Chapter 12
The Declare Service

Maja Pesic, Helen Schonenberg, and Wil van der Aalst

12.1 Introduction

The Declare Service is a YAWL Custom Service that enables decomposing YAWL tasks into DECLARE workflows, i.e. workflows supported by the workflow management system (WfMS) called DECLARE. The goal of this service is to enable a particular kind of flexibility. Chapter 6 describes a constraint-based approach to workflow models and the ConDec language. This approach, supported by the DECLARE WfMS, allows for more flexibility, i.e. execution of tasks is allowed if it is not explicitly forbidden by some constraint. This chapter describes DECLARE and the Declare Service for YAWL.

Sometimes it is easier to express a process in a procedural language (e.g. the native workflow language of YAWL) and sometimes a declarative approach is more suitable. Moreover, in a larger process it may be useful to express parts of the process in a procedural language and specify other parts in terms of constraints. Using the service-oriented architecture of YAWL, this can easily be realized. A YAWL task may decompose into a DECLARE process and a task in DECLARE can be decomposed into a YAWL process. Arbitrary decompositions of DECLARE and YAWL models allow for the integration of declarative and YAWL workflows on different abstraction levels.[1] This way the designer is not forced to make a binary choice between a declarative and a procedural way of modeling. Hence, a seamless integration can be achieved, where parts of the workflow that need a high degree of flexibility are supported by declarative DECLARE models and parts of the processes that need centralized control of the system are supported by YAWL models.

Consider, for example, the decomposition of the Carrier Appointment process shown in Figure 12.1. At the highest level of decomposition, the main process is modeled using a procedural YAWL *Carrier Appointment* net, which is described in Appendix A. However, the parts of the net that refer to the Truck Load and Less than

[1] Note that the service oriented architecture also allows for decompositions involving worklets, which are described in Chapters 4 and 11.

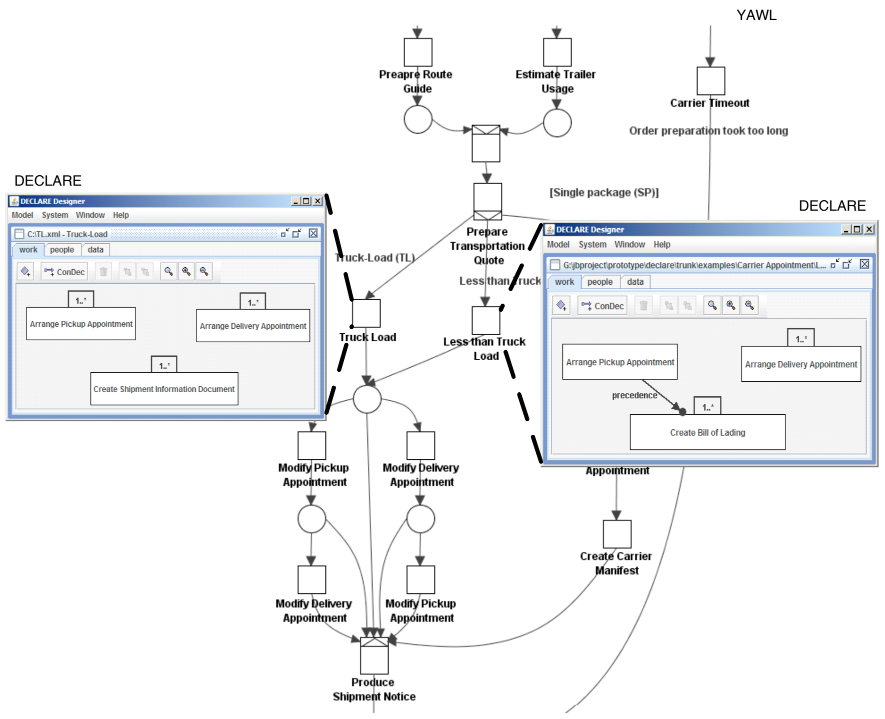
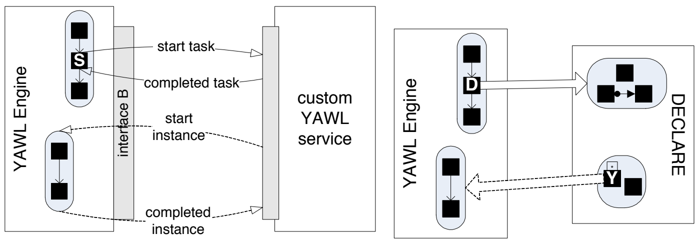


Fig. 12.1: An example illustrating the service-oriented architecture: tasks *Truck Load* and *Less than Truck-Load* in the top-level YAWL net are decomposed into DECLARE processes

Truck Load scenarios are now decomposed and specified as declarative workflows, as described in Chapter 6. Although this example only presents decomposition of YAWL nets into DECLARE sub-models, it is also possible to decompose DECLARE models into YAWL sub-nets. This way, procedural and declarative workflows can be combined.

12.2 Service Architecture

As Figure 12.2 shows, the Declare Service uses Interface B in two ways. First, YAWL can delegate a task to the service, instead of to the Resource Service. For example task *S* in Figure 12.2(a) can be delegated to a Custom Service. Second, a Custom Service can request the launch of a new YAWL instance. In both cases, relevant instance and task data elements are exchanged between YAWL and the service. The Resource Service is a Custom Service, which is described in Chapter 10. By default, all tasks in YAWL models are delegated to the Resource Service, i.e. all tasks are by default executed by users through their YAWL worklists. If a task should be delegated to another Custom Service, then this must be explicitly specified in the model. The Declare Service enables arbitrary decompositions of DECLARE and models in two ways, as shown in Figure 12.2(b). First, it is possible that a YAWL task triggers the execution of a DECLARE instance: when the YAWL task *D* becomes enabled, DECLARE will initiate the required constraint model. YAWL will consider the completion of the launched DECLARE instance as a completion of its task. Second, DECLARE task *Y* can trigger the execution of a YAWL instance.



(a) Custom Service interface (b) DECLARE as a service

Fig. 12.2: DECLARE as a YAWL Custom Service

As Figure 12.3 shows, DECLARE consists of three components; the *Designer*, the *Framework* and the *Worklist*. The *Designer* component is used for creating constraint templates, constraint models, and by verification of constraint models. For example, Figure 12.1 shows the *Designer* component containing the Carrier Appointment model. The *Framework* component facilitates instance creation and change of constraint models. Finally, the *Worklist* component offers users access to active instances of constraint models. Also, users can execute tasks of active instances in their *Worklist*. The Declare Service acts as a bridge between the YAWL Engine and the DECLARE *Framework*, as shown in Figure 12.3. Note that users execute 'normal' tasks (i.e. tasks that are not decomposed or subcontracted to another system or process) of DECLARE and YAWL instances in a default manner by using the respective worklists.

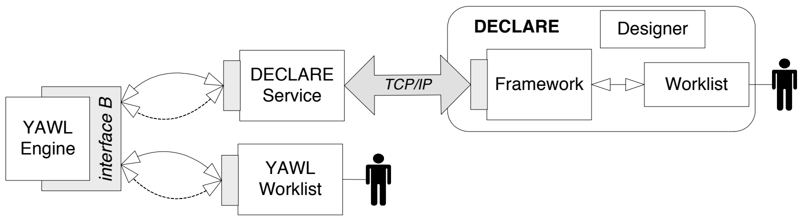


Fig. 12.3: Interface between YAWL, Declare Service and DECLARE

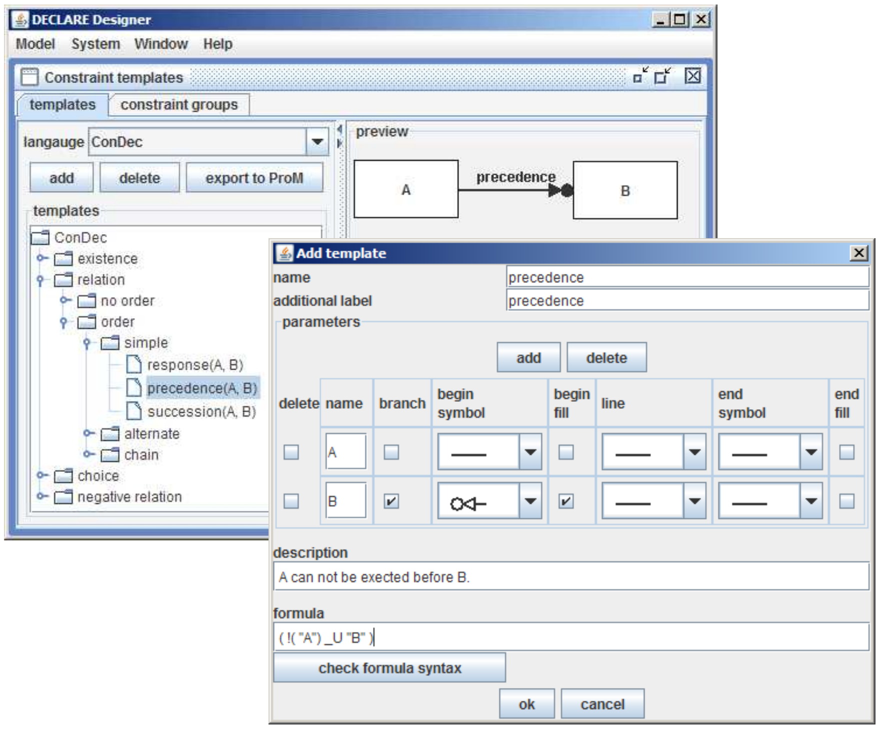
After presenting the architecture of DECLARE and the way it is embedded into the YAWL environment, we now focus on the functionality provided by DECLARE. In the remainder, we show the following capabilities:

- The possibility to define *constraint templates*. DECLARE is multi-lingual and can support multiple constraint languages at the same time. Moreover, it is possible to define new languages using constraint templates.
- The possibility to define *constraint models*. Using the constraint templates it is possible to specify specific constraints for a particular process.
- The *verification* capabilities of DECLARE. While modeling or redesigning workflows errors may be introduced. Hence, DECLARE offers a wide range of verification capabilities.
- The *enactment* of constraint model instances. Based on such constraint languages, the DECLARE can automatically generate the support needed, i.e. mandatory constraints can not be violated while optional constraints are used to generate warnings.
- The possibility to *change* constraint models and to *migrate* instances. Although DECLARE allows for a declarative way of modeling, other flexibility issues such as migrating cases are supported.
- The *integration* of YAWL and DECLARE. As explained using Figure 12.1 it is rather easy to support an arbitrary nesting of workflow languages.

12.3 Constraint Templates

An arbitrary number of constraint-based languages can be defined in DECLARE. For example, Figure 12.4 shows how the ConDec language (cf. Chapter 6) is defined in the *Designer* tool. A tree with the language templates is shown under the selected language. On the panel on the right side of the screen the selected template is presented graphically.

An arbitrary number of templates can be created for each language. Figure 12.4 shows a screenshot of the *Designer* while defining the *precedence* template. First, the template name and additional label are entered. Next, it is possible to define an arbitrary number of parameters in the template. The *precedence* template has two parameters: *A* and *B*. For each parameter it is specified whether it can be *branched* or not. When creating a constraint from a template in a model, a task replaces each of the template's parameters. If a parameter is branchable, then it is possible to replace the parameter with more tasks. In this case, the parameter will be replaced by a *disjunction* of selected tasks in the formula (cf. Chapter 6). The graphical representation of the template is defined by selecting the kind of symbol that will be drawn next to each parameter and the style of the line. Figure 12.4 shows that the *response* template is graphically represented by a single line with a filled arrow-and-circle symbol next to the second task (*B*). Furthermore, a textual description and the LTL formula are given.



langauge: ConDec

name: precedence

additional label: precedence

delete	name	branch	begin symbol	begin fill	line	end symbol	end fill
☐	A	☐	——	☐	——	——	☐
☐	B	✓	○◁—	✓	——	——	☐

description: A can not be exected before B.

formula: (!("A") _U "B")

Fig. 12.4: Creating a language by defining constraint templates using the DECLARE *Designer* tool

12.4 Constraint Workflow Models

Constraint models can be developed in the *Designer* tool after selecting one of the languages defined in the system. In this chapter we only use the ConDec templates described in Chapter 6. However, as explained before, DECLARE is extensible and multi-lingual. As an example, we consider the Less than Truck Load (LTTL) process described in Chapter 6. Figure 12.5 shows the ConDec LTTL model in DECLARE.

Each constraint in the model in Figure 12.5 is created using a ConDec template. For example, the constraint between tasks *pickup* and *bill* is created by applying the *precedence* template[2]. The creation of this template was already shown in Figure 12.4. However, when making a constraint model there is no need to define new constraints from scratch. Instead, constraints are defined by applying templates. Figure 12.6 shows the application of the *precedence* template. The template is selected in the top left corner of the screen. Underneath the template all its parameters are shown. Tasks are assigned to parameters by selecting a task (or multiple tasks in case

[2] For the purposes of simplicity, in this chapter we will use shorter names of the relevant tasks: *pickup* for *Arrange Pickup Appointment*, *delivery* for *Arrange Delivery Appointment*, *bill* for *Create Bill of Lading* and *shipment* for *Create Shipment Information Document*.

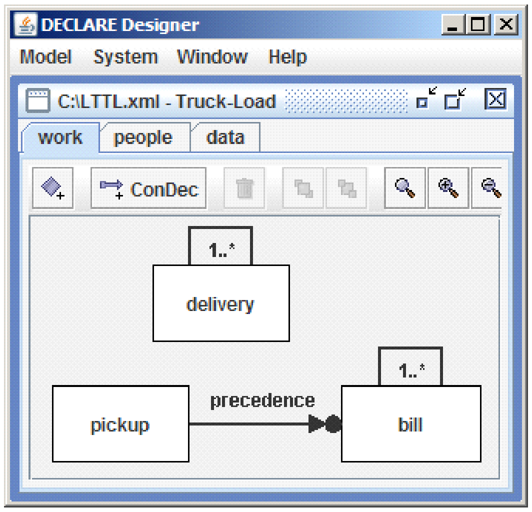
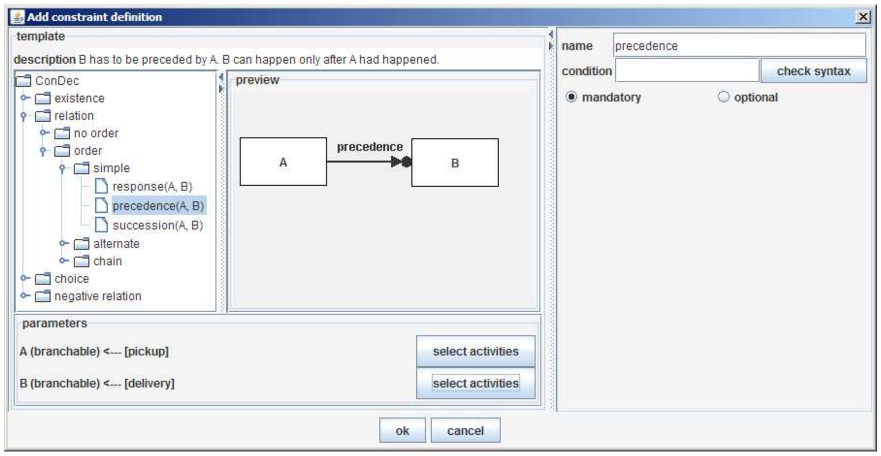


Fig. 12.5: The LTTL model in DECLARE



description B has to be preceded by A. B can happen only after A had happened.

A (branchable) <--- [pickup]

B (branchable) <--- [delivery]

name precedence

condition

[✓ mandatory] optional

Fig. 12.6: Defining a constraint in DECLARE based on the *precedence* template constructed in Figure 12.4

of branching) from the model. On the right side of the screen some additional information can be given. Initially, constraints have the template name, but constraints can have arbitrary names and can be renamed. Constraints can have a condition involving some data elements from the model. For example, condition “totalPrice < 1000” on a constraint would mean that the constraint should hold only if the data element *totalPrice* has a value less than 1000. Finally, the type (mandatory or optional) of the constraint must be specified. The constraint shown in Figure 12.6 is mandatory.

12.5 Verification of Constraint Models

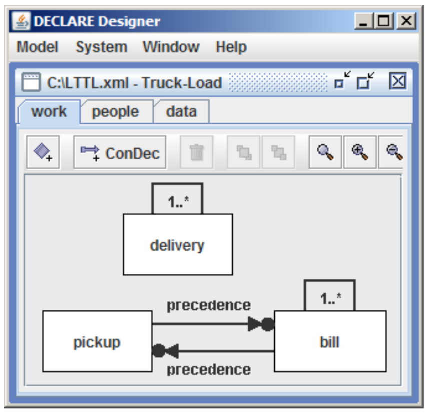
DECLARE uses the methods presented in Chapter 6 to detect dead tasks, conflicts and the smallest set of constraints that cause them. To illustrate this, we use the example for a ConDec model with a conflict from Chapter 6. Figure 12.7(a) shows this model in DECLARE: the two *1..** constraints specify that each of the tasks *delivery* and *bill* must be executed at least once. The two *precedence* constraints specify that (1) task *bill* cannot be executed before task *pickup* and (2) task *pickup* cannot be executed before task *bill*. Besides for detecting error(s) in models, for each error DECLARE searches through (the powerset) mandatory constraints to find the smallest group of constraints that causes the error. DECLARE applies the verification methods described in Chapter 6 for each group of constraints in order to detect the smallest group of constraints that causes the error.

Figures 12.7(b), (c) and (d) show that the DECLARE verification procedure reports three errors in this model. First, the two *precedence* constraints cause the *bill* task to be dead (cf. Figure 12.7(b)). Second, the two *precedence* constraints cause the *pickup* task to be dead (cf. Figure 12.7(c)). Finally, the two *precedence* constraints and the *1..** constraint on task *bill* cause a conflict (cf. Figure 12.7(d)). Because of this conflict it is not possible to satisfy all constraints at the same time. At least one of the three related constraints mentioned needs to be dropped.

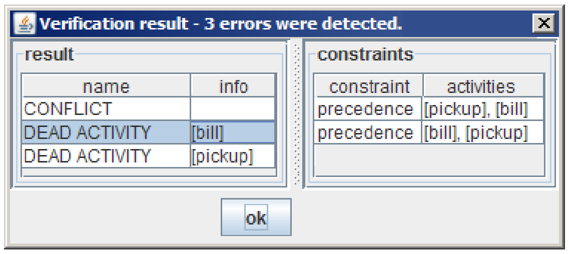
Detailed verification reports in DECLARE aim at helping model developers to understand error(s) in the model. The goal is to assist the resolution of such problems. As discussed in Chapter 6, errors can be eliminated from a model by removing at least one constraint from the group of constraints that together cause the error. For example, if one of the two *precedence* constraints would be removed from the DECLARE model in Figure 12.7(a), there would no longer be errors in this model. Also, removing the *1..** constraint on task *bill* would remove the conflict, but tasks *bill* and *pickup* would still be dead.

12.6 Execution of Constraint Model Instances

DECLARE determines which tasks can be executed next, i.e. determining the enabled tasks, the state of an instance and states of constraints using the approach presented in Chapter 6. Each instance is launched (i.e. created) in the *Framework* tool and users can work on instances via their *Worklists* (cf. Figure 12.3 on page 335). A user can work on instances from his/her *Worklist*. Figure 12.8 shows a screenshot of a *Worklist*. All available instances are shown in the list on the left side of the screen, underneath the 'instances' header. In Figure 12.8, there are two instances of the *Less*



(a) DECLARE model


Verification result - 3 errors were detected.

result

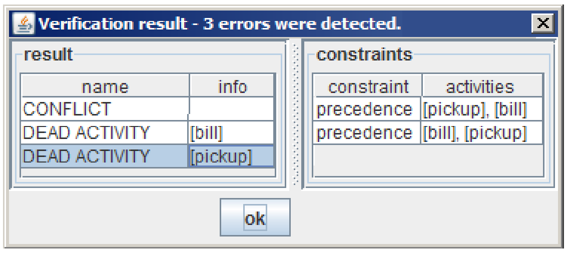
name	info
CONFLICT	
DEAD ACTIVITY	[bill]
DEAD ACTIVITY	[pickup]

constraints

constraint	activities
precedence	[pickup], [bill]
precedence	[bill], [pickup]

ok

(b) task *bill* is dead


Verification result - 3 errors were detected.

result

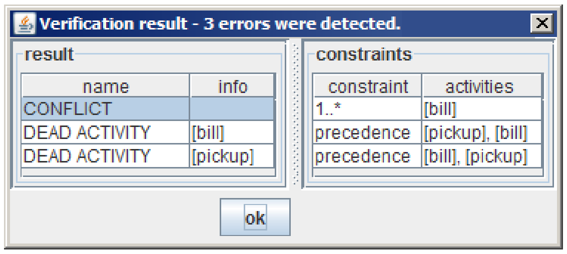
name	info
CONFLICT	
DEAD ACTIVITY	[bill]
DEAD ACTIVITY	[pickup]

constraints

constraint	activities
precedence	[pickup], [bill]
precedence	[bill], [pickup]

ok

(c) task *pickup* is dead


Verification result - 3 errors were detected.

result

name	info
CONFLICT	
DEAD ACTIVITY	[bill]
DEAD ACTIVITY	[pickup]

constraints

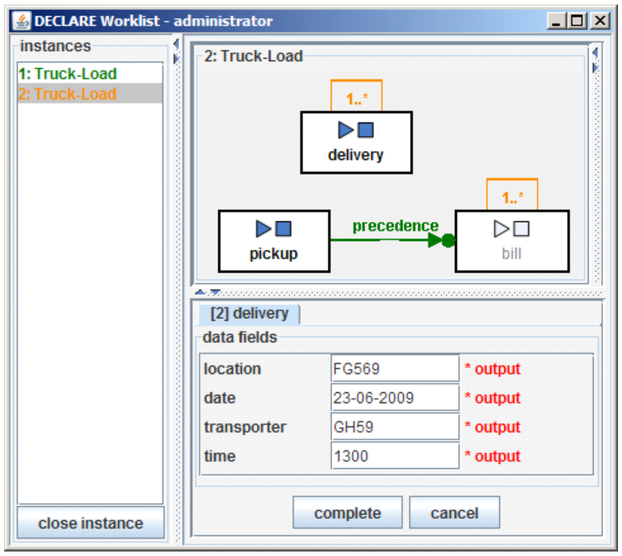
constraint	activities
1..*	[bill]
precedence	[pickup], [bill]
precedence	[bill], [pickup]

ok

(d) a conflict

Fig. 12.7: A DECLARE model with a conflict and two dead tasks

than Truck Load model presented in Figure 12.5 on page 338: ‘1: Less than Truck Load’ and ‘2: Less than Truck Load’. The model of the selected instance is shown on the right side of the screen. After the user starts a task by double-clicking it, the task will be opened in the ‘task panel’ under the model.



location	FG569	* output
date	23-06-2009	* output
transporter	GH59	* output
time	1300	* output

Fig. 12.8: The DECLARE *Worklist* showing details of a selected instance

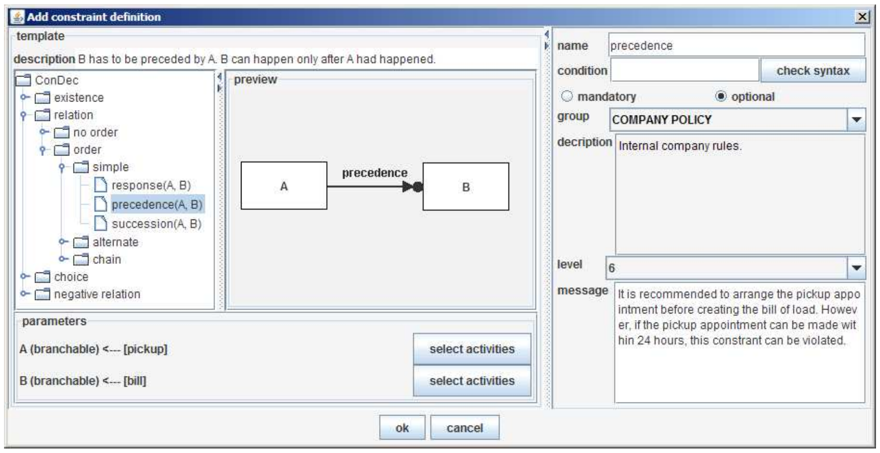
Although the structure of the process model is the same as in the *Designer* (cf. Figure 12.5), the *Worklist* presents some additional symbols and colors to users to indicate which tasks are enabled, the current state of the instance and each constraint. Each task contains ‘start’ (play) and ‘complete’ (stop) icons, that indicate whether users can start/complete the task at the moment. State *satisfied*, *violated* and *temporarily violated* are indicated by the colors green, red and orange, respectively. This color scheme is used for both constraint states and instance states.

The initial state of the process instance in Figure 12.8 shows that it is only possible to start tasks *delivery* and *pickup*, because the corresponding start symbols are enabled. Starting and completing task *bill* is not possible, as indicated by the disabled start icon. In addition, all currently disabled tasks are colored grey (cf. task *bill*). This initial state of the process instance is influenced by the *precedence* constraint, which specifies that task *bill* can not be executed until task *pickup* is executed. Also note that each constraint is colored to indicate its state. The two *1..** constraints are *temporarily violated* (i.e. *orange*), while the *precedence* constraint is *satisfied* (i.e. *green*).

Figure 12.8 shows the instance after starting task *delivery*. This task is now “open” in the ‘task panel’ on the bottom of the screen. Data elements that are used in this task are presented in the ‘task panel’. In this case, four data elements are available – patient *location*, *date*, *transporter* and *time*. In this way, users can manipulate data elements while executing tasks. The task can be completed or canceled by clicking on the appropriate buttons on the ‘task panel’.

12.7 Optional Constraints

So far in this chapter, we have used models where all constraints are mandatory. However, as discussed in Chapter 6, constraints can also be optional, i.e. they can be violated. If the user is about to violate an optional constraint, DECLARE issues a warning. This warning contains additional information about the constraint, and it should help the user when deciding whether to proceed and violate the constraint, or to abort and not violate it. As Figure 12.9 shows, additional information for optional constraints contains: (1) a group; (2) the importance level on a scale from 1 (for low importance) to 10 (for high importance); and (3) some local message that should be displayed. This additional information for an optional constraint is presented in the warning given when a user is about to violate this constraint.



description B has to be preceded by A. B can happen only after A had happened.

A (branchable) <--- [pickup]

B (branchable) <--- [bill]

name: precedence

condition:

mandatory [✓ optional]

group: COMPANY POLICY

decription: Internal company rules.

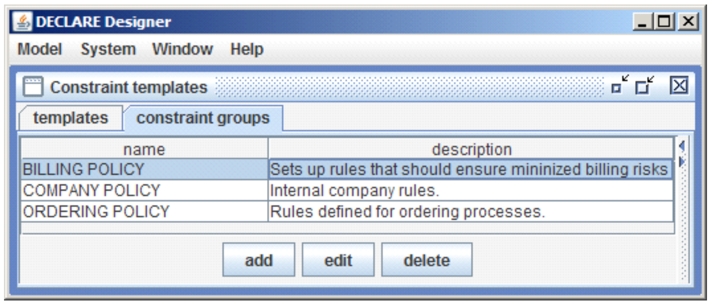
level: 6

message: It is recommended to arrange the pickup appointment before creating the bill of load. However, if the pickup appointment can be made within 24 hours, this constrant can be violated.

Fig. 12.9: Defining the optional constraint *precedence*

Groups for optional constraints are defined in the *Designer* component and can be used to define corporate policies, etc. Each group has a name and a short description, as illustrated by Figure 12.10, and end-users should be able to interpret them. For example, it should be easier to decide to violate a constraint belonging to the *Ordering Policy*, than a constraint that belongs to the *Billing Policy*.

In some cases, triggering an event or closing the instance can violate optional constraint(s). For example, consider an instance of the DECLARE model shown in Figure 12.11, where the user attempts to start executing task *bill*. Note that, despite



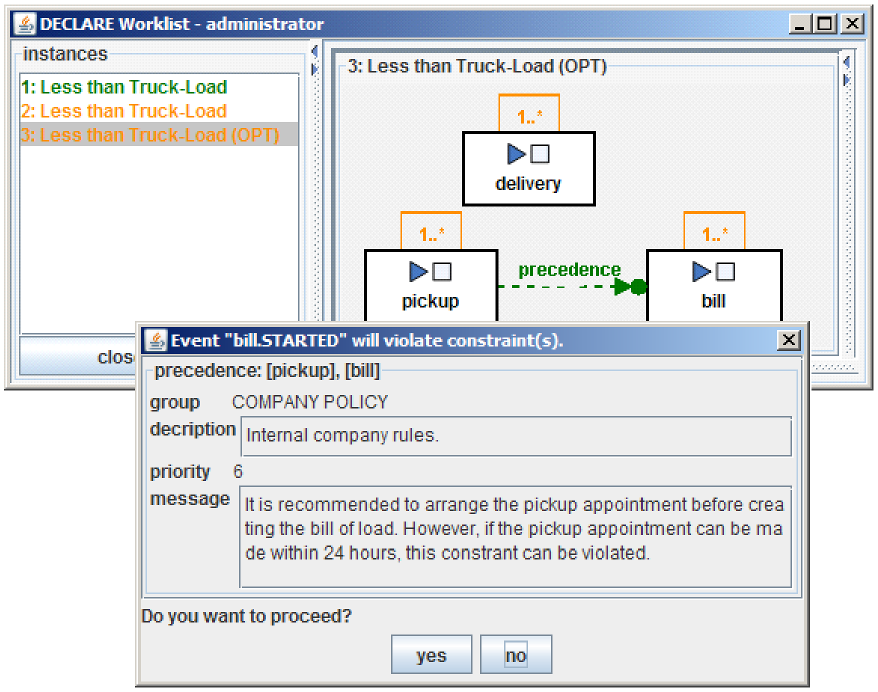
name	description
BILLING POLICY	Sets up rules that should ensure mininized billing risks
COMPANY POLICY	Internal company rules.
ORDERING POLICY	Rules defined for ordering processes.

Fig. 12.10: Warnings related to possible violations are linked to so-called constraint groups

the *precedence* constraint, it is possible to start task *bill* in this instance. This is because this constraint is optional. If the user would now attempt to start executing task *bill*, DECLARE would first issue a warning associated with the optional constraint, as shown in Figure 12.11. The user can now decide based on the information presented in the warning whether to start the execution of task *bill* and violate the constraint or not. Note that, in case of mandatory constraints, this is not possible, i.e. if this would be a mandatory constraint, it would not be possible to start executing task *bill* at this moment.

12.8 Dynamic Instance Change

Instances in DECLARE can be changed dynamically by adding and removing tasks and constraints. After the change, DECLARE creates an automaton for the whole mandatory formula (i.e. the conjunction of all mandatory constraints expressed in LTL) of the new model. If the instance trace can be 'replayed' on this automaton, the dynamic change is accepted. If not, the error is reported and the instance continues its execution based on the old model. Naturally, when applying a dynamic change, it is also possible to verify the new model against dead tasks and conflicts but these errors will not disable the change. Actually, the procedure for dynamic change is very similar to the procedure for starting instances in DECLARE, as Figure 12.12 shows. It is possible to perform basic model verification in both cases. The only difference is in the execution of the automaton. When an instance is started, the execution of the automaton begins from the initial state. In case of a dynamic change, DECLARE first makes an attempt to 'replay' the current trace of the instance on the new automaton, i.e. the new model is verified against the current trace. If this is possible, the dynamic change is successful and the execution continues from the current set of possible states in the new automaton, i.e. the instance state, enabled tasks, and states of constraints are determined using the new automaton and the current trace.



Event "bill.STARTED" will violate constraint(s).

precedence: [pickup], [bill]

group COMPANY POLICY

decription Internal company rules.

priority 6

message It is recommended to arrange the pickup appointment before crea ting the bill of load. However, if the pickup appointment can be ma de within 24 hours, this constrant can be violated.

Do you want to proceed?

Fig. 12.11: Warning: starting task *bill* violates the optional constraint *precedence*

If this is not possible, the dynamic change fails and the instance must continue using the old model.

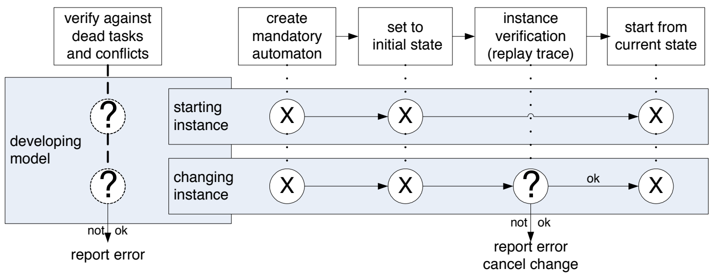
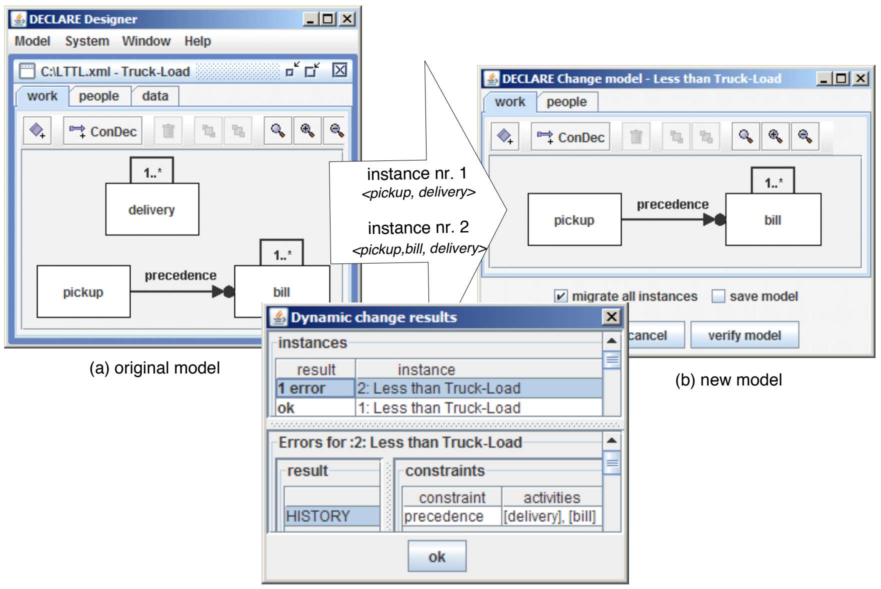

verify against dead tasks and conflicts
create mandatory automaton
set to initial state
instance verification (replay trace)
start from current state
developing model
starting instance
changing instance
?
?
X
X
X
X
X
?
X
ok
not
ok
not
ok
report error
report error cancel change

Fig. 12.12: Procedure for starting and changing instances in DECLARE

Besides changing a single instance, DECLARE offers two additional options: *migration* of all instances and *changing* the original constraint model. It is possible to request a migration of all instances, i.e. that the dynamic change is applied to all running instances of the same constraint model. DECLARE performs migration by applying the same procedure for dynamic change to all instances of the same constraint model, i.e. only instances with traces that can be replayed on the new automaton are migrated. It is also possible to also change the original constraint model. In this case, all instances created in the future will be based on the new model.

Consider, for example, the dynamic change scenario described in Chapter 6 with two instances of the ConDec model for the *Less than Truck Load* process shown in Figure 12.13(a). For the first instance, tasks $\langle pickup, delivery \rangle$ have been executed and tasks $\langle pickup, bill, delivery \rangle$ have been executed for the second instance. Figure 12.13(b) shows a DECLARE screenshot of a changed model where task *pickup*, constraint *1..** on the *delivery* task and the *precedence* constraint are removed, and a new *precedence* constraint is added between *delivery* and *bill* is added. As a consequence of adding the *precedence* constraint, task *bill* can now be executed only after task *delivery*. The migration of all active instances of this model are requested. Figure 12.13(c) shows the DECLARE report for the requested dynamic change. The migration is applied to the two currently running instances. The change failed for the first instance due to the violation of the *precedence* constraint as task *bill* already occurs before task *delivery* in the trace $\langle pickup, bill, delivery \rangle$. However, the dynamic change is successfully applied to the second instance, which has trace $\langle pickup, delivery \rangle$. Hence only the second instance is migrated and the first instance remains in the original process.



instances

result	instance
1 error	2: Less than Truck-Load
ok	1: Less than Truck-Load

Errors for :2: Less than Truck-Load

result: HISTORY

constraints

constraint	activities
precedence	[delivery], [bill]

(a) original model

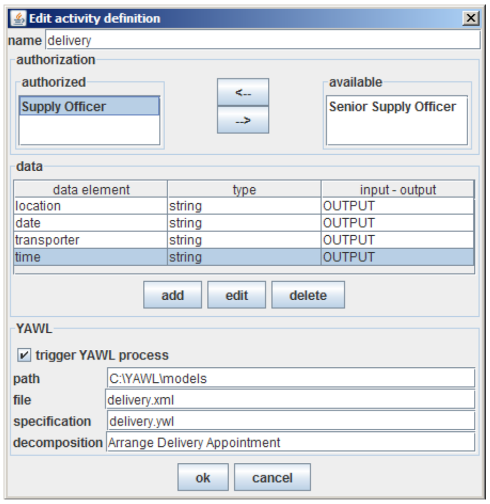
(b) new model

(c) this change cannot be applied to instance nr. 2

Fig. 12.13: Dynamic instance change in DECLARE

12.9 Decompositions of YAWL and Declarative Workflows

In both YAWL and DECLARE models, tasks are offered by default to users to manually execute them. If a task should be delegated to an external application, then this must be explicitly defined in the process model. Figure 12.14 shows how the *delivery* task from the DECLARE model shown in Figure 12.5 could be decomposed into a YAWL model, e.g. a model named Arrange Delivery Appointment. In this case, task *delivery* would be graphically presented with a special "YAWL" symbol in the DECLARE model. Note that, although task *delivery* will not be executed manually by a user, the user will decide *when* this task (i.e. the referring YAWL model) will be executed. However, when the user starts this task, it will not be opened in the 'task panel' in the *Worklist*. Instead, it will be automatically delegated to YAWL, which will launch a new instance of its Arrange Delivery Appointment model.


Edit activity definition

name: delivery

authorization

authorized: Supply Officer

available: Senior Supply Officer

data

data element	type	input - output
location	string	OUTPUT
date	string	OUTPUT
transporter	string	OUTPUT
time	string	OUTPUT

YAWL

[✓] trigger YAWL process

path	C:\YAWL\models
file	delivery.xml
specification	delivery.ywl
decomposition	Arrange Delivery Appointment

Fig. 12.14: DECLARE task *delivery* launches a YAWL instance

Similarly, in a YAWL process model it can be specified that a task should be delegated to a YAWL Custom Service, e.g. DECLARE. Figure 12.15 illustrates a YAWL instance where tasks D_1, D_2 and D_3 are delegated to DECLARE. In the general scenario, DECLARE users must manually select which DECLARE model should be executed for each YAWL request. For example, DECLARE users can choose to execute Model B for task D_1. If the decomposed YAWL task contains an input data element named 'declaremodel', then DECLARE automatically launches a new instance of the referring model. For example, in this particular YAWL process, task D_2 launches a new instance of Model A in DECLARE. If the specified model cannot be found, DECLARE users must manually select a DECLARE process models to be executed. For example, users can select to execute an instance of model Model C for task D_3. Because the ‘declaremodel’ data element is also an output data element in task D_3, DECLARE will return the name of the executed model to YAWL. In this manner YAWL users are informed about the sub-process that was executed for the decomposed YAWL task.

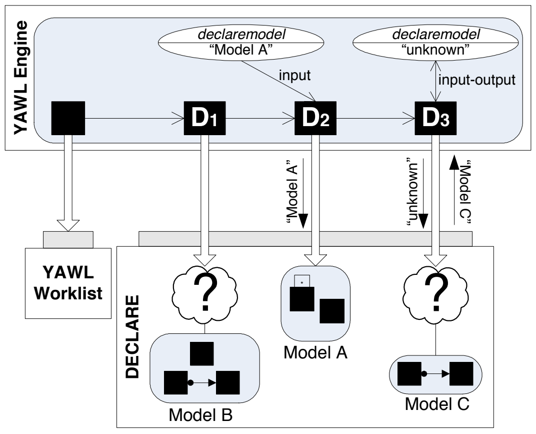


Fig. 12.15: YAWL tasks D_1, D_2 and D_3 launch DECLARE instances

12.10 Conclusions

Depending on the nature of a business process, a higher or lower degree of flexibility is needed when it comes to workflow support. For example, while procedural YAWL processes are appropriate for many of the more structured processes encountered in practice, they may be less suitable for less structured processes (such as our TL and LTTL sub-processes) that require the flexibility of declarative workflow models. Moreover, it might be the case that one business process contains some sub-processes that require the procedural approach and some sub-processes that require the declarative approach.

The Declare Service enables combining the two approaches by allowing for arbitrary decompositions of procedural YAWL and declarative DECLARE workflows. This is achieved by allowing the decomposition of both YAWL tasks into DECLARE workflows, and DECLARE tasks into YAWL nets. This way the workflow designer is not forced to make a binary choice between declarative and procedural workflows. Instead, an integration can be achieved, where parts of the workflow that need a high degree of flexibility are supported by declarative DECLARE models and parts of the processes that need centralized control of the system are supported by more procedural YAWL models. Note that YAWL also provides flexibility via the so-called Worklet Service. The Worklet Service (cf. Chapter 11) offers a different kind of flexibility that is complementary to DECLARE. Since the Declare Service and the Worklet Service are embedded in a similar manner, it is possible to arbitrarily mix procedural YAWL models, declarative models, and worklets.

Exercises

Exercise 1. Tasks *bless*, *pray*, *curse* and *become holy* are four tasks in the Religion process. Users of this process must obey two important rules. First, one should *pray* at least once. Second, every time one *curses*, one must eventually *pray* for forgiveness afterwards. Develop a model in DECLARE for this process using ConDec constraint templates.

Exercise 2. Consider the following four scenarios:

(a) $\langle \textit{become holy}, \textit{curse}, \textit{bless} \rangle$
(b) $\langle \textit{pray}, \textit{bless}, \textit{pray} \rangle$
(c) $\langle \textit{curse}, \textit{bless}, \textit{curse}, \textit{bless}, \textit{pray} \rangle$
(d) $\langle \textit{bless}, \textit{become holy} \rangle$

For each scenario create an instance of the Religion model developed in the previous exercise. Try to execute these four scenarios using DECLARE. Which traces were possible to execute and which were not? Describe why it was not possible to execute some of the given scenarios.

Exercise 3. Launch one instance of the Religion model described in the first exercise and monitor how states of the instance and constraints change while executing tasks $\langle \textit{bless}, \textit{curse}, \textit{bless}, \textit{bless}, \textit{curse}, \textit{become holy}, \textit{pray}, \textit{curse}, \textit{pray} \rangle$.

Exercise 4. Launch one instance of the Religion model described in the first exercise and execute tasks $\langle \textit{bless}, \textit{curse}, \textit{bless}, \textit{bless}, \textit{curse}, \textit{become holy}, \textit{pray}, \textit{curse}, \textit{pray} \rangle$. Now try to change the model of this instance in a way that:

- the constraint specifying that *one should pray at least once* is removed, and
- task *become holy* is removed.

Did DECLARE accept this dynamic change? Explain why or why not.

Exercise 5. Launch one instance of the Religion model described in the first exercise and execute tasks $\langle \textit{bless}, \textit{curse}, \textit{bless}, \textit{bless}, \textit{curse}, \textit{become holy}, \textit{pray}, \textit{curse}, \textit{pray} \rangle$. Now try to change the model of this instance in a way that a constraint specifying that *task become holy cannot be executed before task pray* is added. Did DECLARE accept this dynamic change? Explain why or why not.

Exercise 6. Develop a ConDec model in DECLARE for the process containing tasks *bless*, *pray*, *curse* and *become holy*, where three rules must be followed:

(a) One should *become holy* at least once.

(b) Every time one *curses*, one must eventually *pray* for forgiveness afterwards.
(c) Tasks *become holy* and *curse* cannot be both executed in the same instance.

Verify this model in DECLARE. Does this model have errors? Explain why or why not.

Exercise 7. Develop a ConDec model in DECLARE for the process containing tasks *bless*, *pray*, *curse* and *become holy*, where four rules must be followed:

(a) One should *become holy* at least once.
(b) Every time one *curses*, one must eventually *pray* for forgiveness afterwards.
(c) Tasks *become holy* and *curse* cannot be both executed in the same instance.
(d) One should *curse* at least once.

Verify this model in DECLARE. Does this model have errors? Explain why or why not.

Chapter Notes

This chapter described the Declare Service in YAWL. More details about the service and the DECLARE tool can be found in [19, 20, 192, 193, 195]. The combined use of YAWL, the Declare Service and the Worklet Service, possible due to the service-oriented architecture of YAWL, is described in [5]. More details about the DECLARE tool in relation to the YAWL architecture can be found in [194].