DECLARE: Full Support for Loosely-Structured Processes

Maja Pesic
Department of Technology Management
Eindhoven University of Technology
The Netherlands
Email: m.pesic@tue.nl

Helen Schonenberg
Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands
Email: m.h.schonenberg@tue.nl

Wil M.P. van der Aalst
Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands
Email: w.m.p.v.d.aalst@tue.nl

***Abstract*—Traditional Workflow Management Systems (WFMSs) are not flexible enough to support loosely-structured processes. Furthermore, flexibility in contemporary WFMSs usually comes at a certain cost, such as lack of support for users, lack of methods for model analysis, lack of methods for analysis of past executions, etc. DECLARE is a prototype of a WFMS that uses a constraint-based process modeling language for the development of declarative models describing loosely-structured processes. In this paper we show how DECLARE can support loosely-structured processes without sacrificing important WFMSs features like user support, model verification, analysis of past executions, changing models at run-time, etc.**

I. Introduction

Management of business processes automated by workflow management systems (WFMSs) is limited by properties of these systems [2], [13], [19]. Due to the rigidity of WFMSs, it is hard for organizations to maintain flexible business processes in WFMSs. The issue of flexibility in WFMSs has become one of the most frequently addressed problems within this field [4], [9], [14], [15], [18], [21], [23], [30]. These research approaches typically focus on the central element driving any WFMS: the *process model* describing the ordering or activities.

A process model defines a business process and determines the way the cases (i.e., process instances) are handled in an organization. Therefore, the model heavily influences the degree of flexibility of the WFMS. In traditional WFMS, rigid modeling languages (e.g, BPMN, EPCs, UML-ADs, or more formal languages such as Petri nets [28] and Pi calculus [24]), define a process model as a detailed specification of a step-by-step procedure that should be followed during the execution. This approach is an *imperative* approach because it strictly specifies *how* the process will be executed and yields *highly-structured processes*. The major drawback of this approach is the fact that users have limited influence on the process under execution, since most decisions about the execution are already made during process modeling phase.

Flexible systems aim at offering flexibility to the user by shifting decision making from the system to the user. As a result, research on flexibility in WFMSs focusses on a different notion of the process model. For instance, case-handling systems [27] are WFMSs that offer more flexibility by adding special actions that users can perform while working with imperative models. One example of a case-handling system is FLOWer [25], where users can "soften" the imperative nature of models by re-doing past actions or skipping unnecessary actions. Adaptive systems [21] are another example of flexible systems that use imperative models. Systems like ADEPT [29] use powerful and complex mechanisms that allow users to change process models (by inserting, moving or deleting activities) during execution. Although these solutions increase flexibility to a great extent, the imperative nature of models remains, which can produce other burdens for users. For example, in case-handling systems the user cannot choose to re-do only one activity from the past – she will have to re-do all successor activities. On the other hand, the adaptive approach requires users to have modeling expertise to change models for occurring deviations.

One of the key problems of existing workflow languages is that they force or stimulate the designer to *over-specify* things. For example, it is possible to model all kinds of choices. However, it is not possible to simply state that two activities should never occur together. Instead, the user is forced to provide a detailed strategy to implement this simple requirement.

We believe that replacing the imperative approach with a declarative one is essential for making WFMSs more flexible while avoiding the problems mentioned above. Therefore, we propose a system for supporting declarative (loosely-structured) process models: DECLARE. DECLARE is developed as a constraint-based system and uses a declarative language grounded in temporal logic [20] for the development and execution of process models. Even though it is a declarative system, DECLARE can offer most features that traditional WFMSs have: model development, model verification (finding errors in models), automated model execution, changing models at run-time (i.e., adaptivity), analysis of already executed processes, and decomposition of large processes. In addition, DECLARE can be used to overcome the everlasting paradox/trade-off between user support and flexibility by providing the user with history based recommendations during process execution. To achieve this, DECLARE cooperates with two other tools, as shown in Figure 1: YAWL [1] and ProM [11]. YAWL is an open-source workflow management system inspired by the well-known workflow patterns [3] and

good at handling structured workflows. YAWL and DECLARE work together in such as way that structured parts of the process are handled by YAWL while unstructured parts are handled by DECLARE, i.e., there can be an arbitrary nesting of of YAWL and DECLARE processes. The process-mining tool ProM [11] is used for analysis of past executions of DECLARE models and recommendations. DECLARE itself consists of three typical components that every WFMS has: (1) a modeling component called *Designer*, that is used for system settings and process model development, (2) a component for process enactment, called *Framework* which is also used for communication with YAWL and ProM, and changing models at run-time, and (3) a component for process execution, called *Worklist* which is a simple tool for users to execute processes and see recommendations.

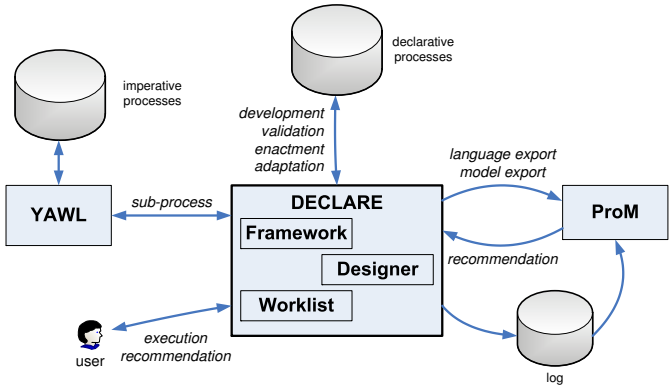


Fig. 1. System architecture

The remainder of the paper is organized as follows. Section II starts with the description of the constraint-based approach and main features of DECLARE. Section III describes how DECLARE cooperates with YAWL [1]. Analysis of past executions of DECLARE models with a process-mining tool ProM [11] is described in Section IV. User support with history based recommendations is presented in Section V. Related work and future work are described in Sections VI and VII, while Section VIII concludes the paper.

II. Constraint Based Approach

As opposed to traditional imperative approaches to process modeling, DECLARE uses a constraint-based declarative approach. Figure 2 shows the differences between the two approaches. An imperative model focuses on specifying exactly how to execute the process, i.e., all possibilities have to entered into the model by specifying its control-flow. A declarative model specifies a set of constraints, i.e., rules that should be followed during the execution. In this way, the declarative model implicitly defines the control-flow as all possibilities that do not violate any of the given constraints, as shown in Figure 2(a).

By defining how to execute the process, imperative languages tend to over-specify the process model. Consider, for example, a situation where there are two activities “A” and “B” that exclude one another, i.e., if “A” is executed for a particular case (process instance), then “B” cannot be executed for the same case and vice versa. Using DECLARE this can be modeled easily as shown in Figure 2(b), the connection between “A” and “B” describes this constraint graphically. Imperative languages do not have such a construct.[1] As a result, the requirement needs to be translated into lower level constructs. In this case, the obvious approach is to add a decision activity “X” which makes a choice between “A” and “B” as shown in Figure 2(c). The conditions “c1” and “c2” shown in Figure 2(c) need to be mutually exclusive. Although activity “X” did not play a role in the initial requirement, the designer needs to specify this activity and decide when it is executed. Moreover, conditions “c1” and “c2” need to be specified. Clearly, such an approach leads to an *over-specification* of the desired behavior even though the initial requirement was simple and unambiguous.

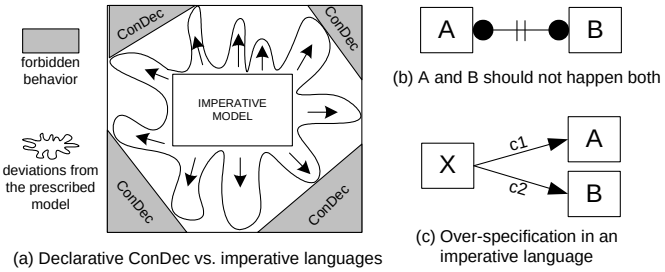


Fig. 2. Imperative vs. declarative approach

We will use a simple process of renting rooms in a hotel as an illustrative example to describe process modeling and execution in DECLARE. This process consists of seven activities: (1) “register client data” - enter client name, preferred way of payment (e.g., cash or credit card), identification, etc; (2) “bill” - altering the billing specification with costs for stay, room service, laundry service, etc (used to charge the total price of the stay); (3) “room service” - register the room service for client; (4) “laundry service” - register the laundry service for client; (5) “additional cleaning” - although rooms are cleaned on a regular basis, additional cleaning is sometimes necessary; (6) “charge” - client is charged with an amount for received service(s); and (7) “check-out” - client checks out at the reception. Figure 3 shows a DECLARE model (in the *Designer* component) for the hotel example containing all the mentioned activities. Actually, this is already a valid model in DECLARE that can be saved and executed. In this case, the number of possible executions (i.e., process instances, cases) is infinite; it is possible not to execute any activity, or to execute each of the activities an arbitrary number of times (0..*, i.e., zero or more), and activities can be executed in any order.

[1]Note that DECLARE is not a single, fixed language but allows for the definition of multiple languages. One of the languages realized using DECLARE is ConDec [26]. The notation shown in Figure 2(b) is part of the ConDec language. We will elaborate on the subtle difference between DECLARE and ConDec later in this paper.

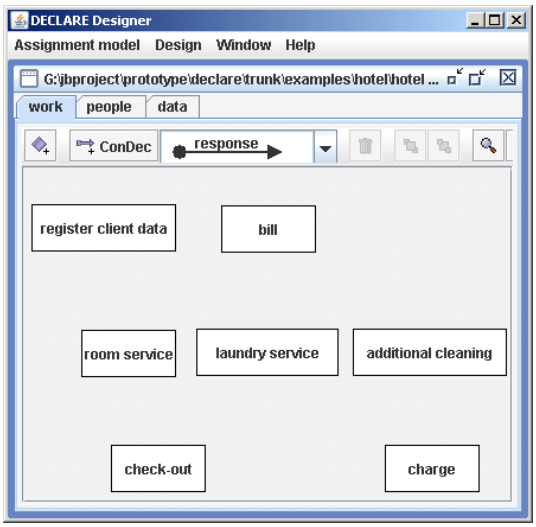


Fig. 3. Activities in the hotel example

Naturally, the hotel process needs to follow some simple rules:

- (C.1) Every process instance has to start with activity "register client data". Data can also be altered at later stages (e.g., the client changes payment details).
- (C.2) Activity "bill" must be executed at least once, i.e., at least the number of nights will be billed. However, it might be that the bill is altered multiple times for a given case (e.g., room service during stay, damage in the room, etc.).
- (C.3) Every "room service" must be billed. However, it is possible that several services are billed at once, instead of billing each service separately.
- (C.4) Every "laundry service" must be billed. However, it is possible that several services are billed at once, instead of billing each service separately.
- (C.5) When the client "checks-out" the bill must be "charged". It might be the case that the bill was charged before check-out, during check-out or even after check-out (e.g., credit card payment). Also, it must be possible that the total amount is charged at several stages during the stay.

Note that this set of rules also allows for undesired but unavoidable behavior. For example it is undesired that clients leave the hotel without paying, i.e., cases where activity "charge" is never executed. However, if the client forgets to "check-out" the system cannot enforce that the client pays (activity "charge"). This situation and many other exceptional situations are covered by the hotel process, as long as the previous five rules are followed. Note that in a traditional WFMS there is a tendency to describe an ideal world where these exceptional situations do not exist. However, in the real world these exceptional situations do exist and need to be handled.

Figure 3 allows for any behavior involving the seven hotel activities mentioned. DECLARE offer the possibility to specify rules such as C.1-5 as *constraints*.

A. Constraint Templates

While traditional modeling languages offer a predefined set of types of relations between activities (sequence, choice, parallelism, and loop), DECLARE allows for customized specification of relation types, or as we call it *constraint templates*. In our small, illustrative example there are two constraints of the same type. Constraint (C.3) and (C.4) specify the same type of relation between activities, namely every execution of "room service" (C.3), or every execution of "laundry service" (C.3) will eventually be followed by at least one execution of "billed". This type of relation is known as "response" and is one of the many possible *constraint templates* supported by DECLARE.

In DECLARE it is possible to create a constraint template for a relation type. Each constraint template has (1) an unique name, (2) *semantics* specified in Linear Temporal Logic[2] (LTL) [20] and (3) *graphical representation*. LTL is a special type of logic that uses (in addition to classical logical operators) several temporal operators: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), and next time ($\bigcirc$). The major benefit of constraint templates is that users do not have to be LTL experts to work with the system; they work with a graphical representation of templates, while the underlying LTL formula remains "hidden".

Figure 4 shows a screenshot of DECLARE while defining a constraint template. Here the definition of the "response" template is shown. Note that the "response" template is defined as a "binary" relation between two activities. The graphical representation of any template is a line for which the beginning, end and middle part can be defined. Figure 4 shows that the "response" template is graphically represented by a single line with a filled circle next to the first activity ("parameter 1"), a filled arrow symbol next to the second activity ("parameter 2") and without a special symbol in the middle. Furthermore a textual description and the LTL formula for "response" are defined.

Defined constraint parameters can be used in LTL expressions, as shown in the lower part of Figure 4. In the LTL expression $\Box(A \Rightarrow \Diamond(B))$ for the "response" template parameters "A" and "B" are used in the template formula to define the desired relation between the parameters. When used in the model, formal parameters of the template are replaced by real activities in the model (e.g., parameter "A" is replaced with activity "room service" and parameter "B" with activity "bill"). If a parameter is specified to be "branched", this would allow the parameter to be replaced with a conjunction of several activities.

The "response" template is an example of a binary template, i.e., it specifies a relation between two activities. Clearly, it is also desirable to define non-binary templates. For example, it is possible to define unary templates, which involve only

[2]DECLARE has been developed in a way that it allows for implementation of specification languages other than LTL.

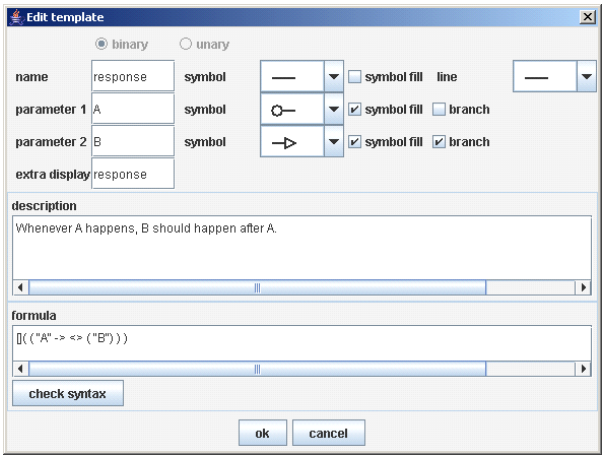


Fig. 4. Constraint template "response"

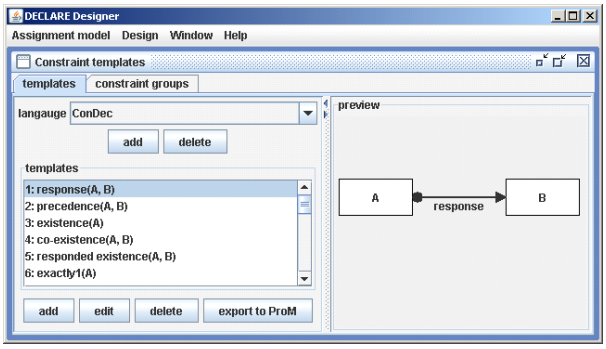


Fig. 5. Defining a language

one activity (e.g, specifying that an activity has to executed at least two times). DECLARE aims at using templates with an arbitrary number of parameters to allow for complicated splits and joins, e.g., the n-out-of-m split/join. Currently, DECLARE partially supports such constructs and is being extended to support templates with an arbitrary number of parameters.

Note that DECLARE uses somewhat more granular approach to parameters than shown in Figure 4. Written as it is, parameters "A" and "B" are replaced by "A.completed" and "B.completed", respectively. This means, that when used in our example, the "response" template between activities "room service" and "bill" specifies a relationship between events "room service.completed" and "bill.completed". However, it is possible for a template to specify relations between events of starting or canceling activities (e.g., $\Box(A.started \Rightarrow \Diamond(B.started))$).

Different application domains can require a different set of constraint templates. Therefore, DECLARE facilitates the definition of sets of constraint templates, also called *languages*. In other words, DECLARE is not a fixed language but allows for the definition of different languages. Each language has an unique name and is defined by a set of constraint templates. ConDec [26] is one of the languages created using DECLARE. DecSerFlow [5] is another language created using DECLARE tailored towards the specification of web services. ConDec and DecSerFlow are very similar. However, DECLARE could be used to define completely different languages with different constraint templates, symbols, etc.

Figure 5 shows that DECLARE indeed allows for the definition of different languages. Here the link between a language (in this case ConDec) and constraint templates (in this case the list starting with "response") is established. Note that in this paper, we often use the term "DECLARE" to refer to the ConDec language realized using the DECLARE system.

B. Process Modeling

In Section II-A we described five constraints: `C.1-5`. Now we add these constraints to the model of Figure 3 by using constraint templates. To do this we need to map the textual descriptions given in Section II-A onto predefined constraint templates. A constraint can be added to the model of Figure 3 by selecting the appropriate template from the drop-down list of templates, and dragging it between the related activities. The result is shown in Figure 6. The DECLARE process model in Figure 6 consists of activities and some constraints between activities. Each of the constraints in Figure 6 represents one of the hotel rules mentioned earlier (`C.1-5`).

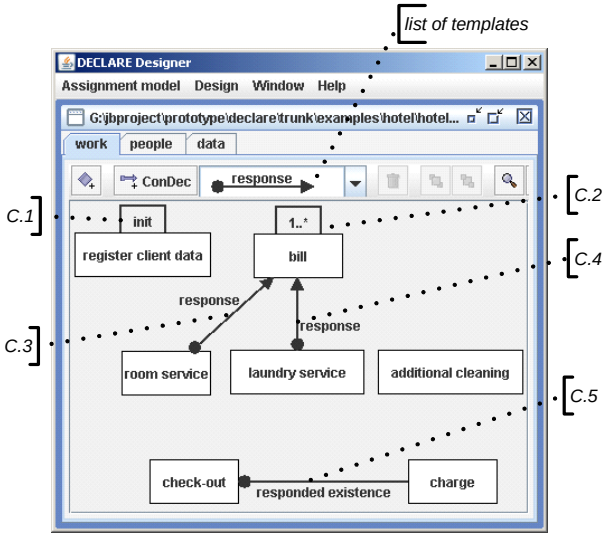


Fig. 6. Constraints in the hotel example

The number of constraints in a model is arbitrary. As indicated before, the model of Figure 3, which did not contain any constraint, is already an executable model. By adding constraints to the model, we impose rules that users have to follow during execution. Clearly, these rules limit the allowed behavior. In the next paragraph we describe the semantics of each constraint depicted in Figure 6.

Because of the constraint "init" on the activity "register client data" (rule (`C.1`)), users have to start each execution with the activity "register patient". It is still possible to execute this activity multiple times and at any moment during the execution to change client data at later stages (e.g., the

preferred way of paying is changed).

Before adding the "1..*" constraint on the activity "bill" (rule (C.2)) it was possible to never execute this activity and still complete the case. After adding the constraint, it became necessary to execute this activity at least once. This constraint enables the receptionist to add various items to the bill (e.g., room service, additional cleaning, etc.) at any moment during the execution.

With the "response" constraint between activities "room service" and "bill" (rule (C.3)) it is obligatory that after every execution of the activity "room service" at least one execution of the activity "bill" follows. The constraint allows execution of other activities between activities and "room service" and "bill". For example, it is possible that after the "room service" first "register client data" is executed and only afterwards "bill". The same holds for the "response" constraint between activities "laundry service" and "bill" (rule (C.4)). The two "response" constraints also allow users to wait and "bill" at once several "room services" and "laundry services".

Constraint "responded existence" between activities "check-out" and "charge" (rule (C.5)) specifies that if "check-out" was executed then "charge" must have been executed before or must be executed after "check-out". Other activities can be executed between activities "check-out" and "charge".

A process model containing multiple constraints is defined as a conjunction of the constraints, i.e., actions of users during execution must fulfill all the constraints.

C. Mandatory and Optional Constraints

DECLARE supports two types of constraints: *mandatory* and *optional* constraints. The system forces its users to follow all mandatory constraints in the model. In Figure 6 all constraints are mandatory constraints. In case of optional constraints users may decide whether to follow the corresponding rule or to violate it.

For example, in the hotel example every "room service" and "laundry service" is billed because they impose additional costs for the hotel. Although "additional cleaning" imposes additional cost, it is not necessarily billed. Suppose that management of the hotel noticed that in some cases "additional cleaning" is a consequence of irresponsible behavior. If this is the case, then the costs of cleaning should be billed. It is up to the receptionist to decide in which cases "additional cleaning" should be billed and in which not. This rule can be implemented as an *optional* "response" constraint between activities "additional cleaning" and "bill". Figure 7 shows this rule as an optional constraint. Note that the line is dashed to indicate that it is optional.

Optional constraints are not enforced by DECLARE system during execution. When a user is about to perform an action that violates an optional constraint, a warning about the violation is presented and the user can decide whether to continue with the action and violate the constraint or to cancel the action and follow the constraint. The text of the warning can be specified in the definition of the constraint. Figure 8 shows the form for defining a constraint in DECLARE.

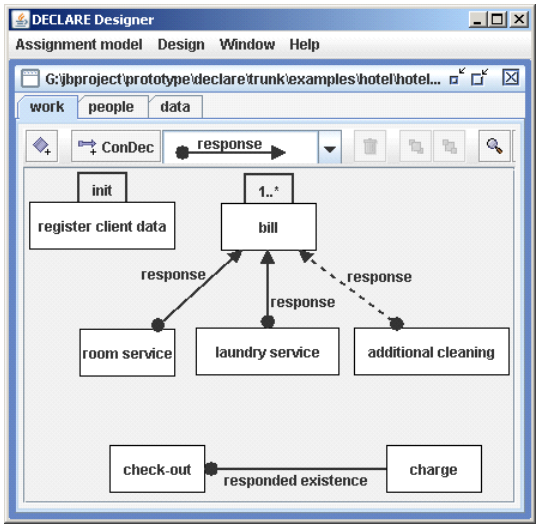


Fig. 7. Mandatory and optional constraints

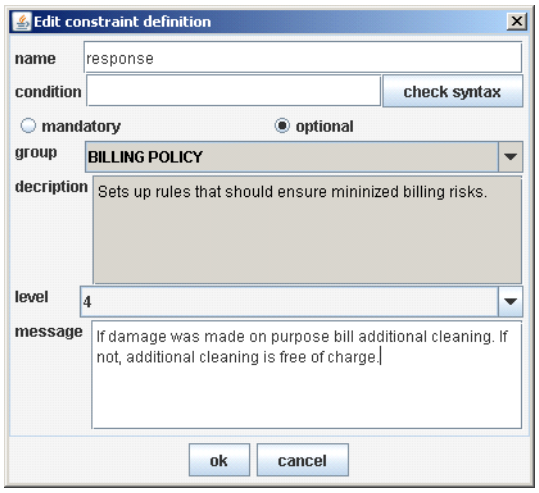


Fig. 8. Settings for optional constraint

The form contains the name of the constraint. By default this is the name of the template, but this name can be changed. Also, a condition for the constraint can be specified (e.g., a constraint should hold only if "$price < 1000$"). Moreover, the constraint is either mandatory or optional. If the constraint is optional, the information presented to users needs to be specified. Groups of constraints represent policies and can be defined on the system level in DECLARE by specifying a name and description for each group. For example, there could be groups like "Tourism Ministry Policy", "Hotel Policy", "Personnel Policy", "Billing Policy", etc. The appropriate group needs to be selected for each optional constraint. The importance of the constraint is given by the "level" on a scale 1-10. The higher the level is, the more dangerous it is to violate the constraint. Finally, a context-related message is specified that gives more detailed instructions to users.

Figure 9 shows the warning that a user will get when she is about to close a case where activity "additional cleaning"

was not followed by activity “bill”. This warning contains information about the billing policy, the violation level and an advising message to help the user to decide weather to violate this constraint or not.

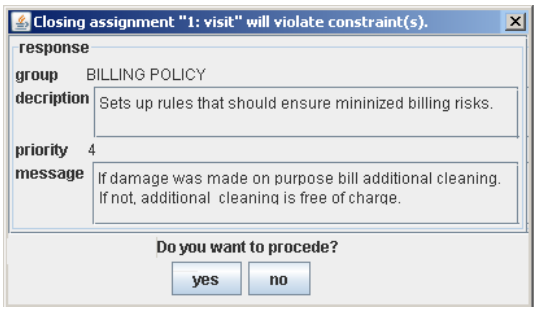


Fig. 9. Violation warning for optional constraint

D. Process Execution

A model in DECLARE is mapped onto a set of LTL formulas. Based on these LTL formulas, automata are automatically generated [16] to support enactment. Many algorithms that generate automata from LTL formulas have developed and these are widely used in the field of model checking [20]. DECLARE uses an algorithm that creates finite-words automata [17] from LTL formulas of the constraints that are used. These automata are used both to drive the execution and to monitor the state of each constraint.

After process model is loaded in the *Framework* tool (cf. Figure 1), users can execute the model in their *Worklists*. Figure 10(a) shows the initial Worklist for the hotel example. A list of all running cases (process instances, assignments) is shown on the left side of the screen. The process model of the selected case is shown on the right side of the screen. After the user starts an activity by double-clicking it, the activity is opened in the panel under the model. Although the structure of the process model is the same as in the Designer, the Worklist uses some additional symbols and colors to help users to understand the current state of the model, the activities and the constraints.

First, each activity contains “start” (play) and “complete” (stop) icons, that indicate if users can start/complete the activity at the moment. The initial state of the process instance in Figure 10(a) shows that it is only possible to start activity “register client data”, because the corresponding symbol is enabled. Starting and completing any of the other activities is not possible, as indicated by the disabled icons. In addition to the two icons, all currently disabled activities are colored grey. This initial state of the process instance is caused by the “init” constraint on the activity “register client data”, i.e., this activity is the first activity to be executed.

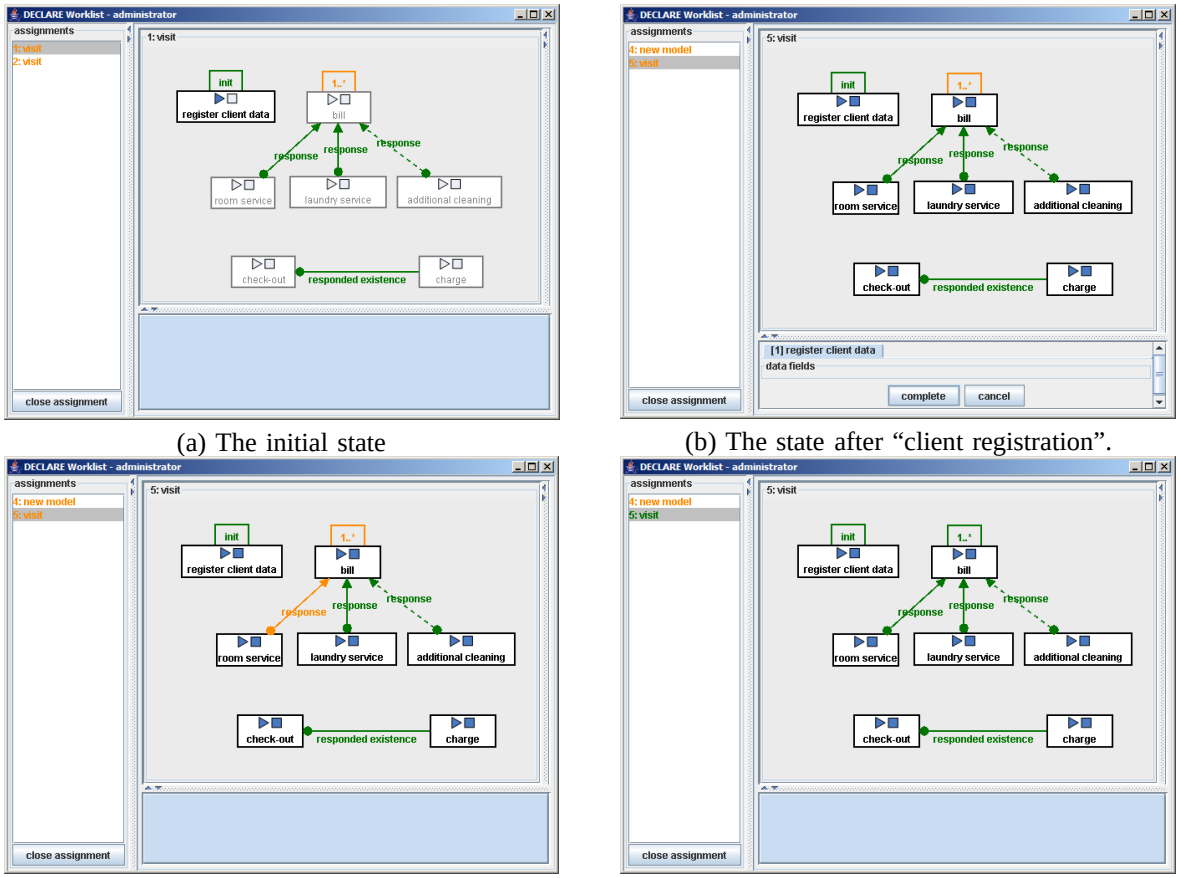
Second, each constraint is colored to indicate its state. Constraints are rules that should be fulfilled at the end of the execution. However, it is not realistic to expect that each constraint is fulfilled at each moment of time during the whole execution. Generally, at any moment each constraint can be in one of the following states: (1) *fulfilled* – constraint is represented by a *green* color, (2) *temporarily violated*, i.e., it can be fulfilled in the future – constraint is represented by a *orange* color, and (3) *permanently violated*, i.e., it cannot be fulfilled in the future – constraint is represented by a *red* color. For example, when the process instance of the hotel example is started (before executing any activity), constraint “1..*” on the activity “bill” is not fulfilled because “bill” was never executed. However, it is only temporarily violated, i.e., it can be fulfilled later (when the activity “bill” is executed for the first time). Therefore, it presented using an orange color. All other constraints are fulfilled at this moment, as can be seen from their green color. Naturally, DECLARE will prevent users from permanently violating mandatory constraints, i.e., only optional constraints can be “red”.

Figure 10(b) shows the state of the case after starting activity “register client data”. Three observations can be made here. First, the activity is now open in the panel under the model. Second, now it is possible to start other activities in the case. Third, only constraint “1..*” of the bill is not yet fulfilled and colored in orange. This constraint can be satisfied by executing activity “bill”. Figure 10(c) shows the state of the case after executing “room service”. The “response” constraint between this activity and activity “bill” becomes temporally violated, since it requires the execution of activity “bill” in the future. This is indicated by coloring the constraint orange. Executing activity “bill” results in the fulfilment of two constraints: (1) the “response” constraint between activities “room service” and “bill” and (2) the “1..*” constraint on the activity “bill”, as shown in Figure 10(d).

Depending on the state of its *mandatory* constraints, a process instance has its own state, as shown by the color of the instance in the list on the right side of the four Worklist windows in Figures 10. The process instance is in the “green” state if all its mandatory constraints are “green”. If at least one mandatory constraint is “orange”, the process instance is also “orange”.

E. Changing Process During Execution

Adaptivity is an important feature of flexible WFMSs [29]. Adaptive systems allow changing the process model during its execution. Adaptivity is one of the main features of DECLARE; it is possible to change its declarative process models during execution. Not only it is possible to add, delete (together with relating constraints) and change (data elements used in) activities, but it is also possible to add, remove and change (e.g., make optional, change condition) constraints. Before confirming an adaptation, DECLARE verifies the compliance of the changed model and instance history, i.e., history based errors are detected (cf. Section II-F). After the adaptation, the changed model is re-initiated with the procedure that is also used to start the process instance. The required automata are again generated for the new set of constraints and the history of the instance is replayed on these new automata.



(a) The initial state

(b) The state after "client registration".

(c) The state after "room service".

(d) The state after "bill".

Fig. 10. Execution of the Hotel example

F. Verification of Process Models

The addition of constraints to a process model in DECLARE may cause errors that lead to problems at run-time. Therefore, DECLARE verifies process models against three types of errors and finds a minimal set of constraints that causes the error. All models can be verified against *dead activities* and *conflicting constraints*. In addition to this, when a model is altered during its execution, it can be verified against *history-based errors*.

1) Dead activities: A dead activity is an activity that can never be executed in the model. Figure 11 shows the hotel example with one additional constraint – the "responded absence" constraint between activities "check-out" and "charge" specifies that if activity "check-out" is ever executed, then activity "charge" must never be executed (neither before or after "check-out"). If activity "check-out" would be executed in the model, it would not be possible to fulfill both constraints "responded existence" and "responded absence" between activities "check-out" and "charge". Therefore, activity "check-out" is a dead activity, i.e., it will never be possible to execute this activity.

DECLARE will detect this error during verification as shown in Figure 12. On the left part of the screen a list of

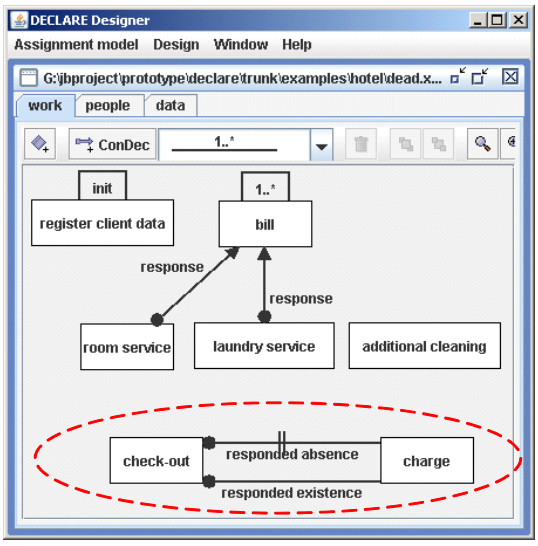


Fig. 11. Activity "check-out" is dead

detected errors is shown. In this case, one "dead activity" error is detected for activity "check-out". The list on the right side of the screen shows the minimal set of constraints that causes

the selected error.

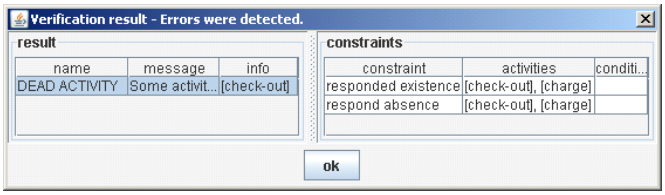


Fig. 12. Verification result for dead activity

2) Conflicting constraints: A set of constraints is conflicting if there exists no execution that would fulfill all constraints. If a constraint specifying that activity "check-out" has to be executed at least once would be added to the model in Figure 11, the result would be a process model with a conflict, as shown in Figure 13. This is because there exists no execution that would satisfy the following three constraints: "1..*" on activity "check-out", "responded existence" between activities "check-out" and "charge", and "responded absence" between activities "check-out" and "charge".

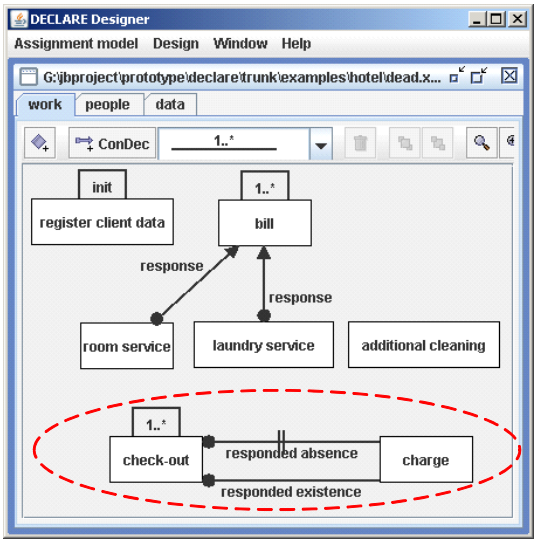


Fig. 13. Conflict

Figure 14 shows the conflicting error that was detected in DECLARE during verification.

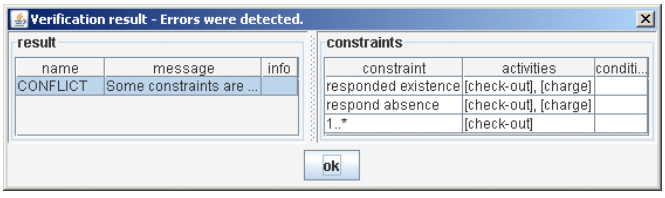


Fig. 14. Verification result for conflict

3) History based violations: As described in Section II-E, DECLARE models can be changed during the execution. Changes (especially adding new constraints) can be conflicting with the history of the case. For example, assume that activities "register client data", "bill", and "check-out" are executed in the current process instance. At this point the receptionist decides that client should not "check-out" before activity "charge" is executed, and adds a "precedence" constraint between activities "check-out" and "charge" (see Figure 15).

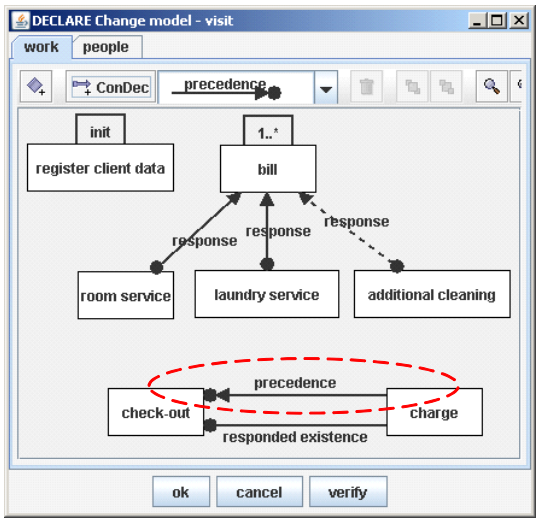


Fig. 15. History violation

This adaptation is in conflict with the history of the case, because activity "check-out" is already executed before activity "charge". DECLARE will detect this error and inform the user that the new constraint causes a history-based error, as shown in Figure 16.

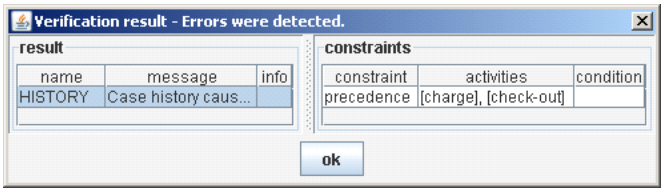


Fig. 16. Verification result for history violation

III. Combining DECLARE and YAWL

This section shows how DECLARE and YAWL can be combined to support arbitrary mixtures of loosely-structured and highly-structured processes.

DECLARE is not particularly suitable for modeling large and/or highly-structured processes. In both cases, a DECLARE model would have many constraints, which can easily create problems. First, errors can be easily introduced during process development when the number of constraints is high. Second, it is hard for users to understand the whole model during execution if the model has too many constraints. In addition, the performance of the system is poot for models with many constraints, because the automata become too large to be handled efficiently. Therefore, we propose using the YAWL [1] workflow management system in combination

with DECLARE. YAWL can easily deal with large highly-structured processes and its service-oriented architecture allows for an easy integration.

We propose YAWL for highly-structured processes and DECLARE for loosely-structured ones. The decomposition of processes using DECLARE and YAWL can be two-fold, i.e., a DECLARE model can be a sub-process or super-process of YAWL model(s). Figure 17 shows an example of a decomposition of a large hotel process. First, a highly-structured global process for managing rooms in a hotel is developed in YAWL – on the first level of decomposition a room is "booked", "cleaned", "rented", and then "cleaned" again. At this level, we decompose the task[3] "rent" to our hotel example in DECLARE (the second level). Within this DECLARE model, we can specify that activity "room service" should invoke another highly-structured process in YAWL, where after the "order" is taken, it is "prepared", "delivered" and "registered" (the third level).

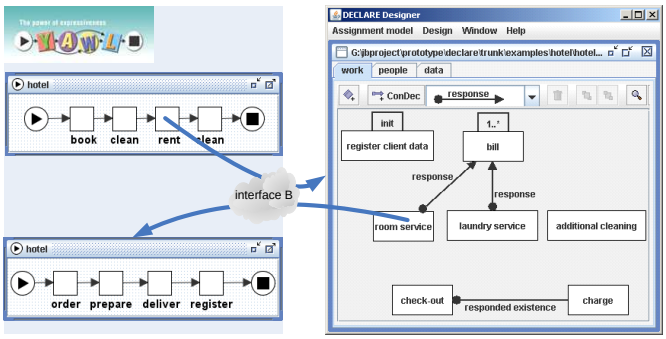


Fig. 17. A DECLARE model serving both as a sub-process and super-process for YAWL

To achieve this decomposition, DECLARE communicates with YAWL via its "interface B". YAWL is developed using a service oriented architecture where a YAWL process can serve both as a service consumer and as a service provider. Tasks in YAWL may be subcontracted to another service. This way YAWL acts as a service consumer. In the context of YAWL, several services have been developed. For example, the default Worklist handler is an example of a service than can communicate with YAWL via "interface B". Other services are the SMS service, Worklet service [7], etc. From the viewpoint of YAWL, DECLARE is just another service that YAWL can use. Moreover, YAWL can also act as a service provider for DECLARE, i.e., an activity in DECLARE can be subcontracted to YAWL by initiating a new process instance. This can be mixed an arbitrary ways, e.g., YAWL may subcontract a task to DECLARE, in the corresponding DECLARE process an activity is subcontracted to YAWL, in the corresponding YAWL process a task is subcontracted to the Worklet service, etc. This allows for arbitrary mixtures of highly-structured processes (YAWL), loosely-structured processes (DECLARE), emerging/rule-based processes (e.g., Worklets [7]), etc.

[3]For clarity we use different terms for the smallest unit of work in YAWL and DECLARE. The term *task* is used to denote the smallest unit of work YAWL, i.e., a task is not decomposed further in YAWL but may refer to a DECLARE process. DECLARE uses the term *activity* for the smallest unit of work.

IV. Analysis of Past Executions

WFMSs can execute a variety of process instances over time. Most systems record detailed logs about all completed executions. Data stored in such logs can be various: starting cases; starting, completing, canceling activities; changing value of data elements; deadline expiry, etc. These logs can be used for discovering and analysis of executed process models – *process mining* [6]. Process mining tools (e.g., ProM [11]) use various techniques of log analysis to discover the process model, verify certain properties of the model, discover the social network, etc.

DECLARE stores all events related to activities in execution logs using the MXML format [10]. This format is also used by the ProM tool and thus this export allows for all kinds of analysis techniques ranging from locating bottlenecks in a process to constructing a social network for the actors involved. Amongst others, ProM has a feature that can be used to verify various properties of executions stored in logs – the *LTL Checker*. LTL checker enables verification of logs against properties specified in Linear Temporal Logic. For example, it is possible to verify if a "junior officer has approved a claim worth more than 10000 this year". DECLARE enables two types of export to LTL Checker readable files, as shown in Figure 18.

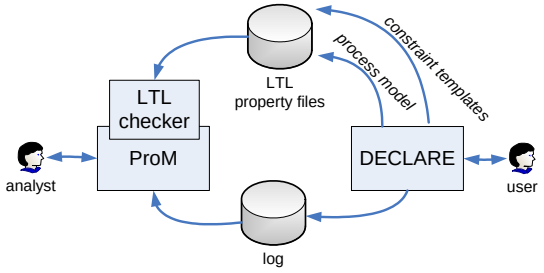


Fig. 18. DECLARE is able to export event logs, models, and constraint templates to ProM

First, constraint templates can be exported to LTL Checker files from DECLARE. These files can be used to verify properties in logs in an generic way, e.g., the "response" property (template) can be checked against different pairs of activities. Second, a DECLARE process model can be exported to an LTL Checker file. Using this file, process logs can be verified against constraints from the process model, e.g., the "response" property (existing constraint) can only be checked against existing activities "room service" and "bill".

V. User Support by Recommendations

Despite many benefits that flexibility brings, its major drawback it the lack of *support* that users get in flexible systems. Support of a WFMS can be seen as an extension in which the system is able to make decisions for the user. Figure 19 shows flexibility and support as two opposing properties, i.e., rigid

systems provide support by sacrificing flexibility, and flexible systems provide flexibility by sacrificing support. The variety of options in flexible systems makes it difficult for users to make the right decision. For example, an inexperienced user, or a user working on an exceptional case will find it difficult to decide between many options, and would greatly benefit from support.

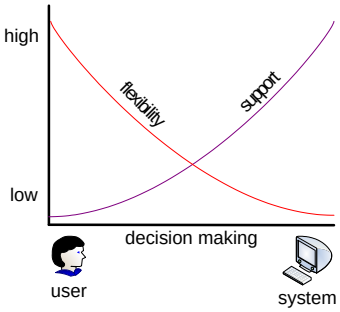


Fig. 19. Trade-off: flexibility vs. support [10]

Support for flexible systems should focus on offering *recommendations* for decisions, rather than taking these decisions for the user. Recommendations are generated based on past experiences and a specific goal. DECLARE stores past experiences in MXML format. Based on the goal of a user/organization, past experiences can be rated in terms of their desirability. For example, the goal may be to minimize throughput time and, therefore, cases which were handled quickly are considered positive examples. The recommendation service of ProM [11] generates recommendations for DECLARE based on the comparison of the current process instance (*partial instance*) with past executions (*logs*), while favoring those executions that satisfy the specified *goal*, as shown in Figure 20. Currently, various recommendation algorithms have been implemented in ProM but outside the scope of this paper.

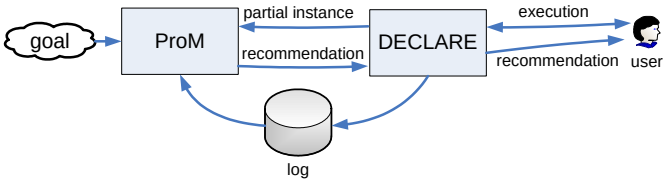


Fig. 20. ProM as recommendation provider for DECLARE

DECLARE does not enforce recommendations to users. On the contrary, recommendations are presented to users as independent information, as shown on the right side of the Figure 21. The user can choose to follow or not to follow the recommendation to “start activity bill”.

VI. Related Work

Many approaches aim at “relaxing” the rigid nature of traditional process modeling languages and workflow management systems. These process models precisely prescribe how the process should be executed and workflow systems force users to execute these models step-by-step.

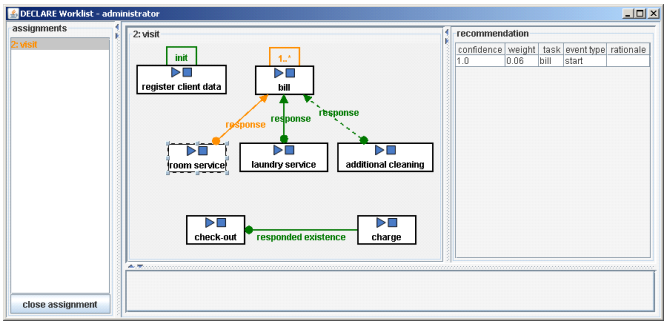


Fig. 21. Recommendation is to start activity “bill”

Approaches like case-handling and adaptive systems change the way the system manages the execution of rigid models. An example of a case-handling system is FLOWer [25]. FLOWer does not enforce a strict execution of process models, but allows users to open tasks that should be executed later (according to the model), re-do tasks that were executed before or even skip tasks that should be executed. When working with adaptive systems like ADEPT [29], users can change the process model while executing the model by adding, moving or deleting activities in the model. Both approaches use imperative models and they consider variations in executions to be exceptions, which can have negative consequences. For example, if a user of a case-handling system wants to re-do an activity that was already executed before, she will also have to execute all activities that followed it. Frequently changing process models in adaptive systems are time consuming and require users to be experts in process modeling. DECLARE uses a different approach that does not require users to redesign the process to deviate from the normal flow. Deviations are not seen as exceptions and are included the allowed behavior. Moreover, unlike case handling, there is not a fixed set of automatically included deviations (e.g., skip and redo).

Loosely-structured process can be handled using declarative languages, which “describe the dependency relationships between tasks, rather than procedurally describing sequences of action” [12]. DECLARE is not the first attempt to use a more declarative language [12], [23], [30]. Instead of modeling a detailed control-flow, declarative languages propose modeling constraints that (as rules that should be followed) drive the model enactment [12], [23], [30]. Constraints describe dependencies between model elements and are specified using pre and post conditions for target task [30], dependencies between states of tasks (enabled, active, ready, etc.) [12] or various model-related concepts [23]. DECLARE distinguishes itself from these earlier approaches in many respects. For example, DECLARE is based on LTL, it does not use a fixed language and users can extend the language, it supports optional and mandatory constraints, it supports verification, it can supports on-the-fly model changes, and it is equipped with a recommendation service.

This paper builds on two papers: [26] and [5]. [26] presents the ConDec language and [5] introduces the DecSerFlow lan-

guage. The first language is tailored towards teamwork while the second is tailored towards the specification of services. These two papers do not describe the DECLARE tool in any detail. In fact most of the functionality described in this paper, was realized after the publication of [5], [26]. Moreover, the innovative features of DECLARE in relation to process mining, recommendation, verification, optional constraints, and model change have not been described before. This illustrates the original contribution of the current paper.

VII. Future work

Currently, DECLARE uses a simple constraint specification approach that considers only events regarding execution of activities (the control-flow). This can be extended by using other process model elements (like user roles, data elements, etc.) in the constraint specification. For example, it can be necessary to specify a constraint that prevents one user to execute two crucial activities in the process model (the so-called "four eyes principle"), e.g., it is not possible that the same person who filed a request for salary rase approves this request. Here these is also a link to the topic of semantical correctness presented in [22].

Another interesting extension would be adding deadlines in DECLARE process models. For example, the "response" constraint template can be extended with a deadline: "A" has to be followed with "B" within five days. To introduce deadlines, a logic extended with time dimension and time automata can be used (e.g., Extended Timed Temporal Logic [8]).

DECLARE is currently being extended to support constraint templates with multiple parameters, instead of only one or two parameters. This will enable creating more advanced templates and constraints involving more that two activities.

VIII. Conclusions

The DECLARE system supports loosely-structured processes without sacrificing useful features that traditional workflow management systems have. DECLARE uses a temporal logic (LTL) as a basis and combines this with an extendible graphical language. In fact, DECLARE can support multiple languages in parallel and end users can make domain specific languages. To support enactment, DECLARE automatically constructs automata to guide (or force) to user.

DECLARE connects to the workflow management system YAWL and the process mining tool ProM. Through YAWL, it becomes possible to support large processes containing mixtures of loosely-structured and highly-structured fragments. The connection to ProM allows for the analysis of processes supported by DECLARE. Moreover, using ProM's recommendation service it is possible guide users based on past experiences. This way flexibility and learning are combined in a powerful manner.

The DECLARE provides many innovative features and can be downloaded from http://is.tm.tue.nl/staff/mpesic/declare.htm.

References

[1] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004.

[2] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.

[3] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[4] W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.

[5] W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Berlin, 2006.

[6] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[7] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems, OTM Confederated International Conferences, 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308, Berlin, 2006. Springer-Verlag.

[8] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model-checking for extended timed temporal logics. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326, London, UK, 1996. Springer-Verlag.

[9] C. Bussler, S. Jablonski, and H. Schuster. A new generation of workflow-management-systems: beyond taylorism with mobile. *SIGOIS Bull.*, 17(1):17–20, 1996.

[10] B.F. van Dongen and W.M.P. van der Aalst. A Meta Model for Process Mining Data. In J. Casto and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*, volume 2, pages 309–320. FEUP, Porto, Portugal, 2005.

[11] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, Lecture Notes in Computer Science, pages 444–454. Springer-Verlag, Berlin, 2005.

[12] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw. Freeflow: mediating between representation and action in workflow systems. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 190–198, New York, NY, USA, 1996. ACM Press.

[13] M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems*. Wiley & Sons, 2005.

[14] D. Georgakopoulos. Teamware: An evaluation of key technologies and open problems. *Distributed and Parallel Databases*, 15(1):9–44, 2004.

[15] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[16] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.

[17] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.

[18] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A comprehensive approach to flexibility in workflow management systems. In *WACC '99: Proceedings of the international joint conference on Work activities coordination and collaboration*, pages 79–88, New York, NY, USA, 1999. ACM Press.

[19] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.

[20] E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

[21] M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, volume 9 of *Special issue of the journal of Computer Supported Cooperative Work*, 2000.

[22] L. Thao Ly, S. Rinderle, and P. Dadam. Semantic correctness in adaptive process management systems. In S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2006.

[23] P. Mangan and S. Sadiq. On building workflow models for flexible processes. In *ADC '02: Proceedings of the 13th Australasian database conference*, pages 103–109, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[24] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[25] Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.

[26] M. Pesic and W.M.P. van der Aalst. A Declarative Approach for Flexible Business Processes. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, Workshop on Dynamic Process Management (DPM 2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, Berlin, 2006.

[27] H. Reijers, J. Rigter, and W.M.P. van der Aalst. The Case Handling Case. *International Journal of Cooperative Information Systems*, 12(3):365–391, 2003.

[28] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[29] S. Rinderle, M. Reichert, and P. Dadam. Flexible Support of Team Processes by Adaptive Workflow Systems. *Distrib. Parallel Databases*, 16(1):91–116, 2004.

[30] J. Wainer and F. de Lima Bezerra. *Groupware: Design, Implementation, and Use*, volume 2806, chapter Constraint-Based Flexible Workflows, pages 151 – 158. Springer Berlin / Heidelberg, 2003.