.

This paper is structured as follows. A brief introduction to the SPIN model checker is given in Section 2. Section 3 contains the definitions of workflow patterns and their translation to PROMELA. Two case studies are given in section 4 as well as the verification of some properties. Section 5 discusses related work. Finally, in section 6 conclusions of this work are provided.

2 SPIN

The SPIN model checker is a tool to verify software systems developed by G. J. Holzmann [2]. SPIN provides a specification language, PROMELA, that describes the target system to be a collection of PROMELA processes with channel communications. The language allows for the dynamic creation of processes and both synchronous (rendezvous) and asynchronous communication through communication channels.

A SPIN model is a PROMELA model. Although called a program, it is more an executable model. A PROMELA program consists of variables, channels and processes. Processes are global objects, while variables and channels may be declared either as global or local to a process.

To create a process, one has to define a process template with the keyword `proctype`, and then use the `run` statement to create the process from the template. PROMELA also has the keyword `inline`, which behaves similar to macros and the inline functions in C++ although it has more restrictions. Often, several consecutive statements can be seen as one logical state change. It is possible to group these statements into an `atomic` or `d_step` block, which abstracts the statements into one state change.

Further, SPIN allows to express various properties in terms of linear temporal logic (LTL) [3] and to check if the program satisfies these LTL properties. When an error is found, SPIN reports it and shows the path of execution that led to the error. the model can be in individually during the execution of the model, and examines each state one by one.

In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

3 Workflow Patterns Translation

In this section it is introduced a translation into PROMELA of each workflow pattern belonging to the original set of twenty patterns, considering their reviewed definition, described in [1]. We focus on the translation of the generic workflow constructs. In this translation, processes, sub-processes and activities are mapped into PROMELA processes and control flow paths into PROMELA channels. The constructs are translated into `inline` definitions, which will be

used in the description of control flow dependencies between activities and subprocesses in a workflow process. Messages between processes will be represented, without loss of generality, by integers in PROMELA.

In the translation of these patterns into the PROMELA language, we will use the following notation: `q` will represent a channel and `msg` the message sent or received in channel `q`; `qs` will denote an array of channels of `sizeq` and `msgs` an array of messages of `sizeq` to be sent or received in each channel in the array `qs`. The variable `choice` will be used to denote, when needed, the index of the channel in the array `qs` in which will be sent the message. Also, `choices` is an array of indexes of chosen channels and `activeqs` is an array indicating the active channels. In some patterns, it is also necessary to consider the construct `myRun` that receives a process identification number `id` as a parameter and executes a new instance of the given process. The `id` of each process must be defined within the model. The construct `myRun` has also an instance identification i, which is used for activity instances in the multiple instance patterns. When there is no need to identify the new activity instance executed by the process `myRun`, the instance identification parameter will be -1.

The two workflow constructs that are widely used in the workflow patterns are the `send` and the `receive`. These constructs will be translated as `inline send(q,msg){ q!msg;}` and `inline recv(q,msg){ q?msg;}`, respectively. Note that variables are only declared in the scope of the `proctypes` that use these `inline` definitions.

3.1 Basic Control Flow Patterns

With respect to the Basic Control Flow Patterns, we will present the translation of the patterns **Sequence**, **Parallel Split**, **Syncronization**, **Exclusive Choice** and **Simple Merge**.

Sequence - An activity `B` in a workflow process is enabled after the completion of a preceding activity `A` in the same process. The process `A` in PROMELA should use the `send` definition and the process `B` should use the `recv` definition.

```
chan q = [1] of {int};

proctype A(){
  /* Do work. */
  send(q,1); /* To activate process B. */
}

proctype B(){
  int x;
  recv(q,x); /* Waiting token. */
  /* Do work. */
}
```

Note that the comments `/* Do work. */` in the above processes should be replaced by the details of each activity.

Parallel Split - This pattern is defined as being a mechanism that will allow activities to be performed concurrently, rather than serially. A single path through the process is split into two or more paths so that two or more activities

will start at the same time. This pattern is translated by the following `inline` definition, where each channel in the array `qs` is used to communicate with each activity.

```
inline parallelSplit(qs, sizeq, msg){
  int n;
  n=0;
  atomic {
    do
      :: n<sizeq -> qs[n]!msg[n]; n++;
      :: n>=sizeq -> break;
    od; }
}
```

The process in PROMELA which represents the activity that splits the process must use the `parallelSplit` definition and the processes which represent the activities to be initiated must use the `recv` definition.

Synchronization - The Synchronization pattern combines the paths that were generated by the **Parallel Split** pattern. The final set of activities within the flows must be completed before the process can continue. This pattern is translated to the following `inline` definition.

```
inline synchronization(qs, sizeq, msgs){
  int n, count;
  n=0; count=0;
  /* MAXARRAYSIZE: The capacity of the arrays defined in the file
   * which contains the translations of the workflow patterns */
  int aux[MAXARRAYSIZE];
  do
    ::n<sizeq -> aux[n]=0; n++;
    ::n==sizeq -> n=0; break;
  od;
  skip;
  S:
    if
      ::((aux[n]==0) && (len(q[n]) > 0) && count<sizeq)->
            aux[n]=1; q[n]?msg[n]; count++
      ::count>=sizeq -> goto E
      ::else -> skip;
    fi;
  n++;
  if
    ::n==sizeq -> n=0; timeout;
    ::n<sizeq -> skip;
  fi;
  goto S;
  E: skip;
}
```

The process in PROMELA that receives the input branches must use the above definition and the processes to be syncronized must use the `send` definition. It

is denoted by `aux` an auxiliary array of size `sizeq` to distinguish between the activities already completed from the others. After finishing, each activity sends a message through a channel in the array `qs` to report it (e.g. if the activity that comunicates through channel `qs[n]` is finished, it will send a message through this channel reporting that). In this pattern translation, all the channels in the array `qs` are transverse to see if there is something to receive from each one of them. If there is, it will be received and marked it in the array `aux` (e.g `aux[n]=1`). Thus, the process in PROMELA which use the `syncronization` definition will only continue if it has received a message from each channel. The use of the keyword `timeout` is to avoid process starvation, giving the opportunity to other processes to execute.

ExclusiveChoice - This pattern is defined as being a split of the control flow into two or more exclusive alternative paths. The pattern is exclusive in the sense that only one of the alternative paths may be chosen for the process to continue. This pattern is translated by the following `inline` definition.

```
inline exclusiveChoice(qs,sizeq,choice,msg){
  if :: (choice>=0 && choice<sizeq) -> qs[choice]!msg;
     :: else -> skip;
  fi;
}
```

The process representing the activity which makes the choice must use this definition, and the alternative processes must use the `recv` definition.

Simple Merge - The Simple Merge pattern provides a means of merging two or more control flows of distinct activities without synchronizing them. This pattern corresponds to an exclusive OR-join and is translated as follows.

```
inline simpleMerge(qs,sizeq,msg){
  int n;
  n=0;
  skip;
  S: if :: len(qs[n])>0 -> qs[n]?msg; goto E
        :: len(qs[n])==0 -> n++; goto L
     fi;
  L: if :: n==sizeq -> n=0; timeout; goto S
     /* timeout: to give chance to another process to execute. */
        :: n<sizeq -> goto S
     fi;
  E: skip;
}
```

In this translation, each channel in the array `qs` will represent the communication channel between a specific PROMELA process and the process which merges all the control flow paths. The process in PROMELA which merges the control flows of distinct processes must use the above definition. The processes that represent the activities whose control flow will be joined must use the `send` definition.

3.2 Advanced Branching and Synchronization Patterns

The next five patterns describe more complex ways of splitting and joining the flow of a business process.

Multiple Choice - The Multiple Choice pattern differs from the Exclusive Choice pattern in that the Multiple Choice pattern allows from one to all of the alternative paths may be chosen at execution time. Technically, this pattern may allow zero paths chosen, but this could be considered an invalid situation where the process flow stops unexpectedly. This pattern is translated to the following `inline` definition.

```
inline multiChoice(qs,sizeq,msgs,choices,activeqs){
  int n;
  n=0;
  skip;
  S:if
      ::choices[n]==1 -> activeqs[n]!1;qs[n]!msgs[n]
      ::choices[n]==0 -> activeqs[n]!0
    fi;
    n=n+1;
    if
      ::n<sizeq -> goto S
      ::n>=sizeq -> skip
    fi;
}
```

The process that represents the activity which makes the choice must use this definition, and the alternative processes must use the `recv` definition. Note that `activeqs`, the array that indicates the active channels, is necessary for the **Structured Synchronized Merge pattern**, which will merge some of the alternative paths resulted from a multi-choice. For each of them, a message will be sent through its channel in the array `qs` (e.g `qs[n]`) and thus, a token is also sent through the corresponding channel in the array `activesq` (e.g `activesq[n]`). These tokens will provide information about which channels have been activated.

Structured Synchronizing Merge - This pattern provides a means of merging the paths resulting from a specific **Multi-Choice** construct earlier in a workflow process into a single flow. In this merging is implicit the synchronization of all of the threads of execution resulting from the preceding **Multi-Choice**.

```
inline syncronizeMerge(qs,activeqs,sizeq,msgs){
  int n, total, count1, count2;
  n=0; total=0; count1=0; count2=0;
  /*total: number of active channels.
   *If we receive 1 by the channel activesqs[i],
   *then qs[i] is an active channel.*/
  int aux[MAXARRAYSIZE];
  /* MAXARRAYSIZE: The capacity of the arrays defined in the file
   * which contains the translations of the workflow patterns */
  do
    ::n<sizeq -> aux[n]=0; n++;
```

```
    ::n==sizeq -> n=0; break;
  od;
  skip;
  L: if
       ::((len(activeqs[n])>0) && count1<sizeq) ->
               activeqs[n]?aux[n];count1++;
               if
                 ::aux[n]==1 -> total++
                 ::aux[n]==0 ->skip
               fi;
       ::(count1==sizeq) -> n=0; goto S
     fi;
     n++;
     if
       ::(n==sizeq) -> n=0; timeout; goto L
       ::(n<sizeq) -> goto L
     fi;
  S: if
       ::((aux[n]==1) && (len(qs[n]) > 0) && count2<total)->
               aux[n]=2; qs[n]?msgs[n]; count2++
       :: (count2 >= total) -> goto E
     fi;
     n++;
     if
       ::(n==sizeq)-> n=0
       ::(n<sizeq) -> skip
     fi;
     goto S;
  E: skip;
}
```

With respect to the above definition, for each alterative path resulted from a previous multi-choice, a message will be sent through its channel in the array `qs` (e.g `qs[n]`) and thus, a token is also sent through the corresponding channel in the array `activesq` (e.g `activesq[n]`). These tokens will provide information about which channels have been activated. In addiction, it is also needed in this translation an auxiliary array, `aux`, to distinguish between the active channels in the array `qs` through which it was already received a message, from the active channels in the array `qs` which is still necessary to wait for. The process that represents the activity which makes the merging must use the above definition, and the processes that represent the activities whose control flow will be merged must use the `send` definition.

Multi-Merge - The Multi-Merge pattern represents the convergence of multiple paths in a process into a single path. In this pattern there is no synchronization and each thread of control which is currently active in any of the preceding paths will instanciate de subsequent activity. This pattern is translated to the following `inline` definition.

```
inline multiMerge(qs,activeqs,sizeq,msgs,id){
  int n, aux;
```

```
  n=0;
  skip;
  L: if
       ::(n<sizeq && (len(activeqs[n])>0) && (len(qs[n])>0))->
                  activeqs[n]?aux; qs[n]?msgs[n]; run myRun(id,n);
                  goto E;
       ::else -> n++;
     fi;
     if
       ::n<sizeq -> goto L
       ::n==sizeq -> n=0
     fi;
  S: if
       ::(n<sizeq && (len(activeqs[n])>0)) -> n=0; goto L
       ::else -> n++;
     fi;
     if
       ::n<sizeq -> goto S
       ::n==sizeq -> n=0
     fi;
  E: skip;
}
```

The `id` is the identification number of the PROMELA process that represents the activity which will be instacianted by each active thread of control of each paths. The process in PROMELA, which represents the activity that merges the control flows of distinct processes, must use the `multiMerge` definition. The processes that represent the activities whose control flow will be merged must use the `send` definition.

Structured Discriminator - This is another way of combining the paths that were generated from a **Parallel Split** pattern. It differs from the **Synchronization** pattern in that the thread of control of the first path generated from a **Parallel Split** pattern is passed to the subsequent activity, while the other generated paths do not result in the thread of control being passed on. The discriminator construct resets after receiving the thread of control of all the generated paths. This pattern is translated to the following `inline` definition. It must also include a global variable and a global array, which we will denote by `x` and `array`, respectively. The global variable is to distinguish between the initial process that receives the first thread of control from the process created by the previous one, which has the same `id` and is created only to receive the remaining threads of control of the other paths. The initial process will continue, passing the thread of control. The global array is to control what are the threads of control which remains to receive.

```
inline structuredDiscriminator(qs,sizeq,msgs,id,x,array){
  int n, total;
  n=0; total=1;
  skip;
  L:
    if
```

```
    ::(n<sizeq && len(qs[n])>0 && x==-1) -> x=n; array[n]=1;
                qs[n]?msgs[n]; run myRun(id,n); goto E
    ::(n==sizeq && x==-1) -> n=0; timeout; goto L
    ::(n<sizeq && len(qs[n])==0 && x==-1) -> n++; goto L
    ::(x>=0) -> n=0; goto S
   fi;
  skip;
  S:
   if
    ::(n<sizeq && total<sizeq && len(qs[n])>0 && array[n]==0)->
               qs[n]?msgs[n]; n++; total++; array[n]=1
    ::(total<sizeq && n==sizeq) -> n=0; timeout; goto S;
    ::total==sizeq -> goto E;
    ::else -> n++; goto S;
   fi;
  E: skip;
}
```

The process in PROMELA which represents the activity that combines the control flows of distinct processes must use the above definition. The processes that represent the activities whose control flow will be combined must use the `send` definition.

3.3 Structural Patterns

In what concerns to the Structural Patterns, it will be presented the translation of the patterns **Arbitrary Cycles** and **Implicit Termination**.

Arbitrary Cycles - This pattern can be implemented through a combination of two other patterns, namely **Exclusive Choice** together with **Simple Merge**. It can be seen an example of this situation in one of the case studies presented in the next section.

Implicit termination - This pattern describes a situation where a given process (or sub-process) instance does not have any remaining work to do and so it should terminate. This pattern is implicit in all PROMELA processes which does not have any remaining work to do.

3.4 Multiple Instante Patterns

Multiple Instances without Synchronization - This pattern provides a means of creating multiple instancies of an activity (the number of individual activities required is known before) within a given process instance. These instances are independent of each other and is not necessary to synchronize them. This pattern can be implemented through a combination of two other patterns, namelly **Multi-Choice** together with **Multi-Merge**.

Multiple Instances with a priori Design-Time knowledge - This pattern provides a means of creating multiple instances of an activity (the number of individual activities required is known at design-time) within a given process instance. These instances are independent of each other and is necessary to synchronize them. This pattern is translated to the following `inline` definition.

It will be needed two arrays of channels of `sizeq`, one for sending a message to each one of the multiple instances of an activity that have been created, another to receive a messagem from each of them. Thus, we will denote the arrays of channels by `qsS` and `qsR`, respectively. Similarly, we will denote the arrays of messages to be sent and to be received as `msgS` and `msgR`, respectively.

```
inline mIDTKnowledge(qS, qR, sizeq, msgS, msgR, id){
  int n;
  n=0;
  atomic {
    do
      ::n<sizeq -> run myRun(id, n); n++;
      ::n>=sizeq -> n=0; break;
    od; }
  parallelSplit(qS , sizeq , msgS);
  synchronization(qR , sizeq , msgR);
}
```

After creating the specified number of instances (`sizeq`) of a PROMELA process that represent a specific activity, this pattern uses the **Parallel Split** pattern to split one path into `sizeq` paths of execution, and then syncronize them using the **Synchonization** pattern. The process in PROMELA which creates the multiple instances must include the `mIDTKnowledge` definition.

Multiple Instances with a priori Run-Time knowledge - This pattern provides a means of creating multiple instancies of an activity within a given process instance. The number of individual activities required is known before the activity instances must be created, but is runtime dependent. These instances are independent of each other and is necessary to synchronize them.

The only difference between this pattern and the **Multiple Instances with a priori Design-Time Knowledge** pattern is that the number of instances required is runtime dependent. However, the number of instances required is also known before the activity instances must be created. Thus, since the previous translation is independent of the number of instances to be created (is a parameter of the `mIDTKnowledge` definition), this pattern is translated to the `mIDTKnowledge` definition.

Multiple Instances without a priori Run-Time knowledge - This pattern provides a means of creating multiple instancies of an activity within a given process instance. The number of individual activities required is runtime dependent and it is not known until the final instance has completed. These instances are independent of each other and is necessary to synchronize them.

This pattern requires that the workflow model has globally defined the maximum number of activities that could be instanciated by a process and an array of channels of this size to communicate between the process and the activities. At the end the process must implement the **Synchronization** pattern in order to syncronize the multiple instances of an activity that have been created.

3.5 State Based Patterns

Deferred Choice - This pattern represents a type of exclusive decision, similar to Exclusive Choice pattern. However, the basis for determining the path

depends on the interaction with the operating environment. The only difference between this pattern and the **Exclusive Choice** pattern is that the basis for determining the path depends on the interaction with the environment. Thus, this pattern can be implemented as an **Exclusive Choice** pattern. So, the process representing the activity which makes the choice must use the `exclusiveChoice` definition, and the alternative processes must use the `recv` definition.

Interleaved Parallel Routing - In this pattern, the activities are sequencially performed in any order that accords with a given parcial order of execution. It follows the translation of this pattern.

```
inline parallelRouting(matrix,sizem){
 int i, j, total;
 i=0; j=0; total=0;
 skip;
 I: if
      ::(j<sizem && matrix.l[sizem].c[j]==1) -> total++; j++;
                goto I;
      ::total==sizem -> j=0; goto E;
      ::(j<sizem && matrix.l[sizem].c[j]==0) -> total=0; goto L;
      ::else -> j=0; goto I;
    fi;
 L: if
      ::(j<sizem && i<sizem && matrix.l[j].c[i]==0) -> i++;
                goto L;
      ::(j<sizem && i<sizem && matrix.l[j].c[i]==1) -> i=0; j++;
                goto L;
      ::(j<sizem && i==sizem) -> i=0;
                L1:
                   if
                     ::len(qs[j])>0 -> qs[j]?msgs[j];
                              matrix.l[sizem].c[j]=1; goto S;
                     ::len(qs[j])>0 -> timeout; goto L1;
        fi;
      ::j==sizem -> j=0;
    fi;
 S: if
      ::i<sizem -> matrix.l[i].c[j]=0; i++; goto S;
      ::i==sizem -> i=0; j=0; goto I;
    fi;
 E: skip;
}
```

In the translation, `matrix` is a bi-dimensional array that express the parcial order of execution of the `N` activities to be executed. The matrix has `N1+` lines and `N` columns. The last line indicates if the process has been executed or not. Each entry of the matrix, `matrix.l[i].c[j]` in which `0 <= i,j <N`, can be interpreted in the following way: if the activity with identification number `i` must be executed after the activity with identificator number `j`, then `matrix.l[i].c[j]=1`; otherwise `matrix.l[i].c[j]=0`.

Milestone - The Milestone pattern represents the conditional execution of an

activity (or sub-process) where the process instance is at a given state. The activity is enabled only if the process instance is at a specific state (the milestone); otherwise, the activity can not be enabled.

We denote by `idqx` the index of the channel in the array `qs` that activates the activity whose execution is conditional to the state of the process instance. Each channel in the array `qs` corresponds to alternative paths of control flow that can exist. The information if the process is at a milestone is given by the parameter `x`: if `x=1` it is; if `x=0`, it is not.

```
inline milestone(qs,sizeq,choice,msg,idxq,x){
  if
    :: x>=1 -> exclusiveChoice(qs,sizeq,choice,msg);
    :: x==0 -> if
                 ::(choice>=0 && choice<sizeq &&
                           (choice< idxq || choice>idxq)) ->
                              qs[choice]!msg;
                 ::(choice==idxq) -> skip;
               fi;
  fi;
 }
```

The process representing the activity whose thread of control could activate the activity whose execution is conditional to a milestone should use this definition.

3.6 Cancellation Patterns

With respect to the Cancellation Patterns, we will present the translation of the **Cancel Activity** and **Cancel Case** patterns. To translate these it is necessary that each PROMELA process that may be canceled have a specific channel for this purpose. Thus, if the process receives a message from that channel, it should terminate. We will denote a single cancel channel as `qCancel` and an array of canceling channels as `qsCancel`.

Cancel Activity - The Cancel Activity pattern provides a mean of withdraw an enabled activity before starting to execute. However, if the activity has already started, it is disabled and, where possible, the currently running instance is halted. To translate this pattern it is necessary that the PROMELA process which represents the activity to be canceled includes an escape sequence as follows.

```
unless { len(qCancel)>0; skip; }
```

Cancel Case - This pattern describes the situation where is necessary to remove a complete process instance. This includes executing activities, those which may execute at some time and all sub-processes. More generally, this may be used to cancel individual activities, regions or the whole workflow instance. The PROMELA model needs to have a global array `piIds` representing the relations between the processes, e.g., the parent of the process with identification number `x` is the process with identification number `piIds[x]`. In the following `inline` definition, `id` is the identification number of the process where this inline definition is being included.

```
inline cancelCase(qsCancel,sizeq,piIds,msgs,id){
  int i;
  i=0;
  do :: i<sizeq && piIds[i]==id -> qsCancel[i]!msgs[i];i++;
     :: i==sizeq ->break;
     :: else -> i++;
  od;
}
```

To implement this pattern, the process and all its activities and sub-processes must include an escape sequence as follows.

```
unless{ len(qsCancel[id])>0;
        cancelCase(qsCancel,sizeq,piIds,msgs,id) }
```

Each process that receives a message from its canceling channel `qsCancel[id]`, sends a canceling message, before terminate, to each one of its sub-processes or activities in order to cancel them.

4 Case Studies

In order to ensure the reliability of business process, formal verification methods are needed. This section shows how the above translation of workflow patterns can be useful for the formal verification of business process models. Two standard examples of business processes are the Loan Approval and the Travel Agency. In this section it will be illustrated how to apply the translation described above to both business process examples and, subsequently, properties of these processes will be checked. It will be also illustrated how these business processes can be modeled in Business Process Modeling Notation (BPMN) [4].

4.1 Loan Request

This case study consists of a simple loan approval where customers can submit their requests for loans. A description of this process can also be found in [5].

The loan process starts with the activity **Loan Request**. This activity is initiated with a request made by a costumer for a loan. The costumer must fill an application specifying personal information and the amount being requested. The activity **Validate Application** will check the request made in the previous activity and will send the result to the next activity, **Application Ok**. In the activity **Application Ok** it may occur one of the three possible situations: the application is correct and is necessary to send that information to the next activity, **Approve**; the application must be checked again and it is necessary to send a message to the activity **Validate Application** in order to revalidate the application; the application is not correct and is necessary to send that information to the next activity, **Send Rejection**. In the activity **Approve** the received information is processed and sent to the activity **Approved?**. Finally, in the activity **Approved?** there are three possible decisions: the loan request can be approved and thus it sends a message to notice the **Send Approval** activity; the loan request cannot be approved yet, but it is possible to approve

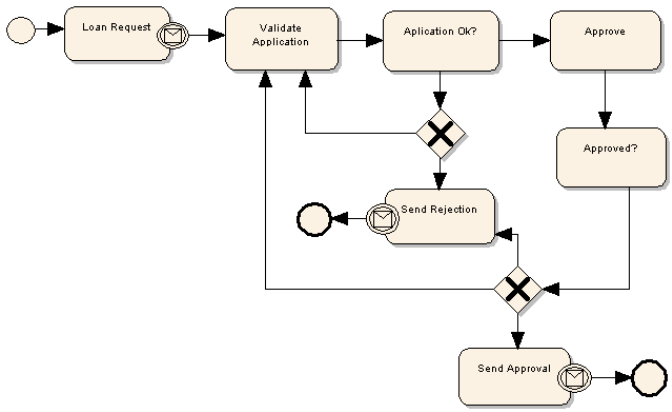

Figure 1: A BPMN diagram for the Loan Request process.

it after revalidation (in this case, it is necessary to send a message to the activity **Validate Application**); the loan request must be rejected and then it is necessary to send a message to the **Send Rejection** activity.

Figure 1 shows the business process modeled as a BPMN diagram. We choose to present the model as a BPMN diagram because BPMN supports the arbitrary cycles pattern, needed to specify the Loan Approval example.

The PROMELA Model

In what follows, we will specify as PROMELA processes the activities involved in Loan Approval process. In this PROMELA model, we will need nine global channels (denoted by the array of channels `qs`) and each one will allow to establish the communication between two given activities, e.g., `qs[0]` in the case of **Loan Request** and **Validate Application**.

We denote by `qs1`, `qs2` and `qs3` the arrays of channels defined within the processes. Each of these auxiliary arrays of channels include only some of the channels in the array `qs`, as required by the patterns translation.

Loan Request - This activity initiates by receiving a request from a costumer for a loan and is translated by the following PROMELA process.

```
proctype getLoan(){
  /* The client request a loan. */
  send(qs[0],1); /* To activate process validadeApplication. */
}
```

Validate Application - This activity check the request made by the costumer, which can be evaluated more than once. It is translated to the PROMELA process `validateApplication`. Note that the value of `x` could be changed in this process. It will depend on the details of the business process, which we are not considering. So, by default, `x` will be 1. However, `x` will be -1 when the loan request is approved or when it is rejected.

```
proctype validateApplication(){
  int x; chan qs1[4]= [1] of {int};
```

```
  qs1[0]=qs[8]; /* To receive from applicationOk. */
  qs1[1]=qs[6]; /* To receive from approved. */
  qs1[2]=qs[7]; /* To receive from sendRejection or from
                 * sendApproval. */
  qs1[3]=qs[0];
  recv(qs[0],x); /* Receive from getLoan. */
  /* Check the application filled in the activity Loan Request. */
  send(qs[1],x); /* Send to applicationOk. */
  L: simpleMerge(qs1,4,x);
     /* Revalidate the application filled. */
     if
       ::x==-1 -> skip;
       ::x!=-1 -> send(qs[1],x); goto L
     fi;}
```

Application Ok - As stated before, in this activity is chosen one of three possibilities. Since the choice of one of the possibilities depends on the details of the business process, we choose non determinalistically, without loss of generality to the verification, one of the three possibilities. **Application Ok** is translated to the PROMELA process `applicationOk`.

```
proctype applicationOk(){
  int y, x;
  chan qs2[3]=[1] of {int};
  qs2[0]=qs[8]; /* To send to validateApplication. */
  qs2[1]=qs[2]; /* To send to sendRejection. */
  qs2[2]=qs[3]; /* To send to approve. */
  S: recv(qs[1],y); /* To receive from validateApplication. */
     /* Note that the if construct chooses, non
      * determinalistically, one of the choices. */
     if
       ::x=0
       ::x=1
       ::x=2
     fi;
     exclusiveChoice(qs2,3,x,1);
     goto S;
}
```

Send Rejection - The Send Rejection activity rejects the loan request and terminates the process. It is translated to the following PROMELA process.

```
proctype sendRejection(){
  int z;
  recv(qs[2],z); /* Receive from applicationOk or approved. */
  send(qs[7],-1); /* Send to validateApplication. */
  /* Send rejection notice to the costumer. */
}
```

Approve - This activity receives information from the activity **Application Ok**, processes the information and activate the next activity.

```
proctype approve(){
  int x;
  T: recv(qs[3],x); /* Receive from ApplicationOk. */
     /* Processes the information. */
     send(qs[4],1); /* Send to approved. */
     goto T;
}
```

Approved? - In this activity, it must be decided whether the loan request can be approved or not. As stated before, there are three possible decisions. Once again, we choose non determinalistically, without loss of generality to verification, one of the three possibilities. This activity is translated to the PROMELA process `approved`.

```
proctype approved(){
  int y, x;
  chan qs3[3]=[1] of {int};
  qs3[0]=qs[6]; /*to send to validateApplication*/
  qs3[1]=qs[5]; /*to send to sendApproval*/
  qs3[2]=qs[2]; /*to send to sendRejection*/
  R: recv(qs[4],y); /*to receive from Approve*/
     if
       ::x=0
       ::x=1
       ::x=2
     fi;
     exclusiveChoice(qs3,3,x,1);
     goto R;
}
```

Send Approval - The loan request is approved and the activity **Validate Application** is informed. The following process shows the translation of this activity.

```
proctype sendApproval(){
  int z;
  recv(qs[5],z); /* To receive from approved. */
  send(qs[7],-1); /* To send to validateApplication. */
  /* Send approval notice to the costumer. */
}
```

Bellow is the corresponding PROMELA model description of the BPMN diagram in figure 1.

```
/* File with the translations of the workflow patterns. */
#include "utils.pr"

chan qs[10] = [1] of {int};
proctype getLoan(){...}
proctype validadeApplication(){...}
proctype applicationOK(){...}
proctype sendRejection(){ ...}
```

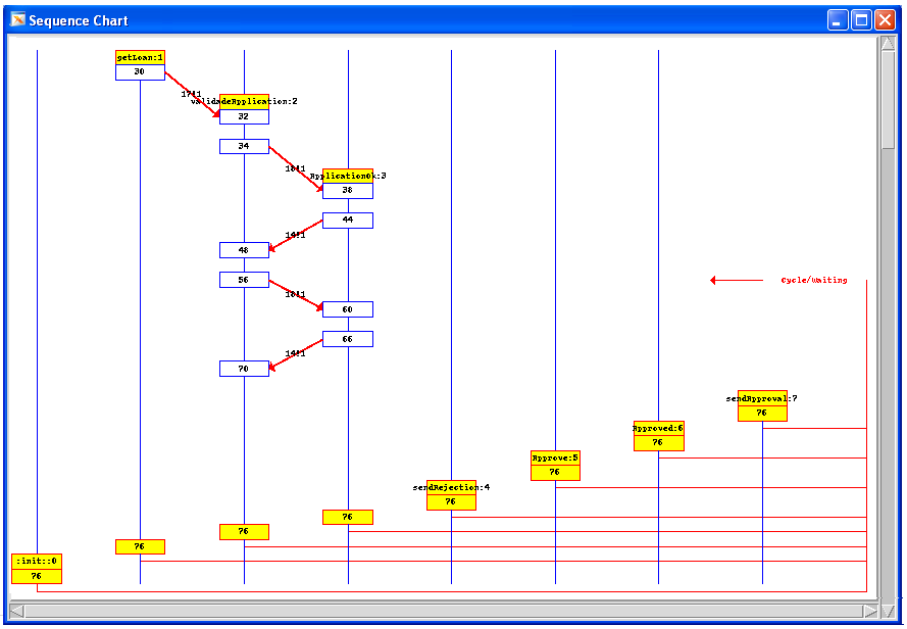

Figure 2: Message Sequence diagram that shows a path of execution that led to the error.

```
proctype approve(){...}
proctype approved(){...}
proctype sendApproval(){...}
init { atomic{ run getLoan(); run validadeApplication();
               run applicationOk(); run sendRejection();
               run approve(); run approved();
               run sendApproval(); } }
```

Properties Verification

It is possible to check if the Loan Approval business process model satisfies certain properties using the SPIN model checker, it will considered two properties that this model should verify. In order to express them, a global `int` variable `s` will be used and it will have the value `0` when the process begins, the value `1` before process `sendApproval` finishes and the value `2` before process `sendRejection` finishes.

One of the properties to check is that the process never approves and rejects the same loan request. By making the definitions, `#define q (s==1)` and `#define r (s==2)`, it is possible to formally express this property in LTL as `!(<>( q && r ))`. This property was automatically verified with the SPIN model checker.

Another property to check is that whenever this business process is invocated, the process either approves or rejects the loan request. By making the definitions `#define p (s==0)`, `#define q (s==1)` and `#define r (s==2)`, it is possible to formally express this property in LTL as `[] (p -> <> (q || r))`. It is interesting to note that this property is not verified by the SPIN model

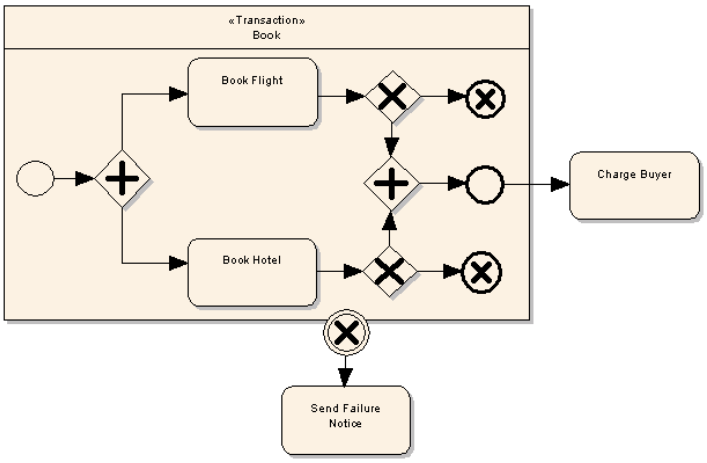

Figure 3: A BPMN diagram for the Travel Agency Process.

checker. SPIN shows the error and the path of execution that led to the error, i.e., the counterexample, as it can be seen in figure 2. In fact, the problem is that there are paths of execution that never lead to a state where the loan request is rejected or approved. This is because that in the specification there are no limit in the number of times that the request could be revalidated. This property is then verified by limiting the number of times that the same loan request could be in the activity `validate application` and in the activity `Approved`.

4.2 Travel Agency

This case study consists of a simple travel agency where customers can book a trip. The process of booking trips involves booking a flight and a hotel. If both bookings succeeds, the payment follows. Otherwise, the booking of the trip is canceled. A description of this process can also be found in [4].

Figure 3 shows the business process modeled as a BPMN diagram. The process of booking is represented by the transaction **Book** and executes in parallel two activities: the activity **Book Flight** and **Book Hotel**. If both activities succeed, the activity **Charge Buyer** follows, otherwise, the process **Book** is canceled and follows the activity **Send Failure Notice**.

The Promela Model

In what follows, the activities involved in Travel Agency example will be specified as Promela processes. The Promela model will have two global channels. The array of channels denoted by `qs` is used for communication between the process **Book** and its sub-processes **Book Flight** and **Book Hotel**. However, in the situation of canceling, the communication among these processes is made by their respective canceling channels, denoted by `qsCancel`. The other two channels, `q1` and `q2`, are used by process **Book** to communicate, when needed, with processes **Failure Notice** and **Charge Buyer**. We denote by `qs1` and `qs2` the arrays of channels defined within the processes. Each one of these auxiliary array of channels include one channel of `qs` and one channel of `qsCancel`.

Book - This PROMELA process translates the process of booking.

```
proctype Book(){
{ int ids=1; /* The id of this process is 1. */
  piIds[ids]=0;
  int msgs[4];
  /* Receive personnal information of the costumer. */
  ids++;
  piIds[ids]=1;
  run myRun(2,-1); /* Run BookFlight. */
  ids++;
  piIds[ids]=1;
  run myRun(3,-1); /* Run BookHotel. */
  parallelSplit(qs,2,1);
  synchronization(qs,2,msgs);
  send(q1,1);
} unless { len(qsCancel[1])>0;
           cancelCase(qsCancel,4,piIds,msgs,1); send(q2,1); }
}
```

Book Flight - This activity represents the booking of a flight. In this activity is decided if it is possible to book a flight suitable for the interests of the costumer. If it is not possible, the activity **Book Flight** is canceled and this information is sent to the process **Book** in order to the whole process be canceled. Since the choice of one of the possibilities depends on the details of the business process, we choose non determinalistically, without loss of generality to the verification, one of the three possibilities.

```
proctype BookFlight(){ /* 2 is the id of this process. */
{ int x, msgs[4];
  chan qs1[2];
  qs1[0]=qs[0]; qs1[1]=qsCancel[1];
  recv(qs[0],x); /* Waiting token. */
  /* Decide if it is possible or not to book a flight. */
  if
    ::x=0 /* Not to cancel. */
    ::x=1 /* To cancel. */
  fi;
  exclusiveChoice(qs1,2,x,1);
  } unless { len(qsCancel[2])>0;
             cancelCase(qsCancel,4,piIds,msgs,2); }
}
```

Book Hotel - This activity represents the booking of a hotel. In this activity is decided if it is possible to book a hotel suitable for the interests of the costumer. If it is not possible, the activity **Book Hotel** is canceled and this information is sent to the process **Book** in order to the whole process be canceled. Once again, we choose non determinalistically, without loss of generality to verification, one of the two possibilities.

```
proctype BookHotel(){ /* 3 is the id of this process. */
```

```
{ int x, msgs[4];
  chan qs2[2];
  qs2[0]=qs[1]; qs2[1]=qsCancel[1];
  recv(qs[1],x); /* Waiting token. */
  /* Decide if it is possible or not to book a hotel. */
  if
    ::x=0 /* Not to cancel. */
    ::x=1 /* To Cancel. */
  fi;
  exclusiveChoice(qs2,2,x,1);
  } unless { len(qsCancel[3])>0;
             cancelCase(qsCancel,4,piIds,msgs,3); }
}
```

Charge Buyer - This activity is activated by the process **Book** in order to charge a payment to the costumer that has requested the booking.

```
proctype ChargeBuyer(){
  int x; recv(q1,x);
  /* Charge the costumer. */}
```

Send Failure Notice - This activity is activated by the process **Book** and must inform the costumer that the booking process has failed.

```
proctype sendFailure(){
  int x; recv(q2,x);
  /* Send failure notice to the costumer. */}
```

Bellow is the corresponding PROMELA model of the BPMN diagram in figure 3.

```
/* File with the translations of the workflow patterns. */
#include "utils.pr"

chan qs[2] = [1] of {int};
chan q1= [1] of {int}; chan q2= [1] of {int};
chan qsCancel[4]= [1] of {int};
int piIds[4];

proctype myRun(int id, int n){
  if
    ::(id==1) -> run Book()
    ::(id==2) -> run BookFlight()
    ::(id==3) -> run BookHotel()
  fi;}

proctype Book(){...}
proctype ChargeBuyer(){...}
proctype SendFailure(){...}
proctype BookFlight(){...}
proctype BookHotel(){...}
init{ atomic{ run Book(); run ChargeBuyer();
              piIds[0]=-1; run SendFailure();} }
```

Properties Verification

It is possible to check if the Travel Agency business process model satisfies certain properties using the SPIN model checker. It will be checked one property that this model should verify. We will express it with the help of one global `int` variable `s`, which has the value `0` when the process begins, the value `1` before process `ChargeBuyer` finishes and the value `2` before process `SendFailure` finishes.

The property is that whenever we invocate this business process, the process either charge the buyer or send him a failure notice to tell him that the booking was not possible. By making the following definitions `#define p (s==0)`, `#define q (s==1)` and `#define r (s==2)`, it is possible to formally express this property in LTL as `[] (p -> <> (q || r))`. This property was automatically verified with the SPIN model checker.

5 Related Work

Currently research has been done into the application of SPIN to workflow specification and verification. In [6] is proposed a method to check correctness properties of workflows implemented in BPEL. Dataflow networks are used to define the formal semantics of the workflow. The BPEL model is mapped into dataflow network and the dataflow network is mapped into a PROMELA model. A method of verifying Web Services Flow Language (WSFL) [7] description by using SPIN is introduced in [8]. In particular, it presents an encoding method that translates WSFL primitives to PROMELA. These approaches are both focused on the translation of a workflow language to PROMELA, while this work is focused on the translation of workflow patterns to PROMELA, which are language and technology independent, in order to formally verify workflow systems.

An approach of giving a representation of the workflow patterns has been developed with π-calculus [9], although it is not oriented towards automated verification. A benefit of using SPIN is in terms of visualizing counterexamples for negative results. Also, as the verification in π-calculus is done by checking bisimulation equivalence, some times results are not obtained in reasonable amount of time, even for the proofs of very simple correctness requirements [10]. There is also being used a subset of π-calculus to model workflow patterns, Calculus of Communicating Systems (CCS) [11], but does not conform to standard CCS, and does not have a verification tool. [12].

In [13] is given a representation of these patterns in Communicating Sequential Processes (CSP) [14]. Comparing CSP and PROMELA, PROMELA is richer and strictly more expressive (e.g. asynchronous communication is supported and channels are first class objects in Promela but not in CSP). Furthermore, Promela's C-like syntax makes it more accessible to non-experts [15].

Yet Another Workflow Language (YAWL) [16] was also used to represent workflow patterns. However, we believe that since SPIN is a model checker and PROMELA has a similar syntax to C it has an advantage over YAWL.

6 Conclusions

In this paper, it is proposed the use of software model-checking technology for the verification of the business processes, namely the SPIN model checker. Since the business processes can be decomposed into patterns, it is proposed a translation of a well known collection of workflow patterns into PROMELA. This translation is applied to two case studies, namely the Loan Request and the Travel Agency examples. It is also illustrated the verification of some properties and it has been seen that, in despite of the simplicity of the Loan Request example, it has an error which was discovered by the SPIN model checker. Thus, this simple example demonstrate how PROMELA models can be useful in business process specification and verification.

Future work will concern an automatic translator of BPMN models (or models described in other process modeling languages) to PROMELA models. It is also interesting to further express required properties of the workflow patterns in linear temporal logic in order to verify them with SPIN model checker.

References

[1] Russell, N., ter Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22 (2006)

[2] Holzmann, G.: The SPIN MODEL CHECKER. Primer and Reference Manual. Addison-Wesley. Pearson Education (2003)

[3] Manna, Z., Pnueli, A.: The temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1991)

[4] OMG: Business Process Modeling Notation. Specification (2006)

[5] Havey, M.: Essential Business Process Modeling. O'Reilly Media, Inc. (2005)

[6] Kovács, M., Gönczy, L.: Simulation and Formal Analysis of Workflow Models. In: Proc. of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Notes in Theoretical Computer Science, Vienna, Austria, Elsevier (2006) 215–224

[7] Leymann, F.: Web Services Flow Language. IBM (2001)

[8] Nakajima, S.: Verification of Web Service Flows with Model-Checking Techniques. IEEE Computer Society **00** (2002) 0378

[9] Puhlmann, F., Weske, M.: Using the *pi*-calculus for formalizing workflow patterns. In: Business Process Management. (2005) 153–168

[10] Song, H., Compton, K.J.: Verifying pi-calculus processes by Promela translation. Technical report, Univ. of Michigan (2003)

[11] Milner, R.: Communication and concurrency. Prentice Hall International (UK) Ltd. (1995)

[12] Stefansen, C.: SMAWL: A SMAll Workflow Language Based on CCS. CAiSE Forum (2005)

[13] Wong, P.Y., Gibbons, J.: A process-algebraic approach to workflow specification and refinement. In: Proceedings of the 6th International Symposium on Software Composition (SC2007). (2007)

[14] Roscoe, A.: Theory and Practice of Concurrency. Prentice Hall (1997)

[15] Currie, A.J.: A Comparison of Three Model Checkers Applied to a Distributed Database Problem. In: IWFM. (2000)

[16] van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Information Systems **30**(4) (2005) 245–275