A Declarative Approach for Flexible Business Processes Management

M. Pesic and W.M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology,
P.O.Box 513, NL-5600 MB, Eindhoven, The Netherlands.
m.pesic@tm.tue.nl, w.m.p.v.d.aalst@tm.tue.nl

Abstract. Management of dynamic processes in an important issue in rapidly changing organizations. Workflow management systems are systems that use detailed process models to drive the business processes. Current business process modelling languages and models are of *imperative* nature – they strictly prescribe how to work. Systems that allow users to maneuver within the process model or even change the model while working are considered to be the most suitable for dynamic processes management. However, in many companies it is not realistic to expect that end-users are able to change their processes. Moreover, the imperative nature of these languages forces designer to over-specify processes, which results in frequent changes. We propose a fundamental paradigm shift for flexible process management and propose a more *declarative* approach. Declarative models specify what should be done without specifying how it should be done. We propose the *ConDec language* for modelling and enacting dynamic business processes. ConDec is based on temporal logic rather than some imperative process modelling language.

Key words: Workflow management, declarative model specification, dynamic workflow, flexibility, temporal logic.

1 Introduction

Companies need to adapt to rapid changes in their environment. In order to maintain agility at a competitive level, business processes are subjected to frequent changes. As software products that are used in companies for automatic driving of the business processes, workflow management systems (WFMSs) [2, 10] should be able to support the dynamics of business processes.

Workflow management systems are generic information systems which can be implemented in variety of organizations to manage the flow of work. In traditional WFMSs, every change of such a business process model is a time consuming and complex endeavor. Therefore, these systems are not suitable for rapidly evolving processes. The rigid nature of today's systems results from the way they model and enact the business process. A business process model can be seen as

a scheme which defines the 'algorithm' of the process execution. During the execution of the model, the system uses the business process model as a 'recipe' to determine the sequence (order) of tasks to be executed. Since the enactment of the model highly depends of the modelling technique and the modelling language, the later two play a determining role in the way the prescribed process can be executed. The weaker this prescription is, the easier it is to deviate from the prescribed process. However, most process models enforce the prescribed procedure without deviations. Flexible WFMSs should allow users to deviate from the prescribed execution path [12]. For example, traditional systems have the reputation to be too rigid because they impose a strictly predefined execution procedure.

The *case-handling* paradigm is usually considered as 'a more flexible approach' [5] because users work with whole cases and can modify to some extent the predefined process model. An example of such a system is FLOWer, which offers the possibility to open or skip work items that are not enabled yet, skip or execute enabled work items and redo executed or skipped items.

Systems for dynamic process management emerge as a necessity to enable dynamic changes in workflow management systems [12, 6]. As response on demand for dynamic business process management, a new generation of *adaptive* workflow management systems is developed [20, 17, 15]. When working with adaptive systems, users can change execution paths (e.g., in ADEPT users can insert, delete or move tasks in process models [17]).

Both in traditional, case-handling and adaptive systems, process models are presented in a process modelling language (e.g., Petri nets [18] ,Pi calculus [16], etc.), which defines the 'algorithm' for the process execution. Based on this algorithm, the system decides about the order of the task execution. Because process modelling languages like Petri nets and Pi calculus precisely prescribe the algorithm to be executed, the resulting models are *imperative*. Although case-handling and adaptive workflow systems allow for deviations/changes of models written in imperative languages, the result remains an imperative model. This can result in many efforts to implement various changes over and over again.

To abandon the imperative nature of contemporary WFMSs, we propose a paradigm shift by using a *declarative* language. In this paper we propose *ConDec* as a declarative language for modelling business processes. Unlike imperative languages, declarative languages specify the "what" without determining of the "how". When working with such a model, the users are driven by the system to produce required results, while the manner in which the results are produced depends on the preferences of users. Figure 1 characterizes the differences between classical imperative languages and ConDec. Figure 1(a) illustrates that ConDec specifies the "what" by starting from all possibilities and using constraints to approximate the desired behavior (outside-to-inside).

Imperative languages start from the inside by explicitly specifying the procedure (the "how") and thus over-specifying the process. To illustrate this, consider the ConDec constraint shown in Figure 1 (b). This constraint implies that the only requirement is that for a given case not both `A` and `B` are executed, i.e.,

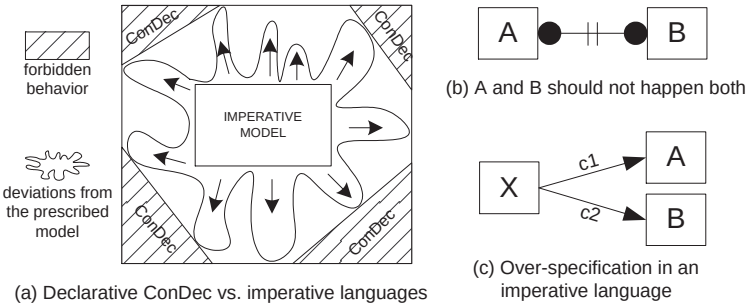

Fig. 1. A shift from imperative to declarative languages.

it is possible to execute `A` once or more times as long as `B` is not executed and vice versa. It is also possible that none of them is executed. In an imperative language one tends to *over-specify* this as shown in Figure 1 (c). Unlike Figure 1 (b), now a decision activity is introduced – `X`. This activity needs to be executed at a particular time and requires rules (e.g. conditions `c1` and `c2`) to make this decision. Moreover, implicitly it is assumed that A or B is executed only once. Hence, there is an over-specification of the original requirement and as a result (1) changes are more likely and (2) people have less choice when using the system.

In this paper we first introduce ConDec, a new language for modelling dynamic business processes (Section 2). In Section 2.1 we show an illustrative example of a ConDec model. Section 2.2 shows how a ConDec model can be enacted by a process management system. Related work is presented in Section 3. Section 4 concludes the paper and proposes future work.

2 Declarative Business Process Models

ConDec is a declarative language that can be used to build a wide range of models: from very 'strict' models that define the process in detail to very 'relaxed' models that state only what work should be done, without specifying how it should be done. Taking an optimistic approach, the simplest model to make in ConDec is a model specifying which tasks are possible. Users can execute such a model in their own preference – they can choose which tasks to execute and how many times, and in which order to execute tasks. However, such a simple too-relaxed model can be too 'anarchic' – there are often some rules that should be followed while working. These rules can also be added to a ConDec model - thus making it more rigid if desired. As an 'strict' extreme, a ConDec model can be supplied with such rules, that it behaves as an imperative model during the execution: the process is strictly prescribed and followed.

Initially, a ConDec model consists of a number of tasks, which are the possible tasks that can be executed. The notion of a task is the smallest unit of work,

like in any other workflow language. In an extreme case, a model consisting only of tasks can be instantiated and enacted. When working on such a process instance, users would be able to execute whichever tasks in whatever order. The next step in developing a ConDec model is to define the relations between tasks. The notion of relations between tasks in ConDec is considerably different than in a Petri net and other traditional workflow models. Relations between tasks in a Petri net describe the order of execution, i.e, *how* the flow will be executed. We refer to the relations between tasks in ConDec as *constraints*. A constraint represents a policy (or a business rule). At any point in time during the execution of the model, each constraint has a boolean value 'true' or 'false', and this value can change during the execution. If a constraint has the value 'true', the referring policy is fulfilled and vice versa – if a constraint has the value 'false', the policy is violated. At every moment during the execution of a process model, the model is evaluated to be correct or not. The execution of a model is correct at one moment of time if all its constraints have the value 'true' at that moment of time. Since some constraints can evaluate to 'false' at the very beginning of the execution, a constraint which has the value 'false' during the execution is not considered an error. Consider an example of a constraint that specifies that, each execution of task A is eventually followed by task B. Initially (before any task is executed), this constraint expression evaluates to *true*. After executing A the constraint evaluates to *false* and this value remains *false* until B is executed. This illustrates that a constraints may be temporarily violated. However, the goal is to end the execution of a ConDec model in a state where all constraints evaluate to *true*.

We use *Linear Temporal Logic* (LTL) [14] for declarative modelling of relations between tasks – constraints. In addition to the basic logical operators, temporal logic includes temporal operators: nexttime ($\bigcirc F$), eventually ($\Diamond F$), always ($\Box F$), and until ($F \sqcup G$). However, LTL formulas are difficult to read due to the complexity of expressions. Therefore, we define a graphical syntax for some typical constraints that can be encountered in workflows. The combination of this graphical language and the mapping of this graphical language to LTL forms the `declarative` process modelling language - ConDec. We propose ConDec for specification of dynamic processes.

Because LTL expressions can be complex and difficult to create for non-experts, we introduce *constraint templates* for creating constraints. Each template consists of a formula written in LTL and a graphical representation of the formula. An example is the "response constraint" which is denoted by a special arc connecting two activities A and B. The semantics of such an arc connecting A and B are given by the LTL expression $\Box(A \longrightarrow \Diamond B)$, i.e., any execution of A is eventually followed by B. Users use graphical representation of the templates to develop constraints in the ConDec model. Every template has an LTL expression of the constraint. It is the LTL expression and not the graphical representation that is used for the enactment (execution) of the model.

We have developed a starting set of more than twenty constraint templates. Constraint templates define various types of dependencies between activities at

an abstract level. Once defined, a template can be reused to specify constraints between activities in various ConDec models. It is fairly easy to change, remove and add templates, which makes ConDec an 'open language' that can evolve and be extended according to the demands from different domains. Currently, there are three groups of templates: (1) "existence", (2) "relation", and (3) "negation" templates. Because a template assigns a graphical representation to an LTL formula, we can refer to such a template as a formula. Presentation of all templates is not necessary for the demonstration of the language. For a full description of the developed templates, we refer the reader to the report [4]. Figure 2 shows an illustrative ConDec model consisting of three tasks: `A`, `B`, and `C`. Tasks `A` and `B` are tagged with a constraint specifying the number of times the task should be executed. These are the so-called "existence formulas" that specify how many times a task can be executed within one case. Since task `C` does not have a constraint on the number of executions, it can be executed zero or multiple times (0..*). The arc between `A` and `B` is a relation formula and corresponds to the LTL expression for "response" discussed before: $\Box(A \longrightarrow \Diamond B)$. The response constraint is satisfied if task `B` is executed after task `A`, but not necessarily as the next task after `A`. This constraint allows that some other tasks are executed after task `A` and before task `B`. The connection between `C` and `A` denotes the "co-existence" relation formula: $(\Diamond C \longrightarrow \Diamond A) \wedge (\Diamond A \longrightarrow \Diamond C)$. According to this constraint, if `C` is executed at least once, `A` is also executed at least once and vice versa. This constraint allows any order of the execution of tasks `C` and `A`, and also an arbitrary number of tasks between tasks `C` and `A`. Note that it is not easy to provide a classical procedural model (e.g., a Petri net) that allows for all behavior modelled Figure 2.

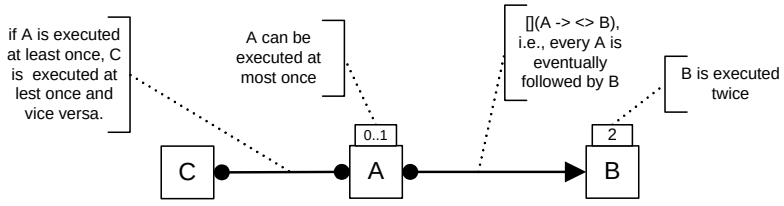

Fig. 2. An simple example of a ConDec model.

Note that a ConDec model can be developed as an imperative model when using the right constraints. For example, we developed a constraint template "chain succession" [4] that can be used to specify a direct succession between two activities.

We use an illustrative example to explain the concept of declarative languages and advantages of declarative process models. First, we develop a Petri net model of a simple example of on-line book purchasing. Next, we show how to develop a ConDec model for the same example. This will show how ConDec specifies the relations between tasks in a more natural and refined way.

Figure 3 shows a Petri net model of a simple proces for on-line book purchasing. A Petri net consists of *places* (represented by circles) and *transitions* (represented by rectangles). Transitions and places are connected with directed arcs – a transition has its input and output places. Initially, there is a token in the place `start`. A transition is enabled when there is a token in each of its input places. If a transition is enabled, it fires by consuming a token from each of its input places and producing a token in each of its output places. In our example, the transition `order` is enabled and will fire the first by consuming and producing one token. The produced token will enable transitions `accepted` and `declined`. If the order is not accepted, the transition `declined` will fire by consuming a token from the input place and producing a token in its output place – `end`. This process execution would end with the initial order being declined and the book would not be purchased. If the order is accepted, the transition `accepted` fires by consuming one token and producing two. This will result in transitions `receive book` and `receive bill` being enabled. We assume that the book and the bill arrive separately, because it is possible that the book arrives from the shipper and the bill from the bookstore. When the book arrives the transition `receive book` fires, and transition `receive bill` fires when the bill arrives. Only after these two transitions fire and produce tokens in two input places of the transition `pay`, the book will be payed by firing this transition, and thus ending the process of book purchasing.

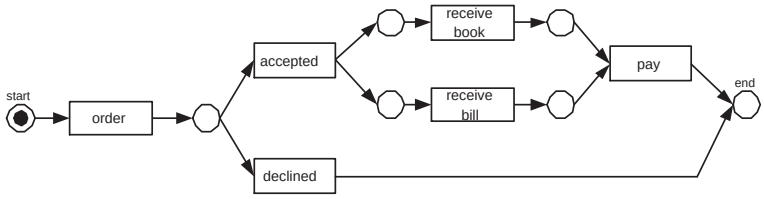

Fig. 3. Petri net - Purchasing a book

2.1 Declaring a Business Process in ConDec

In this section, we develop a ConDec model for the book purchasing example and explain the concept of constraints using this model. Figure 4 shows a ConDec model for the purchasing book example. We first define the same tasks like in the Petri net model in Figure 3. However, instead of defining the relations with Petri net arcs, we create a number of constraints, based on templates presented in [4]. First we develop a number of unary "existence" constraints. These constraints define how many times a task can be executed – the cardinality of a task. The graphical representation of these constraints indicates the cardinality for each task. Task `order` has the "existence" constraint "exactly_1" [4]. This constraint can be seen as the cardinality symbol '1' above the task `order`, and

it specifies that this task will be executed exactly once. All other tasks have the "existence" constraint "absence_2" [4]. The graphical representation for this constraint is the cardinality symbol '0..1' and it specifies that the task can execute at most one time. In this example, the book will be ordered exactly once, and this is why the task `order` has the cardinality '1'. The order can be `accepted` or not. Similarly, the order can be `declined` or not. This is why these two tasks have the cardinalities '0..1'. The book, the bill and the payment will not be executed more that one time. However, due to the possible declining of the order and errors, it might happen that these tasks do not execute at all. Therefore, tasks `receive book`, `receive bill` and `pay` have cardinality '0..1'.

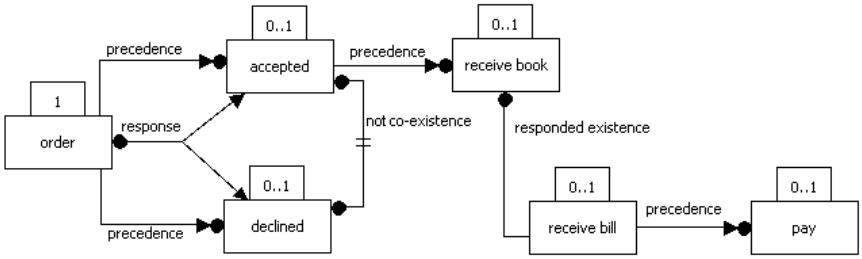

Fig. 4. ConDec - Purchasing a book

Next, we define relations between tasks. Several "relation" and "negation" [4] constrains are added to describe dependencies between tasks in the ConDec model in Figure 4. There is a branched `response` from the task `order`. It has two branches: one to the task `accepted` and the other to the task `declined`. Some binary "relation" and "negation" constraints can be extended with *branches*. The branched `response` in Figure 4 specifies that, after every execution of `order`, at least one of the tasks `accepted` or `declined` will eventually be executed. However, it is now possible that both tasks are executed, and to prevent this we add the `not co-existence` constraint between tasks `accepted` and `declined`. So far, we have managed to make sure that after the task `order` only one of the activities `accepted` and `declined` will execute in the model. One problem remains to be solved – we have to specify that both tasks `accepted` and `declined` can be executed only after the task `order` was executed. We achieve this by creating two `precedence` constraints: (1) one between the tasks `order` and `accepted` making sure that the task `accepted` can be executed only after the task `order` was executed, and (2) one between tasks `order` and `declined` makes sure that the task `declined` can be executed only after the task `order` was executed.

Further, we specify the relation between the activities `accepted` and `receive book`. In the Petri net model we had a strict sequence between these two activities. However, due to some problems or errors in the bookstore it might happen that, although the order was accepted (the task `accepted` is executed), the book does not arrive (the task `receive book` is not executed). However, we assume

that the book will not arrive before the order was accepted – the constraint `precedence` between the activities `accepted` and `receive book` specifies that the task `receive book` cannot be executed until the task `accepted` was executed.

The original Petri net specifies that if the bill arrives also the book will arrive, and vice versa. This may not be always true. The ConDec model in Figure 4 accepts the situation when the bill arrives even without the book being sent. This could happen in the case of an error in the bookstore when a declined order was archived as accepted, and the bill was sent without the shipment of the book. However, we assume that every bookstore that delivers a book, also sends a bill for the book. We specify this with the *responded existence* constraint between the `receive book` task and the `receive bill` task. This constraint forces that if the task `receive book` is executed, then the task `receive bill` must have been executed before or will be executed after the task `receive book`. Thus, if the execution of the task `receive book` exists, then also the execution of the task `receive bill` exists.

The constraint *precedence* between the tasks `receive bill` and `pay` means that the payment will be done after the bill was received. However, after the bill was received the customer does not necessarily pay, like in the Petri net model. It might happen that the received book was not the one that was ordered or it was damaged. In these cases, the customer can decide not to pay the bill. Note that the ConDec model in Figure 4 allows users to pay even before the book has arrived. If the order was accepted, then the book can be received. The bill can be paid as soon as the bill is received, and the bill can be received before the book. This allows for the execution of the model where the book arrives after the received bill had been paid.

Note that in this section we used a Petri net model as a starting point and showed the corresponding ConDec model after some relaxations. For real-life processes we propose *not* to do this. Starting with a classical process model may lead to the introduction of unnecessary constraints that limit users and flexibility. Because of a (potential) large number of different (types of) relations between activities, ConDecmodel can become to complex. Therefore, we recommend a careful selection of a small number of relations (constraints) that are appropriate for the desired ConDec model.

2.2 Enacting Declarative Models

While the graphical notation of constraint templates enables a user-friendly interface and masks the underlying formula, the formula written in LTL captures the semantics of the constraint. A 'raw' ConDec model consists of a set of tasks and a number of LTL expressions that should all evaluate to *true* at the end of the model execution. ConDec models can be executed due to the fact that they are based on LTL expressions, and every LTL formula can be translated into an automaton [14, 11]. The possibility to translate an LTL expression into an automaton and the algorithms to do so, have been developed for and extensively used in the field of *model checking* [14]. The Spin tool [13] uses an automata

theoretic approach for the simulation and exhaustive formal verification of systems, and as a proof approximation system. Spin can verify the correctness of requirements, which are written as LTL formulas, in a system model written in Promela (PROcess MEta LAnguage) [13]. A more detailed explanation about the automata theory and the creation of the Buchi automatons from LTL formulas is out of scope of this article and we refer the interested readers to [13, 14].

We can execute a ConDec model [4] by constructing an automaton [11] for each of the LTL expressions or constructing a single automaton for the whole model (i.e., construct an automaton for the conjunction of all LTL expressions). Figure 5 shows a simple ConDec model and the corresponding automaton[1]. This model consists of tasks `curse`, `pray`, and `bless` and the constraint `response` between tasks `curse` and `pray`. With this constraint in the model, when a person curses (`p2` is not an accepting state), (s)he should eventually pray after this (`p1` is an accepting state). Because there are no "existence" constraints in this model, all three activities can be executed an arbitrary number of times.

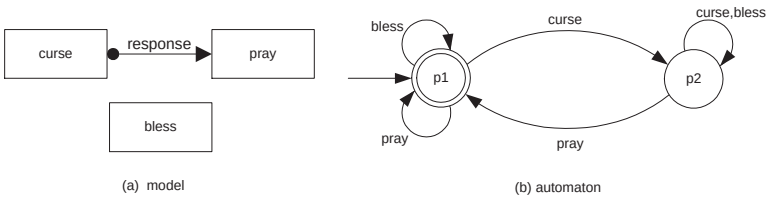

Fig. 5. A simple ConDec model.

Using automata for the execution of models with constraints allows for the guidance of people, e.g., it is possible to show whether a constraint is in an accepting state or not. Moreover, if the automaton of a constraint is not in an accepting state, it is possible to indicate whether it is still possible to reach an accepting state. This way we can color the constraints *green* (in accepting state), *yellow* (accepting state can still be reached), or *red* (accepting state can not be reached anymore). Using the Buchi automaton some engine could even enforce a constraint.

3 Related Work

Although many business processes can be characterized as dynamic processes, traditional rigid WFMSs can not cope with frequent changes. The flexibility of WFMSs can be seen as the ability to change or deviate from the business process and plays an important role in the extend to which such systems can support

[1] Note that the generated Buchi automaton is a non-deterministic automaton. For simplicity we use a deterministic automaton yielding the same behavior.

dynamic processes [12]. The nature of the modelling language itself determines the usability and flexibility of the system [3].

Case-handling systems have the reputation to be more flexible and more appropriate for dynamic business processes [5]. In such systems, users can open a whole case, and work on that case, while in traditional WFMSs, users work with multiple cases. When allowing users to work on whole cases, the system has the privilege to allow for much more maneuver in the process (e.g., opening, skipping and re-doing tasks in FLOWer).

The most advanced solution for dynamic processes management is a class of WFMSs that offers the possibility to change the business process model at runtime [20, 17, 15]. When working with *adaptive* WFMSs, it is possible to change the business process model on the general level (i.e., the change is applied for all business process instances), or on the instance level (i.e., the change is applied only on one instance). Systems like ADEPT [17] develop very complex workflow engines [19] that are able to handle inserting, deleting and moving tasks at runtime.

Declarative properties are used to check whether the model matches the modelled system in [7]. In this approach, causal runs of a Petri net model are generated by means of simulation. Process nets representing causal runs are analyzed with respect to specified properties. The three groups of properties are: facts (the property should *always* hold) [9], causal chains (immediate *causal dependency*) and goals (the property *eventually* holds). While this approach validates Petri net process models, our approach is used to generate and enact the model.

4 Conclusions and Future Work

Flexibility of WFMSs is tremendously influenced by their perception of the notion of a business process. In current systems, the model of a business process is seen as an imperative prescription of the procedure that should be followed during work. The present solutions for dynamic process management lie in a flexible execution of the model (i.e., case handling systems such as FLOWer), and in the possibility to change the model during the execution (i.e., adaptive systems such as ADEPT [17]). However, the approach and the model still remain the same: an imperative prescription of *how* the solution should be reached.

ConDec is a declarative language for modelling business processes. It specifies *what* should be done, and users can decide how they will do it. We take an optimistic approach where, in principle, anything is possible. That is, anything is possible unless we specify some *constraints*. Constraints represent policies that should not be violated. In a way, constraints specify what not to do instead of specifying how to work. This leaves a lot of room for the maneuver of users, who can make decisions and work in various ways with the same ConDec model.

Using automata theory and Linear Temporal Logic, ConDec models can be executed by an engine. Developing a system for management of ConDec models brings various challenges. We are currently developing a prototype of such

a system. Up to now, we have developed an editor where constraint templates can be defined and used to build a ConDec model. The ConDec model for the purchasing book example is developed in this tool (cf. Figure 6). The next challenge is to develop a complete workflow management system. This system will be used together with the YAWL system (`www.yawl-system.com`), where the YAWL language [1] deals with the structured workflows at a higher level. Moreover, the system will be linked to our process mining tool ProM [8] (`www.processmining.org`). This allows for the monitoring of ConDec flexible processes. Actually, ProM already offers an LTL checker for checking the ConDec constraints *after* execution.

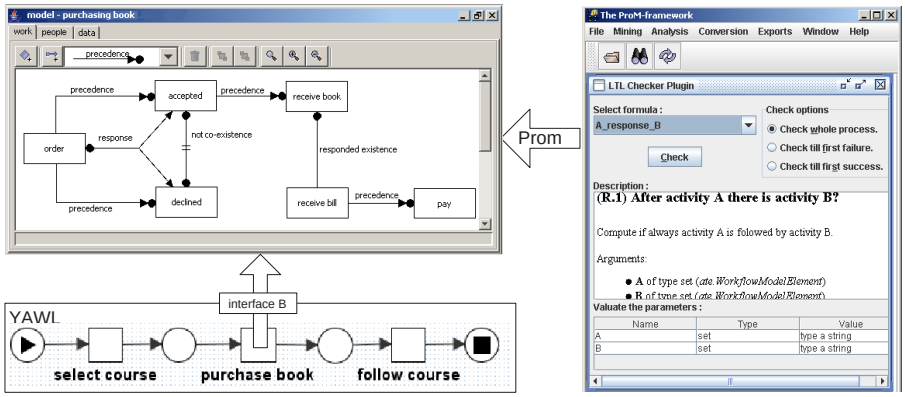

Fig. 6. The ConDec system with YAWL and ProM.

References

1. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004.
2. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
3. W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.
4. W.M.P. van der Aalst and M. Pesic. Specifying, discovering, and monitoring service flows: Making web services process-aware. BPM Center Report BPM-06-09, BPM Center, BPMcenter.org, 2006. http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-09.pdf.
5. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.

6. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. In *ER '96: Proceedings of the 15th International Conference on Conceptual Modeling*, pages 438–455. Springer-Verlag, 1996.
7. J. Desel. Validation of process models by construction of process nets. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 110–128, London, UK, 2000. Springer-Verlag.
8. B. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, Lecture Notes in Computer Science, pages 444–454. Springer-Verlag, Berlin, 2005.
9. H. J. Genrich and G. Thieler-Mevissen. The calculus of facts. *Mathematical Foundations of Computer Science 1976*, pages 588–595, 1976.
10. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
11. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
12. P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A comprehensive approach to flexibility in workflow management systems. In *WACC '99: Proceedings of the international joint conference on Work activities coordination and collaboration*, pages 79–88, New York, NY, USA, 1999. ACM Press.
13. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.
14. E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.
15. P.J. Kammer, G.A. Bolcer, R.N. Taylor, A.S. Hitomi, and M. Bergman. Techniques for supporting dynamic and adaptive workflow. *Comput. Supported Coop. Work*, 9(3-4):269–292, 2000.
16. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.
17. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
18. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
19. S. Rinderle, M. Reichert, and P. Dadam. Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
20. M. Weske. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, volume 7, page 7051, Washington, DC, USA, 2001. IEEE Computer Society.