Chapter 6
Declarative Workflow

Maja Pesic, Helen Schonenberg, and Wil van der Aalst

6.1 Introduction

During the design of any information system, it is important to balance between *flexibility* and *support*. This is of particular importance when designing process-aware information systems. On the one hand, users want to have support from the system to conduct their daily activities in a more efficient and effective manner. On the other, the same users want to have flexibility, i.e. the freedom to do whatever they want and without being "bothered" by the system. Sometimes it is impossible to provide both flexibility and support because of conflicting requirements. The continuous struggle between flexibility and support is illustrated by Figure 6.1.

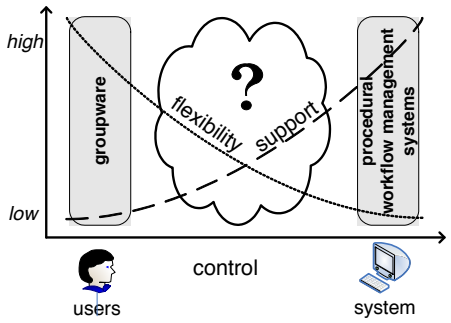


Fig. 6.1: Flexibility vs. Support, adapted from [79]

The right-hand-side of Figure 6.1 shows the part of the spectrum covered by classical *workflow management systems*. These systems focus on processes that are repeatedly executed in some predefined manner and are driven by *procedural languages*. Note that in procedural workflow models there may be alternative paths

controlled by (X)OR-splits/joins. However, the basic idea is that the completion of one task triggers other tasks. The YAWL nets described in earlier chapters provide such a procedural language. Although the YAWL language is highly expressive, its token-based semantics is most suitable for repetitive processes with tight control.

The left-hand-side of Figure 6.1 shows the other end of the spectrum. Here processes are less repetitive and the emphasis is on flexibility and user empowerment. Here it is difficult to envision all possible paths and the process is driven by user decisions rather than system decisions. *Groupware* systems (e.g. 'enhanced' electronic mail, group conferencing systems, etc.) support such processes and focus on supporting human collaboration, and co-decision. Groupware systems do not offer support when it comes to ordering and coordination of tasks. Instead, the high degree of flexibility of these systems allows users to control the ordering and coordination of tasks while executing the process (i.e. 'on the fly').

The trade-off between flexibility and support is an important issue in workflow technology. Despite the many interesting theoretical results and the availability of innovative research prototypes, few of the research ideas have been adopted in commercial systems. This is the reason why there is a question mark in the middle of Figure 6.1.

This chapter presents a completely new way of looking at workflow support. Instead of a procedural language we describe a *declarative language based on constraints*. The basic idea is that anything is allowed and possible unless explicitly forbidden. To implement this idea we use a temporal logic: *Linear Temporal Logic* (LTL). Although LTL has been around for decades, it has never been made accessible for workflow design. Therefore, we provide a graphical notation and a set of supporting tools. This nicely complements the core engine and design environment of YAWL.

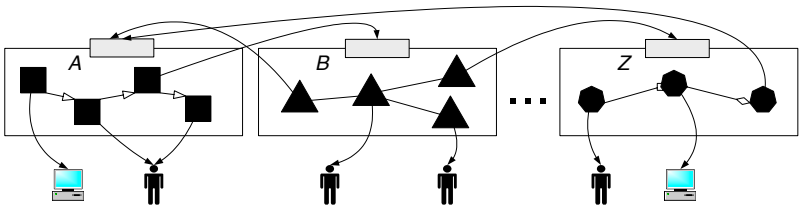


Fig. 6.2: Decomposing processes using various modeling languages

Before elaborating on our constraint-based approach it is important to stress that this is not a silver bullet. In reality, both flexibility and support are needed when it comes to the computer-facilitated execution of processes and the various paradigms offer different advantages. On the one hand, flexibility is needed for unpredictable processes, where users can quickly react to exceptional situations and execute the process in the most appropriate manner. On the other hand, support is needed when it comes to processes that are repeatedly executed in the same manner, in situations that are too complex for humans to handle and where human mistakes must be

minimized. For example, the processing of insurance claims, customer orders, tax declarations, etc. can benefit from a high degree of support because cases are repeatedly executed in a similar manner. Thus, an optimal balance between flexibility and support is needed in order to be able to facilitate processes of various kinds. Moreover, in a large process there may be parts that require more flexibility and parts that require more support. Therefore, procedural and flexible workflows should not exclude each other. Instead, arbitrary decompositions of process models developed in procedural and declarative languages should be possible. These decompositions can be achieved by the seamless integration of different process engines based on different paradigms. Figure 6.2 illustrates how various processes can be combined in arbitrary decompositions. Instead of being executed as a simple (manual or automatic) unit of work, a task can be decomposed into a process modeled in an arbitrary language. The declarative language based on constraints presented in this chapter can be seen as one of the modeling languages in Figure 6.2, while the YAWL nets, and worklets (cf. Chapter 11) can be seen as another alternative.

The basic language used by YAWL provides excellent support for procedural workflow modeling as it supports most of the workflow patterns (cf. Chapter 2). However, in this chapter we focus on adding flexibility to workflow models. A workflow model specifies the order in which tasks can be executed, i.e. possible execution alternatives. An execution alternative is a unique ordering of tasks. More execution alternatives mean more flexibility. There are several ways to increase the flexibility of workflow management systems:

1. *Flexibility by design* is the ability to specify alternative execution alternatives (task orderings) in the workflow model, such that users can select the most appropriate alternative at runtime for each workflow instance.
2. *Flexibility by underspecification* is the ability to leave parts of a workflow model unspecified. These parts are later specified during the execution of workflow instances. In this way, parts of the execution alternatives are left unspecified in the workflow model, and are specified later during the execution.
3. *Flexibility by change* is the ability to modify a workflow model at runtime, such that one or several of the currently running workflow instances are migrated to the new model. Change enables adding one or more execution alternatives during execution of workflow instances.
4. *Flexibility by deviation* is the ability to deviate at runtime from the execution alternatives specified in the workflow model, without changing the workflow model. Deviation enables users to 'ignore' execution alternatives prescribed by the workflow model by executing an alternative not prescribed in the model.

Flexibility of workflow management systems can be increased by using *declarative languages for workflow specification*. This chapter describes how an optimal balance between flexibility and support can be achieved with declarative workflow management systems that use *constraint* workflow models. This chapter shows how the constraint-based approach can provide for all types of flexibility listed in the previous paragraph. A concrete implementation that enables creating decompositions of YAWL (e.g. procedural) and declarative models is presented in Chapter 12

of this book. Figure 6.3 illustrates the difference between procedural and declarative process models.

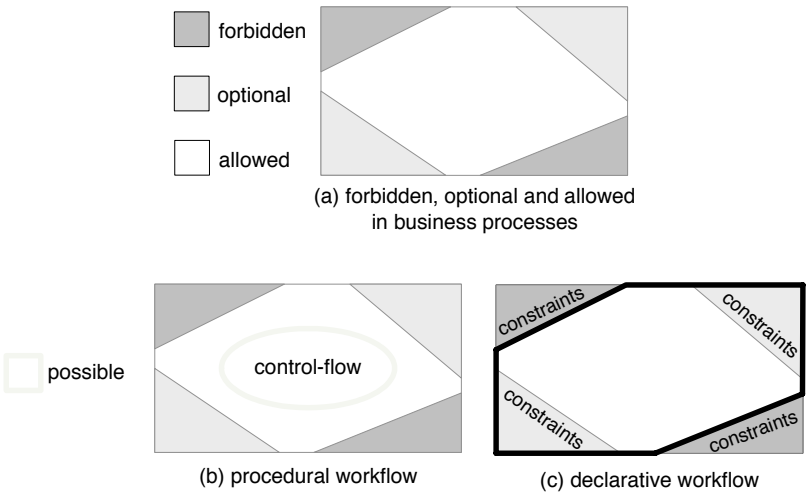


Fig. 6.3: Declarative vs. Procedural workflows

Starting point for the declarative constraint-based approach is the observation that only three types of 'execution alternatives' can exist in a process: (1) *forbidden* alternatives should never occur in practice, (2) *optional* alternatives are allowed, but should be avoided in most of the cases, and (3) *allowed* alternatives can be executed without any concerns. This is illustrated in Figure 6.3(a). Procedural workflow models (e.g. YAWL nets) *explicitly* specify the ordering of tasks, i.e. the *control-flow* of a workflow. In other words, during the execution of the model it will be *possible* to execute a process only as explicitly specified in the control-flow, as Figure 6.3(b) shows. Due to the high level of unpredictability of processes, many allowed and optional executions often cannot be anticipated and explicitly included in the control-flow. Therefore, in traditional systems it is *not possible* to execute a substantial part of all potentially *allowed* alternatives, i.e. users are unnecessary limited in their work and, hence, these systems lack flexibility by design.

Our declarative *constraint-based approach* to workflow models makes it *possible* to execute both *allowed* and *optional* alternatives in processes. Instead of explicitly specifying the procedure, constraint workflow models are declarative: they specify constraints, i.e. rules that should be followed during the execution, as shown in Figure 6.3(c). Moreover, there are two types of constraints: (1) *mandatory* constraints focus on the forbidden alternatives, and (2) *optional* constraints specify the optional ones. Constraint-based models are declarative: anything that does not violate mandatory constraints is *possible* during execution. The declarative nature of

constraint models allows for a wide range of possible execution alternatives, which enhances flexibility.

The remainder of this chapter is organized as follows. First, our constraint-based language for workflow specification is presented in Section 6.2. This section also discusses the possibility to verify constraint-based models and the ability to define alternative languages and templates. Section 6.3 shows how declarative workflows can be enacted, i.e. how to create a runtime environment for execution. The possibility to change models while they are being executed (i.e. the so-called dynamic change) is described in Section 6.4. Section 6.5 concludes the chapter, followed by exercises and chapter notes.

6.2 Constraint-Based Workflow Specification

A constraint-based workflow is defined by the specification of tasks and constraints. In this section, we first show the formal foundation of a constraint specification language. Then we introduce the concept of *constraint templates*, i.e. reusable constructs linking graphical notations to temporal logic. These templates also allow for branching, i.e. a construct can be associated to a variable number of tasks. Then we introduce *ConDec* as an example language. This language is used in the remainder of this chapter. We conclude this section by showing that such models can be verified.

6.2.1 LTL-Based Constraints

Various temporal logics have been proposed in the literature: Computational Tree Logic (CTL), Linear Temporal Logic (LTL), etc. All of these logics could be used to specify constraints. In this chapter, we focus on *Linear Temporal logic* LTL.

LTL can be used to define properties of traces, e.g. "after every occurrence of X eventually Y should occur". In the context of workflow models, traces correspond to sequences of tasks. Every trace represents an executed alternative where the tasks in the trace occur exactly in the order in which they appear.

Definition 1 (Trace). Let $\mathscr{T}$ denote the universe of task identifiers, i.e. the set of all *tasks*. Trace $\sigma \in \mathscr{T}^*$ is a sequence of tasks, where $\mathscr{T}^*$ is the set of all traces composed of zero or more elements of $\mathscr{T}$. We use $\sigma = \langle t_1, t_2, \ldots, t_n \rangle$ to denote a trace consisting of tasks $t_1, t_2, \ldots, t_n$. σ_i denotes the i-th element of the trace, i.e. $\sigma_i = t_i$. $e \in \sigma$ denotes $\exists_{1 \leq i < |\sigma|}[\sigma_i = e]$. We use $+$ to concatenate two traces into a new trace: $\langle e_1, e_2, \ldots, e_n \rangle + \langle f_1, f_2, \ldots, f_m \rangle = \langle e_1, e_2, \ldots, e_n, f_1, f_2, \ldots, f_m \rangle$.

The concept of traces will be used to explain the semantics of LTL. However, we first give the syntax of LTL. LTL provides the classical logic operators ($\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$), and uses several temporal operators ($\bigcirc$, $\Diamond$, $\Box$, U, and W) that can be used

to specify constraints over the sequencing of workflow tasks. Definition 2 defines the syntax of LTL.

Definition 2 (LTL syntax). Each atomic proposition p is an LTL formula. Using various operators, more complex expressions can be constructed. If Φ and Ψ are LTL formulas, then $\neg\Phi, \Phi \vee \Psi, \bigcirc\Phi$ and $\Phi U \Psi$ are also LTL formulas. The semantics of these constructs are given in Definition 3. Moreover, the following operators are so-called derived LTL operators. These operators can be expressed in terms of the basic operators.

$\Phi \wedge \Psi \equiv \neg(\neg\Phi \vee \neg\Psi)$
$\Phi \Rightarrow \Psi \equiv \neg\Phi \vee \Psi$
$\Phi \Leftrightarrow \Psi \equiv (\Phi \Rightarrow \Psi) \wedge (\Phi \Rightarrow \Psi)$
$\texttt{true} \equiv \Phi \vee \neg\Phi$
$\texttt{false} \equiv \neg\texttt{true}$
$\Diamond\Phi \equiv \texttt{true} U \Phi$
$\Box\Phi \equiv \neg\Diamond\neg\Phi$
$\Phi W \Psi \equiv (\Phi U \Psi) \vee \Box\Phi$

LTL formulas can be used to express that, (1) Φ should hold at the next element of the trace ($\bigcirc\Phi$), (2) Φ should hold eventually ($\Diamond\Phi$), (3) Φ should always hold ($\Box\Phi$), (4) Φ should hold until eventually Ψ holds ($\Phi U \Psi$), and (5) Φ should hold until Ψ holds, but Ψ is not required to hold eventually ($\Psi W \Phi$). The formal semantics of the basic temporal operators are given in Definition 3.

Definition 3 (LTL semantics). Let p be an atomic proposition, $\sigma = \langle \sigma_1, \sigma_2, \sigma_3, \ldots \rangle$ be a trace. Then

$\sigma \vDash p \Leftrightarrow \sigma_1 = \text{p}$
$\sigma \vDash \neg\Phi \Leftrightarrow \text{not } \sigma \vDash \Phi$
$\sigma \vDash \Phi \vee \Psi \Leftrightarrow \sigma \vDash \Phi \text{ or } \sigma \vDash \Psi$
$\sigma \vDash \bigcirc\Phi \Leftrightarrow \langle \sigma_2, \sigma_3, \ldots \rangle \vDash \Phi$
$\sigma \vDash \Phi U \Psi \Leftrightarrow \exists_{1 \leq i}[\langle \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots \rangle \vDash \Psi \wedge \forall_{1 \leq j < i}[\langle \sigma_j, \sigma_{j+1}, \ldots \rangle \vDash \Phi]]$

Note that the semantics of the other operators are provided in an indirect manner, e.g. $\Diamond\Phi$ is semantically equivalent to $\texttt{true} U \Phi$, etc. Figure 6.4 shows some example traces to clarify some of the constructs. Note that LTL is typically defined on infinite traces, but all results for infinite sequences can be mapped onto finite sequences.

The set of possible execution traces satisfying an LTL constraint is defined as follows.

Definition 4 (Constraint satisfying traces $\mathscr{T}^*_{\vDash c}$). Let constraint c be an LTL formula. Then the set of traces satisfying c is defined as $\mathscr{T}^*_{\vDash c} = \{\sigma \in \mathscr{T}^* \mid \sigma \vDash c\}$.

In the field of model checking, LTL is extensively used to check whether a system satisfies properties specified by LTL formulae. Using the results from this field, it is possible to generate a non-deterministic finite state automaton that represents *exactly all traces that satisfy the LTL formula*. For a constraint c, specified by an

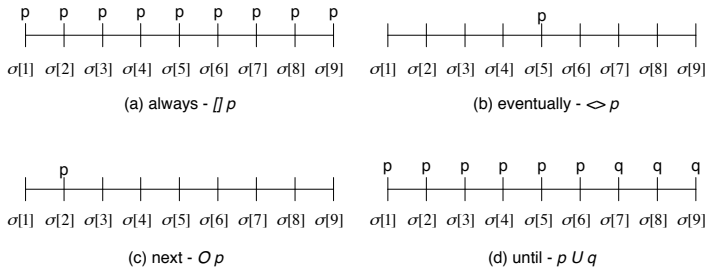


Fig. 6.4: Semantics of some LTL operators

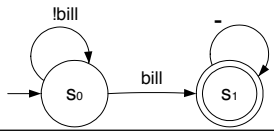
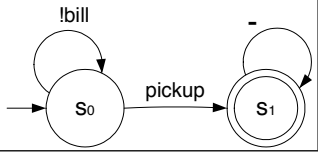
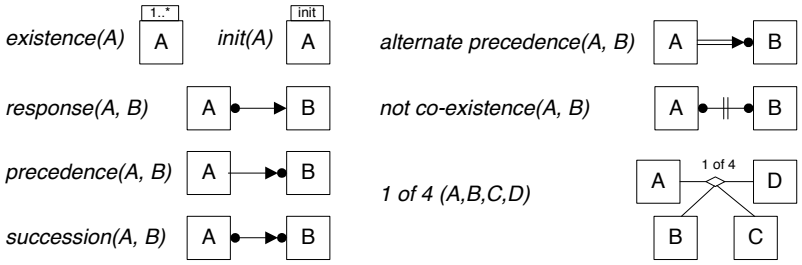
LTL formula for constraint c	automaton representing $\mathscr{T}^*_{\models c}$	semantics
$\Diamond$ *bill*		Task *bill* must be executed at least once.
!(*bill*) *U pickup*		Task *bill* cannot be executed until task *pickup* is executed.

Table 6.1: Formal specification of constraints with LTL

LTL formula, the set of all traces satisfying c ($\mathscr{T}^*_{\models c}$), is exactly the language produced by the automaton generated for this formula using the algorithm of D. Giannakopoulou and K. Havelund. Table 6.1 shows two examples illustrating the general idea. It is not important to understand how the automata are generated, however, to understand our approach it is important to understand that for any LTL constraint a corresponding automaton can be generated. This automaton will be used to enforce constraints, detect violations, and to check the consistency of a specification. Section 6.3 describes how the LTL-based automata can be used for declarative workflow execution.

The next example gives the LTL constraints for a process involving tasks such as *shipment* (i.e. *Create Shipment Information Document*), *bill* (i.e. *Create Bill of Lading*) and *pickup* (i.e, *Arrange Pickup Appointment*). In this example, the number of constraints is small and it is not difficult to determine whether a trace satisfies the given constraints, or not. For larger models this is less intuitive. Therefore, the next subsection proposes *constraint templates* as a graphical way to represent LTL formulas.

Example 1 (Constraint satisfying traces). Consider the two constraints described in Table 6.1 for the execution of tasks *bill* and *pickup*; (c_1) $\Diamond$*bill* and (c_2) !(*bill*) *U pickup*.



Where

$$
\begin{aligned}
&\mathit{existence}(A) && \equiv \Diamond A\\
&\mathit{init}(A) && \equiv A\\
&\mathit{response}(A,B) && \equiv \Box(A \Rightarrow \Diamond B)\\
&\mathit{precedence}(A,B) && \equiv (\neg B)WA\\
&\mathit{succession}(A,B) && \equiv \mathit{response}(A,B) \wedge \mathit{precedence}(A,B)\\
&\mathit{not\ co-existence}(A,B) && \equiv \neg(\Diamond A \wedge \Diamond B)\\
&\mathit{1\ of\ 4}(A,B,C,D) && \equiv \Diamond A \vee \Diamond B \vee \Diamond C \vee \Diamond D
\end{aligned}
$$

Fig. 6.5: Constraint templates

Examples of traces satisfying constraints c_1 and c_2 are $\langle \mathit{pickup}, \mathit{bill} \rangle \models c_1 \wedge c_2$ and $\langle \mathit{pickup}, \mathit{pickup}, \mathit{bill} \rangle \models c_1 \wedge c_2$. Traces not satisfying constraints c_1 and c_2 are $\varepsilon \not\models c_1 \wedge c_2$, $\langle \mathit{pickup} \rangle \not\models c_1 \wedge c_2$ and $\langle \mathit{bill}, \mathit{pickup} \rangle \not\models c_1 \wedge c_2$. Note that $\varepsilon \models c_2$.

6.2.2 Constraint Templates

Procedural workflow languages such as YAWL provide constructs such as AND-split, AND-join, cancelation region, etc. These constructs aim at supporting frequently needed workflow patterns. For our declarative approach we also aim at supporting frequently needed patterns to model relationships (constraints) between tasks. However, we aim at different types of patterns, i.e. declarative constructs aiming at more flexibility. In principle, LTL offers everything needed to support such patterns. However, because LTL formulas can be difficult to understand by non-experts, we provide a graphical representation of constraints that hides the associated LTL formulas from users of declarative workflow, in the form of *constraint templates*. Each template has (1) a name, (2) an LTL formula and (3) a graphical representation. A constraint inherits the name, graphical representation and the LTL formula from its template. Figure 6.5 depicts some example constraint templates. The template parameters are depicted by boxes. Templates can be used to create actual constraints for a specific process. In actual constraints, tasks replace template parameters, both in the graphical representation and the associated LTL formula.

The *existence* template is graphically represented with the annotation "1..$*$" above the task. This indicates that A is executed at least once in each instance. The

template *init*(A) can be used to specify that task *A* must be the first executed task in an instance. Templates *response*, *precedence* and *succession* consider the ordering of tasks. Template *response* requires that every time task *A* executes, task *B* has to be executed afterwards. Note that this is a very relaxed interpretation of the notion of *response*, because *B* does not have to execute straight after *A*, and another *A* can be executed between the first *A* and the subsequent *B*. The template *precedence* specifies that task *B* can be executed only after task *A* is executed. Just like in the *response* template, other tasks can be executed between tasks *A* and *B*. The combination of the *response* and *precedence* templates defines a bi-directional execution order of two tasks and is called *succession*. In this template, both *response* and *precedence* relations have to hold between the tasks *A* and *B*. Template *alternate precedence* strengthens the *precedence* template: tasks *A* and *B* have to alternate. The *not co-existence* template specifies that tasks *A* to *B* cannot both be executed in the same instance. The *1 of 4* template specifies that at least one of the three tasks *A*, *B*, *C* and *D* has to be executed, but all four tasks can be executed an arbitrary number of times as long as at least one of them occurs at least once.

6.2.3 Branching of Templates

The behavior of models containing multiple constraints is given by the conjunction of all constraints. In some cases it might be necessary to specify the disjunction of constraints. This can be done by assigning more than two tasks to one parameter in a template. When a template parameter is replaced by more than one task in a constraint, then we say that this parameter *branches*. An example of a branched *precedence* constraint is shown in Figure 6.6 where parameter *A* has been replaced by the task *bill* and parameter *B* is branched on tasks *pickup* and *delivery*. In case of branching, the parameter is replaced (1) by multiple arcs to all branched tasks in the graphical representation and (2) by a disjunction of branched tasks in the LTL formula. The semantics of branching can vary from template to template, depending on the LTL formula of the template. For example, the branched constraint in Figure 6.6 specifies that each occurrence of *bill* must be preceded by at least one occurrence of task *pickup* or task *delivery*. Note that it is possible to branch all parameters, one parameter or none of the parameters. The number of possible branches in constraints is unlimited. For example, it is possible to branch the parameter *B* in the *response* template to *N* alternatives, as shown in Figure 6.6.

Another way of branching templates are *choice* templates which can be used to specify that one must choose between tasks. An example of such a template is the *1 of 4* template (see Figure 6.5), which is used to specify that at least one of the four given tasks has to be executed.

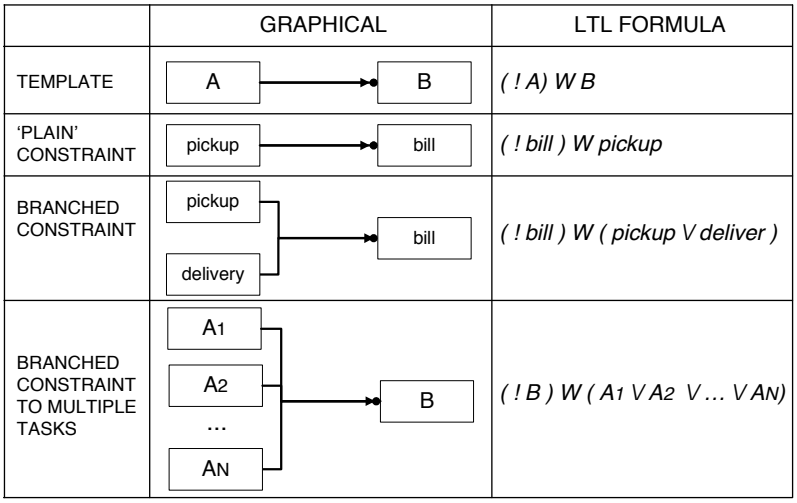


Fig. 6.6: Branching the *response* template

6.2.4 An Example Language: ConDec

As explained before, we use LTL as a basis to define constraint templates. The template is defined by specifying the corresponding LTL formula and graphical representation. This makes it easy to create various languages, as any collection of constraint templates forms a language. In this chapter, the focus is on the ConDec language. ConDec provides more than twenty constraint templates, among which the ones depicted in Figure 6.5. ConDec templates are structured into three groups: (1) *Existence* templates specify how many times or when a task can be executed, e.g. *init* and *existence* templates; (2) *relation* templates define some relation between two (or more) tasks, e.g. *response*, *precedence* and *succession*; (3) *negation* templates define a negative relation between tasks, e.g. *not co-existence*; and (4) choice templates define a choice between tasks, e.g. 1 of 4.

Figure 6.7 depicts the relation between constraint templates, constraints and models for ConDec. At the template abstraction level, templates are defined on a predefined number of abstract parameters (e.g. parameter *A* and *B*). When a constraint is specified, the abstract parameter is replaced by a concrete one, e.g. in Figure 6.7 the left *precedence* constraint concretizes abstract parameter *A* from the *precedence* template with concrete parameter *bill*. A "plain" constraint is a concretization of the predefined parameters of a constraint template (cf. Figure 6.6). Each constraint can easily be extended to deal with more parameters than defined by its template by the means of branching.

Every ConDec template involves a specific number of tasks. For example, templates *existence(A)*, *precedence(A,B)* and *1 of 4(A,B,C,D)* involve one, two, and four tasks, respectively. When a constraint is created based on a template, the constraint

will involve as many real tasks as predefined in the template. Templates can be re-used to specify different constraints, as shown for the *precedence* template in Figure 6.7. Finally, constraints from the ConDec constraint templates can be used to specify ConDec models.

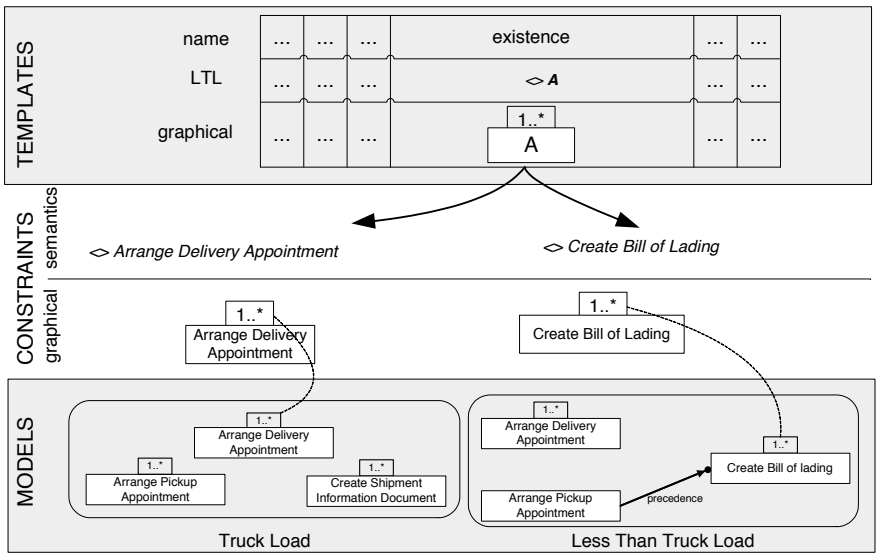


Fig. 6.7: ConDec templates, constraints and models

Example 2 (Carrier Appointment examples in ConDec). Consider, for example, the two highlighted constraints in different models shown in Figure 6.7. First, model *Truck Load* contains the constraint specifying that 'Task *Arrange Delivery Appointment* must be executed at least once'. Second, the *Less Than Truck Load* model contains a constraint specifying that 'Task *Create Bill of Lading* must be executed at least once'. The first constraint can be specified with formula $\Diamond$ *Arrange Delivery Appointment* and the second one with a similar formula $\Diamond$ *Create Bill of Lading*. Both constraints use the same (*existence*) template, their LTL specifications are similar: $\Diamond A$, but the constraints are specified on different tasks. Instead of having to individually specify formulas for every constraint, constraint templates provide a way to re-use formulas.

6.2.5 Constraint Workflow Models

Constraint workflow models consist of tasks and constraints. Moreover, there are two types of constraints: *mandatory constraints* represent rules that must be followed during execution, while *optional constraints* can be violated.

Definition 5 (Constraint model *cm*). Let $\mathscr{T}$ be the universe of all tasks and $\mathscr{C}$ be the set of all constraints. A constraint model *cm* is defined as a triple $cm = (T, C_M, C_O)$, where:

- $T \subseteq \mathscr{T}$ is a set of *tasks* in the model,
- $C_M \subseteq \mathscr{C}$ is a set of *mandatory constraints* where every element $c \in C_M$ is a well-formed LTL formula over T,
- $C_O \subseteq \mathscr{C}$ is a set of *optional constraints* where every element $c \in C_O$ is a well-formed LTL formula over T).

Note that each mandatory and optional constraint is a well-formed LTL formula over tasks from *cm*, i.e. a constraint cannot involve tasks that are not in the model.

During execution mandatory constraints must be followed. Therefore the set of traces that satisfy constraint model *cm* contains all traces that satisfy *all* mandatory constraints in *cm*. The conjunction of all mandatory formulas is defined as the *mandatory formula of the model*.

Definition 6 (Mandatory formula f_{cm}). Let $cm \in \mathscr{U}_{cm}$ be a constraint model where $cm = (T, C_M, C_O)$, then the mandatory formula for model *cm* is defined as

$$f_{cm} = \begin{cases} \texttt{true} & \text{if } C_M = \varnothing; \\ \bigwedge_{f \in C_M} f & \text{otherwise.} \end{cases}$$

The automaton generated for the mandatory formula (f_{cm}) of a model *cm* generates the language $\mathscr{T}^*_{\vDash cm}$, i.e. the set of traces that satisfy *cm*. In the absence of mandatory constraints, all traces over tasks in the model are allowed.

Definition 7 (Constraint model satisfying traces). Let f_{cm} be the mandatory formula for constraint model *cm*.

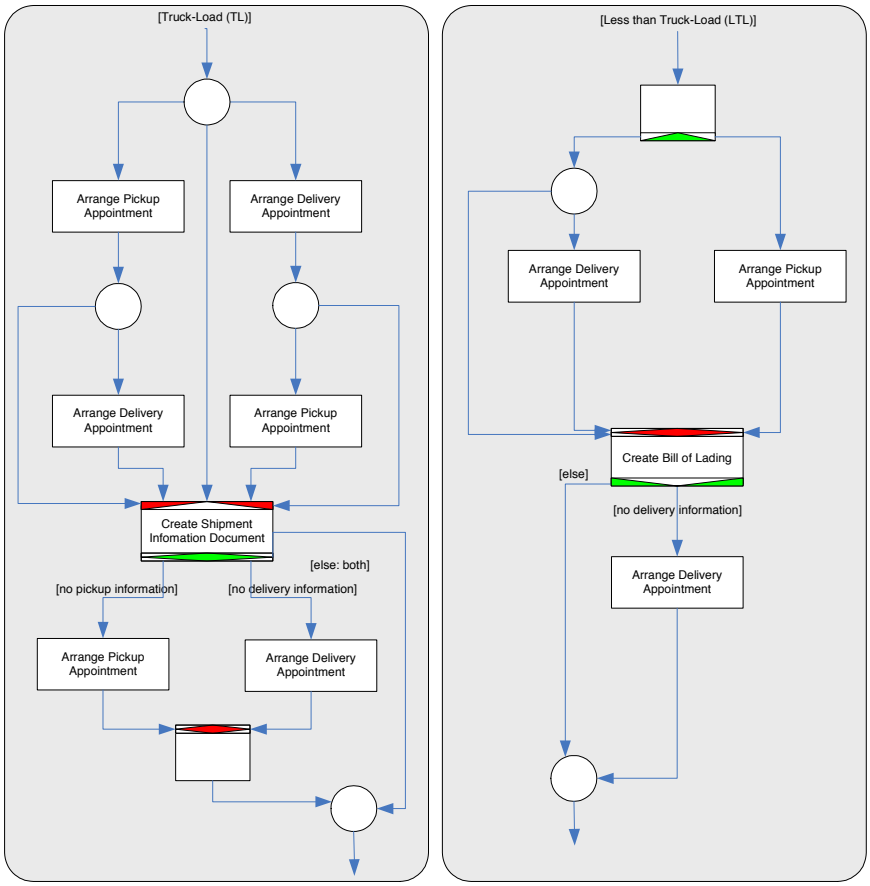
$$\mathscr{T}^*_{\vDash cm} = \begin{cases} \mathscr{T}^* & \text{if } C_M = \varnothing; \\ \mathscr{T}^*_{\vDash f_{cm}} & \text{otherwise.} \end{cases}$$

Furthermore we say a trace $\sigma \in \mathscr{T}^*$ *satisfies* model *cm*, if and only if, $\sigma \in \mathscr{T}^*_{\vDash cm}$ and σ *violates cm*, if and only if, $\sigma \notin \mathscr{T}^*_{\vDash cm}$.

Consider, for example, the *Carrier Appointment* sub-process of the Order Fulfillment process model described in Appendix A. In particular, consider the parts (i.e. branches) of the *Carrier Appointment* net for handling the shipments that require *the truck load* (TL) and *less than truck load* (LTTL)[1]. Figure 6.8(a) shows the *the*

[1] Note that in the original *Carrier Appointment* net this branch of the net is called LTL. In this chapter we will use the LTTL abbreviation to avoid confusion with Linear Temporal Logic (LTL).

truck load part, and Figure 6.8(b) shows the *less than the truck load* part of the *Carrier Appointment* net. For the purpose of simplicity in the remainder of 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*. Examples 3 and 4 show ConDec models for these two parts of the *Carrier Appointment* net.

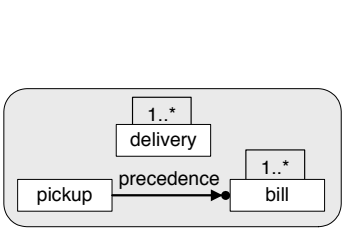


(a) *the truck load* (TL) (b) *less than truck load* (LTTL)

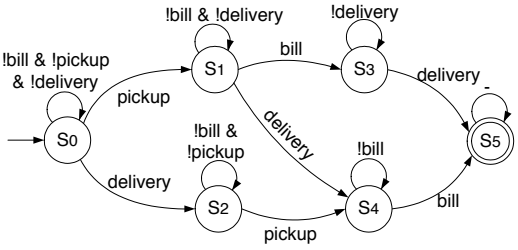
Fig. 6.8: Two parts of the *Carrier Appointment* YAWL sub-net of the Order Fulfillment process

Example 3 (The ConDec model for the LTTL process (cm_{LTTL})). In the LTTL branch of the process the goal is to execute tasks *pickup*, *delivery* and *bill* in such a way that task *pickup* must be executed before task *bill*, and task *delivery* can be executed either before or after task *bill*. While he YAWL net for the *Carrier Appointment* subprocess explicitly specifies all possible ordering of the related tasks, these orderings

can be implicitly specified by using ConDec constraints, as shown in Figure 6.9(a). Constraints *1..** on tasks *delivery* and *bill* specify that each of these tasks must be executed at least once. The *precedence* constraint specifies that task *bill* can be executed only after task *pickup* is executed. The automaton presented in Figure 6.9(b) is generated for the mandatory formula of this model, and it represents exactly all traces that satisfy this model. From the initial state (S_0), the accepting state (S_5) can only be reached if *bill* and *delivery* are executed and *bill* is not executed before *pickup*.



(a) ConDec model



(b) automaton for mandatory formula $f_{cm_{LTTL}} = (\Diamond\ \textit{delivery}) \wedge (\Diamond\ \textit{bill}) \wedge ((!\ \textit{bill})\ W\ \textit{pickup})$ represents the set of satisfying traces $\mathscr{T}^*_{\vDash cm_{LTTL}}$

Fig. 6.9: The *LTTL* model
cm_{LTTL}=({*pickup*,*delivery*,*bill*},{$\Diamond$ *delivery*,$\Diamond$ *bill*,(! *bill*) *W pickup*},$\varnothing$)

Example 4 (The ConDec model for the TL process (cm_{TL})). In the TL branch of the process the goal is to execute tasks *pickup*, *delivery* and shipment in any order. While he YAWL net for the *Carrier Appointment* sub-process explicitly specifies all possible ordering of the related tasks, these orderings can be implicitly specified by using ConDec constraints, as shown in Figure 6.9(a). Constraints *1..** on tasks *delivery* and shipment specify that each of these tasks must be executed at least once. Note that Figure 6.10 does not show he automaton generated for the mandatory formula of this model. This is because the generated automata is too complex (i.e. it has 8 states and 20 transitions) to be presented as an illustrative example.

Note that both the *TL* and the *LTTL* sub-process just contain a small number of constraints, nonetheless there are infinitely many traces that satisfy the constraint model. In fact, both models have infinitely many possible execution alternatives, so are very flexible by design (cf. Section 6.1). *The main difference between procedural and declarative languages is that in procedural languages it takes effort (specifying all paths) to extend the behavior of a process, whereas in declarative languages it takes effort (specifying constraints) to limit the behavior of a process.* In this sense, declarative languages offer a more convenient way to express flexibility than procedural ones. Consider, for example the ConDec models for the *TL* and *LTTL* parts of

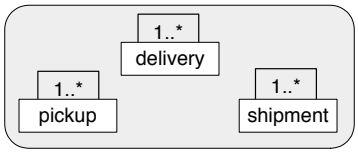


Fig. 6.10: The *TL* model
cm_{TL}=({*pickup*,*delivery*,shipment},{◊ *pickup*,◊ *delivery*,◊ *shipment*},∅)

the *Carrier Appointment* net. Not only that ConDec models define the possible execution alternatives in a more concise way, but they also allow that tasks are executed multiple times when necessary.

6.2.6 Verification of Constraint Models

Verification techniques can be used for detection of possible errors in models. The verification capabilities of YAWL will be described in Chapter 20. Just like for procedural languages having formal semantics, constraint-based model expressed in LTL allow for various kinds of analysis.

Certain combinations of constraints can cause errors in constraint models that lead to undesirable effects at execution-time. We distinguish two types of constraint model errors: *dead tasks* and *conflicts*.

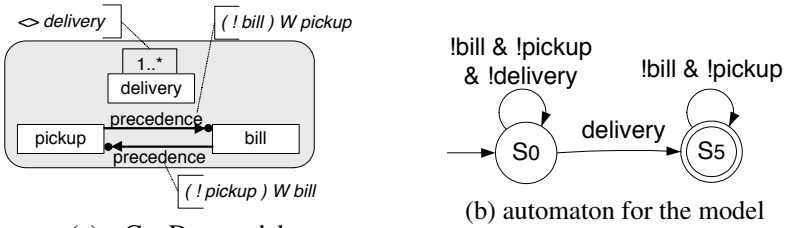
A task in a constraint model is dead if none of the traces satisfying the model contains this task, i.e. the task cannot be executed in any instance of the model without violating the specified (mandatory) constraints.

Definition 8 (Dead task). Let $cm \in \mathscr{U}_{cm}$ be a constraint model. task $t \in \mathscr{T}$ is *dead* in model *cm*, if and only if $\nexists \sigma \in \mathscr{T}^*_{\models cm} : t \in \sigma$.

Example 5 (Dead task). Figure 6.11(a) shows a ConDec model where tasks *pickup* and *bill* are dead. This error is caused by the combination of the two *precedence* constraints. While one *precedence* constraint specifies that task *bill* cannot be executed before task *pickup*, the other *precedence* constraint specifies that task *pickup* cannot be executed before task *bill*. Therefore, it will never be possible to execute tasks *pickup* and *bill* in any instance of the model shown in Figure 6.11(a).

Dead tasks can easily be detected by analyzing the automaton generated for the mandatory formula of a ConDec model. A task is dead if none of the transitions in the automaton allow the execution of this task. Figure 6.11(b) shows the automaton generated for the model in Figure 6.11(a). Indeed, this automaton does not contain transitions that allow for the execution of tasks *pickup* and *bill*.

A constraint model contains a conflict if there are no traces that can satisfy the model, i.e. instances of the model can never become *satisfied*.



(a) a ConDec model

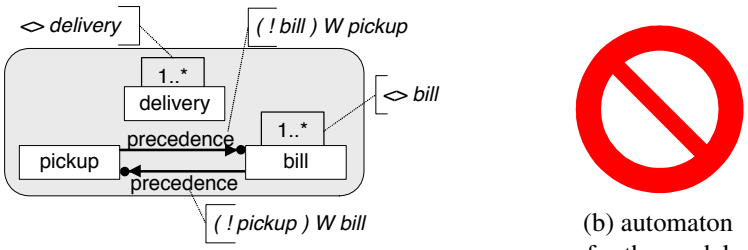
(b) automaton for the model

Fig. 6.11: Tasks *pickup* and *bill* are dead

Definition 9 (Conflict). Model $cm \in \mathscr{U}_{cm}$ has a *conflict* if and only if $\mathscr{T}^*_{\models cm} = \varnothing$.

Example 6 (Conflict). Figure 6.12(a) shows a ConDec model containing a conflict. In addition to the constraints described in Example 5, there now is an additional constraint *1..** requiring that task *bill* should be executed at least once. However, task *bill* is a dead task (see Exercise 5), i.e. it cannot be executed without violating constraints. So, there are no traces that can satisfy all constraints in this model.

Detection of conflicts can also be done using the automaton generated for the mandatory formula of the model. If the automaton is empty (i.e. it has no states), then this model has a conflict, otherwise it is conflict free. Figure 6.12(b) shows the automaton generated for the model from Figure 6.12(a). The automaton is empty thus indicating that Figure 6.12(a) has a conflict.



(a) a ConDec model

(b) automaton for the model

Fig. 6.12: The ConDec model has a conflict as shown by the empty automaton

In the previous examples, dead tasks and conflicts were not caused by the combination of all constraints in the model. Instead, they were caused by a specific group of constraints. Dead tasks *pickup* and *bill* in Example 5 are caused by the two *precedence* constraints and the conflict in Example 6 by the two *precedence* constraints and the *1..** constraint on task *bill*. The smallest subset of mandatory constraints that causes the error is called *the cause of the error*. In ConDec models, the cause of error can be found by searching through the powerset of mandatory constraints.

For each element (i.e. subset of mandatory constraints) in the powerset, the automaton for the mandatory formula can be analyzed for the presence of dead tasks and conflicts as described in previous paragraphs. Detecting the smallest group of mandatory constraints that causes an error supports the user in developing error-free constraint models.

6.3 Enactment of Constraint Model Instances

Thus far, we focused on the modeling and analysis of constraint models. In this section, the focus is on the *enactment* of these models, i.e. the actual execution environment. Section 6.3.1 introduces the notion of constraint model instances, sections 6.3.2 and 6.3.3 describe how states of constraints and instances can be monitored during the execution, respectively. Enforcing a correct instance execution and completion is discussed in sections 6.3.4 and 6.3.5, respectively.

6.3.1 Constraint Model Instances

The execution of constraint models is driven by the constraints. These constraints should ensure the correct execution of instances of the model, i.e. it should not be possible to violate mandatory constraints during execution and all mandatory constraints should be satisfied after execution. To ensure the correct execution of an instance, it is necessary to keep track of the execution trace of that instance. Formally, a *constraint workflow model instance* is defined as follows.

Definition 10 (Constraint model instance ci). A constraint model instance ci is defined as a pair $ci = (\sigma, cm)$, where:

- $\sigma \in \mathscr{T}^*$ is the instance's trace, and
- $cm \in \mathscr{U}_{cm}$ is a constraint model.

We use $\mathscr{U}_{ci}$ to denote the set of all constraint instances.

6.3.2 Monitoring Constraint States

The sequence of executed tasks for an instance is kept in its instance trace. Depending on the instance trace, constraints from the model can be *satisfied*, *violated*[2] and *temporarily violated*, which means that the constraint is currently violated, but can still be satisfied. Consider, for example, the *LTTL* ConDec model shown in Figure 6.9 on page 192. Although trace $\langle pickup \rangle$ violates both *1..** constraints, once

[2] In Section 6.3.4 we explain how this state is avoided during execution.

pickup and *bill* are executed all constraints will become satisfied. Hence, both *1..** constraints are *temporarily violated*. as long as the execution is in progress and more tasks can be executed, we distinguish these three states.

Definition 11 (Constraint state ν). Let $c \in \mathscr{C}$ be a constraint and $\sigma \in \mathscr{T}^*$ an execution trace. The function $\nu : (\mathscr{C} \times \mathscr{T}^*) \rightarrow \{satisfied, temporarily\ violated, violated\}$ is defined as:

$$\nu(\sigma, c) = \begin{cases} satisfied & \text{if } \sigma \in \mathscr{T}^*_{\vDash c}; \\ temporarily\ violated & \text{if } (\sigma \notin \mathscr{T}^*_{\vDash c}) \wedge (\exists \gamma \in \mathscr{T}^* : \sigma + \gamma \in \mathscr{T}^*_{\vDash c}); \\ violated & \text{otherwise.} \end{cases}$$

The state of a constraint can continuously be monitored via the automaton generated for its LTL formula. Every time a user executes a task, a transition in the automaton is triggered, and the automaton changes state. When the automaton is in an accepting state, then the constraint is *satisfied*. If the automaton is in a non-accepting state from which an accepting state is reachable, then the constraint is *temporarily violated*. Finally, if the trace cannot be 'replayed' on the automaton at all, then the constraint is and will continue to be *violated* for this instance.

6.3.3 Monitoring Instance States

During execution also the state of an instance is monitored. The state of an instance depends on the satisfaction of the mandatory constraints in the model.

Definition 12 (Instance state ω). Let $ci \in \mathscr{U}_{ci}$ be an instance where $ci = (\sigma, cm)$. The function $\omega : \mathscr{U}_{ci} \rightarrow \{satisfied, temporarily\ violated, violated\}$ of instance ci is defined as:

$$\omega(ci) = \begin{cases} satisfied & \text{if } \sigma \in \mathscr{T}^*_{\vDash cm}; \\ temporarily\ violated & \text{if } (\sigma \notin \mathscr{T}^*_{\vDash cm}) \wedge (\exists \gamma \in \mathscr{T}^* : \sigma + \gamma \in \mathscr{T}^*_{\vDash cm}); \\ violated & \text{otherwise.} \end{cases}$$

Recall that $\mathscr{T}^*_{\vDash cm}$ is the set of all traces that satisfy the conjunction of all mandatory formulas (cf. Definition 6 on page 190). Similar to the monitoring of constraint states, the state of an instance can be monitored using the automaton generated for the mandatory formula of the model.

Example 7 (Instance states). Table 6.2 shows how three instances (ci_1, ci_2 and ci_3) of the model shown in Figure 6.9 on page 192 change state during execution. The instance state is always deduced from the automaton's current state (Figure 6.9(b)). As long as the automaton stays in non-accepting states (i.e. S_0, S_1, ..., S_4), the instance is *temporarily violated* (tv). The instance becomes *satisfied* (s) only when the automaton reaches its accepting state S_5. If the instance trace cannot be replayed in the automaton, the instance is *violated* (v).

instance $ci_1 = (\sigma, cm)$			instance $ci_2 = (\gamma, cm)$			instance $ci_3 = (\delta, cm)$		
σ_i	state	$\omega(ci_1)$	γ_i	state	$\omega(ci_2)$	δ_i	state	$\omega(ci_3)$
`initial`	S_0	tv	`initial`	S_0	tv	`initial`	S_0	tv
delivery	S_2	tv	*pickup*	S_1	tv	*delivery*	S_2	tv
pickup	S_4	tv	*delivery*	S_4	tv	*bill*	☒	v
bill	S_5	s	*bill*	S_5	tv			
pickup	S_5	s						

Table 6.2: State change for instances of the ConDec model in Figure 6.9
('s'=*satisfied*, 'tv'=*temporarily violated*, 'v'=*violated*)

The instance state cannot always be directly induced from the states of its individual constraints. When all mandatory constraints are *satisfied*, the instance is also *satisfied*, and if at least one mandatory constraint is *violated*, the instance is also *violated*. However, in the presence of *temporarily violated* constraints and the absence of *violated* constraints, a deeper analysis of the constraints is needed to induce the instance state. Consider, for example, the instance of the ConDec model shown in Figure 6.13. The state of the individual constraints is depicted for execution trace $\langle C, A, B \rangle$. Observe that none of the mandatory constraints is *violated*. The *precedence* constraint is *satisfied*, because task *A* is executed before task *B*. Constraint *1*, which specifies that task *A* must be executed exactly once, is also *satisfied*. Constraint *2*, which specifies that task *A* must be executed exactly twice is *temporarily violated*, and would become *satisfied* if task *A* would be executed once again. However, if task *A* would be executed once again, then the *1* constraint would become *violated*. Since it is not possible to satisfy the mandatory constraints, the instance state is *violated*.

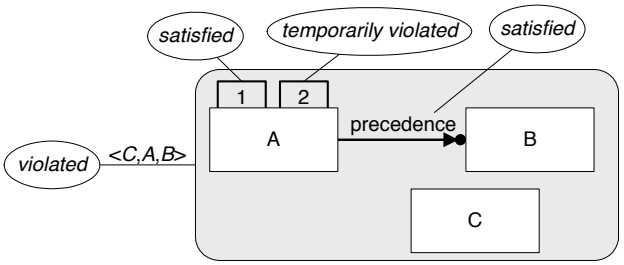


Fig. 6.13: An instance of ConDec model with trace $\langle C, A, B \rangle$

Note that the instance presented in Figure 6.13 cannot be *satisfied* because the instance model contains a conflict. Recall from Section 6.2.6 that when the model contains a conflict, then there is no execution that will satisfy the constraints from the model. The cause of the error is the combination of constraints *1* and *2*. Verifica-

tion can be used to detect and correct such errors. In the next section we explain how instances of correct constraint models can be executed without becoming violated.

6.3.4 Enforcing Correct Instance Execution

The goal of instance execution is to finish the execution in such a way that the instance is *satisfied*. In order for an instance to be executed in a correct manner, the execution of tasks that would eventually lead to the *violated* state of the instance should be prohibited. This can be done using the automaton generated for the mandatory formula of the instance. During execution the automaton changes state when tasks are executed. Given the current state in the automaton, tasks that are not implied by labels on transitions from this state are prohibited, other tasks are allowed.

Example 8 (Instance execution). Again, consider the ConDec model shown in Figure 6.9 on page 192. The automaton represents exactly all traces that satisfy this model. Initially no tasks have been executed and the automaton in state S_0. Due to the *precedence* constraint, only task *bill* should be prohibited, other tasks should be allowed to execute. This can be detected in the automaton by the absence of a *bill* transition from S_0 and the presence of *pickup* and *delivery* transitions. Naturally, the same approach is applied again to all states, as the automaton changes states due to execution of tasks.

6.3.5 Enforcing Correct Instance Completion

The goal of the execution of a constraint workflow model instance is to satisfy all mandatory constraints, so the state of the instance should be *satisfied* at completion. The automaton generated for the mandatory formula of the model is used to monitor the state of the instance. Recall that the state of the instance is *satisfied* if the automaton is in an accepting state. Now the completion of an instance is allowed, if and only if, the generated automaton is in an accepting state.

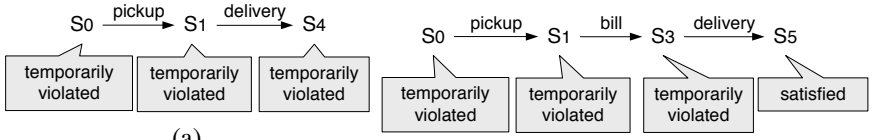
Enforcing the correct execution and completion of instances is done in the following way: (1) The states of all constraints are constantly monitored; (2) The state of the instance is constantly monitored; (3) The correct execution of instance is enforced by prohibiting the execution of tasks that *would eventually* bring the instance into the *violated* state; and (4) The completion of the instance is allowed if and only if the instance state is *satisfied*. All this can be done using the automaton generated for the mandatory formula of the instance's constraint model.

6.4 Dynamic Instance Change

In some cases, it is necessary that the model of the instance changes (i.e. by adding and removing tasks or constraints) although the instance is already executing and the instance trace might not be empty. We refer to such a change as a *dynamic instance change*. Workflow management systems that support dynamic change are called adaptive systems. For example, ADEPT is a workflow management system that uses powerful mechanisms to support dynamic change of procedural process models by allowing to add, remove and move tasks at runtime. The constraint-based approach offers a simple method for dynamic change that is based on a single requirement: the instance should not become *violated* after the change.

Due to the fact that, in dynamic instance change the trace of an instance remains the same whereas the model changes, it might happen that the instance state changes according to the new model (cf. Definition 12). For example, it is possible that after adding a mandatory constraint, the instance state changes from *satisfied* to *violated*, which is an undesired state. Therefore, instance change can only be successfully applied if the change does not bring the instance into the *violated* state. After a successful change, the instance continues execution with an updated model and the original trace. Automata generated for ConDec models enable easy implementation of dynamic change of ConDec instances: dynamic change of a ConDec instance is successful if the instance trace can be 'replayed' on the mandatory automaton of the new model.

Consider, for example, two instances $ci_1 = (\langle pickup, delivery, bill\rangle, cm_{LTTL})$ and $ci_2 = (\langle pickup, bill, delivery\rangle, cm_{LTTL})$ of the model cm_{LTTL} shown in Figure 6.9 on page 192. Figure 6.14 shows that ci_1 is *satisfied* by replaying their traces on the model automaton.



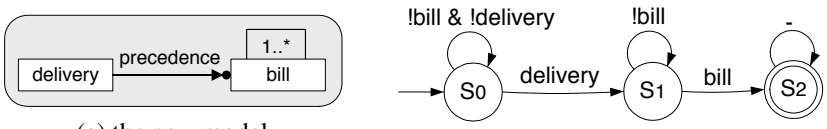
(a) $ci_1 = (\langle pickup, delivery\rangle, cm_{LTTL})$

(b) $ci_2 = (\langle pickup, bill, delivery\rangle, cm_{LTTL})$

Fig. 6.14: Replying traces of two instances of model cm_{LTTL} shown in Figure 6.9 on page 192 on the model automaton

Assume now that we attempt to dynamically change the cm_{LTTL} model in both instances by removing the *pickup* task and the *1..** constraint from task *delivery*, removing the *precedence* constraint and adding a new *precedence* constraint between tasks *delivery* and *bill*. The new model and its automaton are shown in Fig 6.15.

Figure 6.16 shows that this dynamic change involving is *successful* for ci_1 and *not successful* for ci_2. This figure shows how the instance trace is 'replayed' on the automaton. On the one hand, the trace of instance ci_1 can be replayed on the

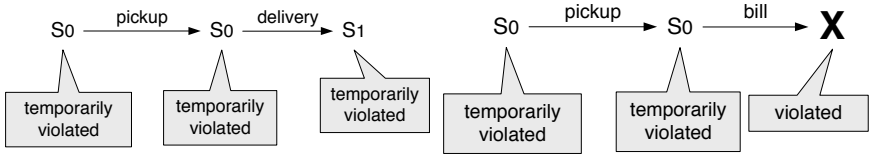


(a) the new model (b) automaton for the new model

Fig. 6.15: Applying a dynamic change to the model cm_{LTL} shown in Figure 6.9 on page 192

new automaton and, although the instance is *temporarily violated*, this is a valid dynamic change. On the other hand, the trace of instance ci_2 cannot be 'replayed' on the automaton because it is not possible to execute task *bill* from the state S_0, i.e. this change would *violate* the instance. Therefore, the dynamic change for instance ci_2 is not possible.

Note that, even though the task *pickup* is removed from the model after it was executed (the instance trace contains task *pickup*) the dynamic change for instance ci_1 is successful. This is due to the property that a trace that satisfies a model can contain tasks that are *not* in the model. The only consequence of removing task *pickup* from the model is the fact that it will not be possible to execute this task again in the future.



(a) the change is *successful* for ci_1 (b) the change is *not successful* for ci_2

Fig. 6.16: The dynamic change involving the model shown in Figure 6.15

6.5 Conclusions

To conclude this chapter, we first reflect on the differences between procedural and declarative languages. Then, we summarize the main capabilities of the approach presented in this chapter by using the flexibility taxonomy given in Section 6.1.

Neither procedural nor declarative workflows represent a better solution when it comes to automation of business processes. On the one hand, business processes that are repeatedly executed in the same manner can clearly benefit from a procedural workflow specification. For example, for the handling of insurance claims, tax declarations, customer orders, etc. a process where always the same procedure is followed is desirable. On the other hand, some processes must be executed in a way

that fits best with frequently changing and personalized circumstances. These processes are better specified in a declarative manner because they need a high degree of flexibility, i.e. it is desirable that they can be executed in various manners. For example, many processes from the health care domain need flexibility because they must be adjusted to specific circumstances and patients.

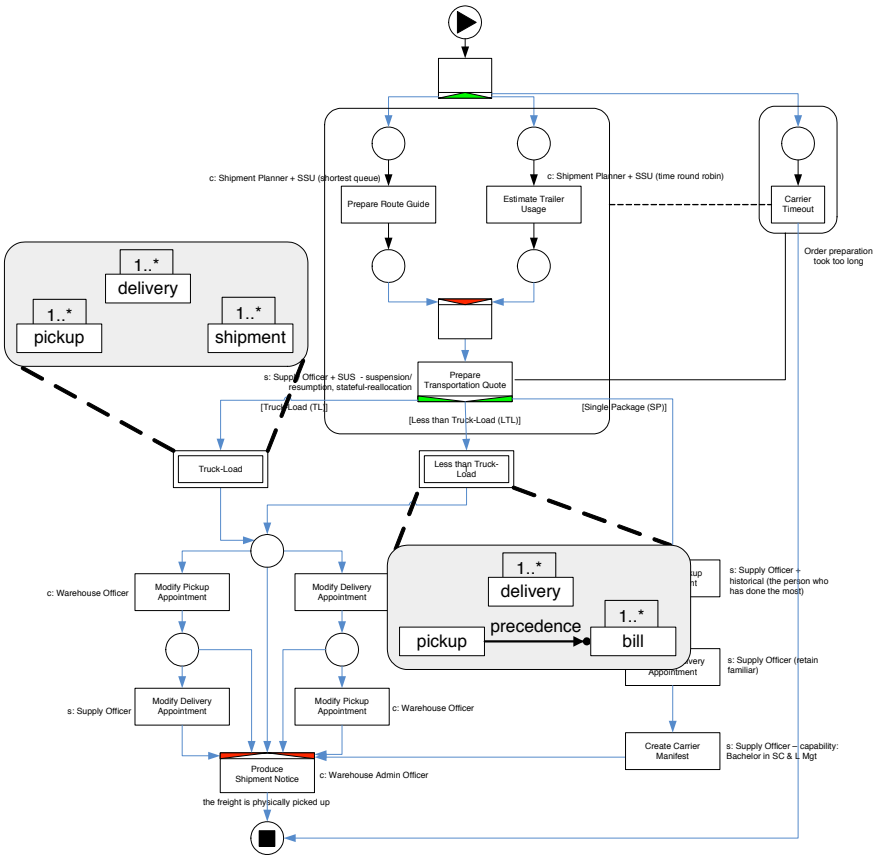


Fig. 6.17: Combining YAWL and declarative workflows

Moreover, for processes that are partly highly structured and partly loosely structured, a combination of both approaches can be chosen to specify the overall business process. For example, the declarative ConDec workflows *Truck Load* and *Less than truck Load* can be sub-processes of the *Carrier Appointment* net, as shown in Figure 6.17. Also, when necessary, a YAWL workflow can be a part of a declarative ConDec model. Using the service orientation it is possible to mix various modeling styles. A concrete implementation that enables creating decompositions of YAWL (e.g. procedural) and declarative models is presented in Chapter 12 of this book.

This chapter introduced a new way of modeling and enacting workflows. The described approach aims to be more flexible than the existing procedural approaches. Table 6.3 shows that declarative ConDec workflows can be used to support multiple types of flexibility.

flexibility	support
flexibility by design	model verification
flexibility by underspecification	monitoring instance states
flexibility by change	ensuring correct instance execution
flexibility by deviation	ensuring correct instance completion
	monitoring states of constraints

Table 6.3: Various types of flexibility and support of constraint-based workflows

First, because they can easily include a wide range of execution alternatives, ConDec workflows have a high degree of *flexibility by design*. For example, the ConDec models shown in figures 6.9, 6.10, 6.11 and 6.15 all have infinitely many execution alternatives (i.e. the sets of satisfying traces tof these models are infinite). Second, decomposing ConDec tasks to sub-workflows (e.g. YAWL workflows) allows for flexibility by *underspecification*. Moreover, this type of flexibility can be achieved either by explicitly specifying the decomposed sub-workflow in the parent-workflow, or by allowing for late specification (at runtime). Third, dynamic change of instances of ConDec models, which is described in Section 6.4, allows for *flexibility by change*. Fourth, the existence of optional constraints in ConDec models allows for flexibility by *deviation*.

Apart from being flexible, declarative workflows should and can support users in several ways. Constraint models can contain an arbitrary number of constraints that interfere in subtle ways, which can cause errors in models. Verification of constraint models provides an automated mechanism for detecting dead tasks and conflicts, as described in Section 6.2.6. In addition to the automated verification of constraint models, it is also possible to detect the exact combination of constraints that causes (each of) the error(s), which helps developing error-free ConDec models.

Execution of instances of constraint models is driven by constraints. In order to execute an instance in a correct way, it is necessary that, at the end of the execution, all mandatory constraints are *satisfied*, i.e. that the instance is *satisfied*. Executing activities in an instance may change the state of one or more constraints, and the instance itself. It is important that the instance state and states of all its constraints are constantly monitored and presented to users throughout the execution of the instance (cf. Sections 6.3.3 and 6.3.2). This helps users who execute instances of ConDec models to understand what is going on and which actions are necessary in order to execute the instance in a correct way. Also, in order to ensure a correct instance completion, it is necessary to allow completion only if the instance is *satisfied* (cf. Section 6.3.5).

Some actions of users might cause an instance (and its constraints) to leave the *satisfied* state. In some of these cases it is possible to take some actions that will eventually lead to a correct, i.e. *satisfied*, instance. We refer to this type of violations as *temporary violations*. In other cases, the instance becomes *permanently violated*, i.e. it becomes impossible to *satisfy* the instance in the future. Especially for instances with multiple constraints, it is very difficult for users to be aware of actions that will permanently violate the instance. Therefore, it is important to *prevent* users from taking actions that lead to permanent violation of instances of ConDec models (cf. Section 6.3.4).

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.

(a) Develop a ConDec model for this process using the constraint templates shown in Figure 6.5 on page 186.
(b) Write down the mandatory LTL formula for the Religion model.

Exercise 2. Figure 6.18 shows the automaton generated for the mandatory formula of the Religion model described in the previous exercise. Using this automaton, mark which of the following execution alternatives are possible in instances of the Religion model:

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

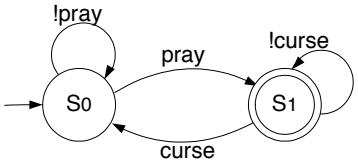
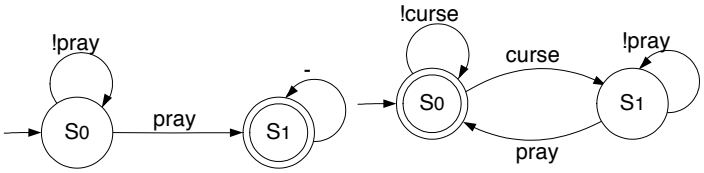


Fig. 6.18: The automaton generated for the mandatory formula of the Religion model

Exercise 3. Figure 6.19 shows automata generated for the two constraints from the Religion model from the first exercise. Using these two automata and the automaton generated for the mandatory formula of this model (cf. Figure 6.18), write

down how states of the instance and both constraints change while executing trace $\langle bless, curse, bless, bless, curse, become\ holy, pray, curse, pray \rangle$.



(a) one should pray at least once (b) one must always eventually *pray* after *cursing*

Fig. 6.19: Automata generated for the two constraints from the Religion model

Exercise 4. Consider an instance of the Religion model described in the first exercise, where the instance trace is $\langle bless, curse, bless, bless, curse, become\ holy, pray, curse, pray \rangle$. Assume that we attempt 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.

Can this dynamic change be applied to the instance at this moment? Explain why. Hint: use the automata generated for the mandatory formula of the new (i.e. changed) model shown in Figure 6.20.

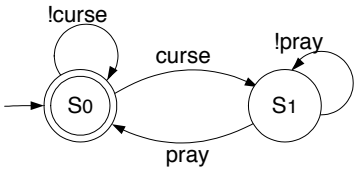


Fig. 6.20: The automaton generated for the mandatory formula of the Religion model after removing task *become holy* and the constraint specifying that *one should pray at least once*

Exercise 5. Consider an instance of the Religion model described in the first exercise, were the instance trace is $\langle bless, curse, bless, bless, curse, become\ holy, pray, curse, pray \rangle$. Assume that we attempt 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.

(a) Develop a ConDec mode for the new model using the appropriate constraint templates from Figure 6.5 on page 186.
(b) Can this dynamic change be applied to the instance at this moment? Explain why. Hint: use the automata generated for the mandatory formula of the new (i.e. changed) model shown in Figure 6.21.

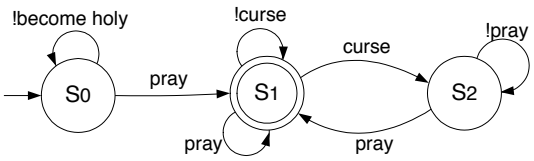


Fig. 6.21: The automaton generated for the mandatory formula of the Religion model after adding a constraint specifying that *task become holy cannot be executed before task pray*

Exercise 6. Develop a ConDec model 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 both be executed in the same instance.

Does this model have errors? Explain why. Hint: use the automata generated for the mandatory formula of this model shown in Figure 6.22.

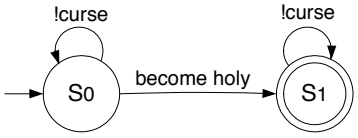


Fig. 6.22: The automaton generated for the mandatory formula of the Religion model described in this exercise

Exercise 7. Develop a ConDec model 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.

Does this model have errors? Explain why. Hint: the automata generated for the mandatory formula of this model is empty, i.e. it does not have any states or transitions.

Exercise 8. Which types of flexibility and support are available in declarative workflows? Explain which properties of declarative workflows enable different types of flexibility and support.

Chapter Notes

The trade off between flexibility and support and the taxonomy of flexibility used in this chapter has been addressed in [236, 117]. The positive influence of *declarative languages for workflow specification* of workflow flexibility has been discussed in [117, 200, 256, 73, 152].

More information on temporal logic, automata generation and model checking can be found in [60, 103]. A more elaborate discussion on the choice for LTL and usage of automata is given in [192]. For details on how LTL can be used for finite sequences see [104, 105, 192].

In this chapter, the focus was on the ConDec language and several ConDec templates were explained and used. The complete list of all ConDec templates can be found in [192]. Other similar constraint languages can be defined if needed. Fore example, more information about a constraint-based language for modeling web services DecSerFlow can be found in [19, 20].

More information about the procedural workflow management system ADEPT and its mechanisms to support dynamic change can be found in [205].