Enhancing Flexibility in Business Process Management using Declarative Languages

José Marcelo A. P. Cestari, Gustavo D. Barddal, Eduardo A. P. Santos, Eduardo de F. R. Loures, Sauro Schaidt
Pontifícia Universidade Católica do Paraná, Industrial Engineering
Imaculada Conceição 1155, 80215 901 Curitiba, Brazil

Abstract

Traditional business process models are based on specifying exactly how to execute the process, e.g., all possibilities have to enter into the model by specifying its control-flow. The drawback of such models is that a rigid behavior is imposed. It means users of a traditional process model have no option unless the paths specified in such model. This kind of model is more adequate to represent repetitive processes, where the designer has to envision and to specify all the possible paths. However, many processes are characterized by some looseness in terms of its structure (e.g. healthcare, disaster management). Nowadays constraint-based processes approaches have received increased interest. In these processes, the control flow is defined implicitly as a set of constraints or rules, and all possibilities that do not violate any of the given constraints are allowed to be executed. Thus, any execution order of activities is possible provided that the constraints are not violated. Thereby, most of time the process execution is driven by user’s choice, thus reaching a great flexibility. The goal of this paper is to present an initial comparison between three declarative approaches used to represent constraint-based processes: Dynamic Condition Response Graphs, Declare/LTL and Supervisory Control Theory.

Keywords
Constraint-based processes, declarative languages, Supervisory Control Theory, Declare/LTL, Dynamic Condition Response Graphs.

1. Introduction

There are basically two types of business processes: pre-specified processes and knowledge-intensive processes [1]. Pre-specified processes are characterized by rigid structuring of order among their activities so that these processes offer little alternative of performing to the end user, there is usually a sequence of well-defined activities and all that is left to the user is following this sequence [2,3]. Pre-specified processes can be also called highly structured processes. Knowledge-intensive processes are characterized by the influence of the user in determining the appropriate sequence of the procedure. There are two types of knowledge-intensive processes: loosely specified processes and data-driven processes. These processes offer greater flexibility in implementation and, unlike pre-specified processes, they do not establish a strict order between activities, the user have the power to choose what must be the next activities to be performed based on your knowledge and professional experience. Loosely specified processes can be also called constraint-based processes [1-5].

Whatever the type of process, it has to be modeled according to its characteristics by a proper language, there are at least two types of process modeling language: imperative languages for highly structured processes and declarative languages for knowledge-intensive processes. Imperative languages provides tools that facilitate the rigid structuring required by highly structured processes while declarative languages make it possible for a user specifies few rules to model knowledge-intensive processes, each language must to provide the level of flexibility expected for every process [2-6]. There are four types of flexibility: variability, adaptation, evolution and looseness. Variability, adaptation and evolution are related to highly structured processes and looseness is associated to knowledge-intensive processes, thus, an imperative language has to provide some degree of variability, adaptation or evolution to highly structured processes whereas a declarative language must to offer looseness to knowledge-intensive processes.
The appropriate environment to execute any process is the Process Aware Information System (PAIS) [1-8]. There are PAIS driven to highly structured processes and PAIS driven to knowledge-intensive processes, when it is driven to highly structured processes it must to provide imperative languages as BPMN or Petri Net [1-8] among others and when it is driven to knowledge-intensive processes it must provide declarative languages as Declare/LTL [2,5,6], SCT [9-15], Case Management Model and Notation (CMMN) [16] and Dynamic Condition Response Graphs (DCRGraphs) [17,18] among others. The aim of this paper is to present some of the mentioned declarative languages - Declare/LTL, SCT, and DCRGraphs - and their application in a constraint-based process.

The goal of this paper is to present an initial comparison between three declarative approaches: Dynamic Condition Response Graphs, Declare/LTL and Supervisory Control Theory. This initial comparison should allow identifying how to make a more advanced comparison between these three approaches. Initially it will be showed few more concepts of process types and flexibility.

2. Types of Processes and Flexibility

As mentioned before, the processes can be divided in two groups: highly structured processes and knowledge-intensive processes [1], and there are two types of knowledge-intensive processes: based-constraint processes and data-driven processes. Another important classification of processes is whether they are based on activities or data. Highly structured and constraint-based processes are based on activities, every time an activity is performed the process evolves. Data-driven processes are based on data values, every time the value of a specific datum changes the process evolves [1].

The main characteristic of highly structured processes is their rigid structure on the order of activities so that the user has few flexibility or power to take a decision at runtime. The power is at build-time when the modeler can construct the process the way he thinks best, obeying the basic rules of process [3-7]. This is possible because the sequence of activities required for successful running of process is already known, i.e. the sequence is well-known before the runtime and the user, at runtime, must to obey it. In general terms, one can say that the logical of the processes are known before execution so they can be pre-specified and their models define, in the addition to order of activities, also controls, dataflow, organizational entities, data objects and documents accessed by users [1-6].

Despite having a rigid structure of activities order, highly structured processes also demand some level of flexibility. The dimensions of flexibility are variability, adaptation, evolution and looseness so that variability, adaptation, evolution are strongly linked to highly structured processes whereas looseness is a characteristic of knowledge-intensive processes [1-7]. Variability concerns to handle a same process in different ways according to different contexts named process variants. For example, the same process executed in different countries may require variations due to differences in regulations. In addition to regulations, process variants can emerge from product and service variability, different groups of customers and temporal differences. Adaptation is the capability to adapt a process instance to the special or exception situations which may or not have been foreseen. Adaptation can be planned or unplanned. It is planned when it is possible to foresee and plan the handling of the special or exception situation and it is unplanned otherwise. Evolution is the capability to change a process model according to changes of a process context. Such changes can emerge from legal or technical aspects among other factors. While adaptation foster alterations only at process instances at run time, evolution make modifications at processes models, at build-time, which also can apply to the running instances of a process.

As highly structured processes must be modeled by imperative languages because these facilitate the rigid structuration of activities order, it is also expected that the same languages supports the user to build processes models that can be suitable to every required flexibility in the real world [2-6]. An important characteristic of the imperative languages is that they require users to specify precisely the, beginning, middle and ending of the process. It is not possible, for example, to specify an isolated block of activities operating according to a specific rule and let it disconnected of the rest of process. Such block must be rigidly connected at some point of the process.

Knowledge-intensive processes are characterized by the influence of the users in determining the appropriate sequence of the procedure such that they provide a large level of flexibility at runtime. This is feasible because, as opposed to highly structured processes they do not force a rigid order between activities, the user has the power to choose which activities must to be performed next based on your knowledge and professional experience. While the highly structured processes describe which activities should be executed and the exact way in which they should be performed,
knowledge-intensive processes focus on the logic that controls the process by defining the activities to be executed and the constraints that prevents the forbidden behaviors of the process. So any sequence of activities that does not violate the constraints previously defined is permitted [1-8].

The flexibility dimension directly related to knowledge-intensive processes is looseness which encompasses three main aspects: non-repeatability, unpredictability and emergence. A process is non-repeatable when no instance is equal to the other, unpredictable when the sequence of activities depends on each situation and emergent when the sequence of activities will be known only during process execution [2-8]. Indeed, non-repeatability, unpredictability and emergence are strong features of knowledge-intensive processes.

As knowledge-intensive processes must be modeled by declarative languages because these facilitate the statement of few constraints giving high flexibility to the process, it is also expected that the same languages supports the user to build processes models that can be non-repeatable, unpredictable and emergent. Unlike imperative languages, declarative languages allow to model a set of activities independently of the others since they have a constraint in common, it does not matter the others activities that are not part of the restriction [1-8].

3. Constraint-based processes models

As constraints-based processes are knowledge-intensive it is impossible to specify their sequence of activities at build-time and only at runtime it will be possible define exactly it. This happens because the user will take the decision about which action must be performed according to the circumstances, his knowledge and expertise [5-8]. Due to this, these processes must be modeled by declarative languages which, in turn, favors the simple declaration of constraints between activities but without forcing a single sequence of performing. Among the declarative languages available we can cite at least four: Declare, DCRGraphs, CMMN and SCT and due to space limit we are going to show the main concepts and an example of application for Declare, DCRGraphs and SCT [2,5,6,9,15-18].

The process to be used as a running example is a developing customized corrugated packaging process. The corrugated packaging process has highly variable duration from two hours to several days according to their complexity. There is an intense negotiation between the departments of sales, product development and production considering various aspects such as priorities development of packaging, machine capacities, delivery time and machine productivity. Because of these characteristics, one cannot easily represent a well-defined sequence of activities before the runtime. The sequence will be defined just at runtime so the process should be described through activities and constraints.

The activities of the packaging process are showed in Table 1 and the constraints in Table 2. The execution of process instances can follow any sequence of activities since they do not violate any of the constraints. Due the page limit we are going to show only the modeling of R5 and R8 constraints. In order to show the difficulty to model this process as a highly structured process we will try to use the BPMN language [1-8] to model the R5 and R8 constraints. R5 constraint specifies that each time interrupt project (a9) is performed negotiate with sales (a2) must be performed afterwards, this means that interrupt project (a9) can be performed at any time and how often it is needed but it is not obligatory to perform it, negotiate with sales (a2) can be performed at any time and how often it is needed and it must be performed after interrupt project (a9) at least one time. As we can see there are more than one condition to consider inside a single constraint. So the language chosen to model the process must enable the user to do this the easy way.

As mentioned previously an imperative language requires user to specify the activities and their exact position in the process. So we cannot specify only a2, a9 and the relation between them only, we must make it clear which is the relation between a2, a9, their constraint and all the rest of the process. But there are other constraints and they do not specify a rigid order to the activities but logic relations as precedence, response, mutual exclusion constraint. Thus it is mandatory to combine all these constraints and produce a foreseeable sequence of activities representing the process as a whole. However, surely it will be a difficult task.

The Figure 1 shows that one can try to model a constraint on the a2 and a9 using the exclusive and or gateways of BPMN but there is no direct way to determine the correct interaction with the rest of the process and define which are the preceding and subsequent activities to a2 and a9. This difficulty is going to repeat to all the other constraints whenever someone wants to use imperative language to model this process. So one possible solution is to use declarative languages since they allow to model each constraint separately.
Table 1 – Activities of the developing customized corrugated packaging process

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1 allocate resource</td>
<td>Allocate the instance of process to a resource.</td>
</tr>
<tr>
<td>a2 negotiate with sales</td>
<td>Negotiate delivery time with sales department.</td>
</tr>
<tr>
<td>a3 structural design</td>
<td>Development of the packaging structure.</td>
</tr>
<tr>
<td>a4 graphic design</td>
<td>Development of the package printing.</td>
</tr>
<tr>
<td>a5 send sample</td>
<td>Send prototype structural design to the client.</td>
</tr>
</tbody>
</table>
| a6 negotiate with
  manufacturing           | Negotiate delivery time with the manufacturing department.                  |
| a7 deliver design         | Final delivery of the project.                                              |
| a8 cancel project         | Cancellation of the project (only by the seller).                            |
| a9 interrupt project      | Interruption of the project (only by seller or customer).                   |

Table 2 – Constraints of the developing customized corrugated packaging process

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>allocate resource (a1) must precede all other activities but cancel project (a8).</td>
</tr>
<tr>
<td>R2</td>
<td>negotiate with sales (a2) must precede all other activities but allocate resource (a1) and cancel project (a8).</td>
</tr>
<tr>
<td>R3</td>
<td>negotiate with sales (a2) must be performed at least one time.</td>
</tr>
<tr>
<td>R4</td>
<td>structural design (a3) must be performed at least one time.</td>
</tr>
<tr>
<td>R5</td>
<td>Each time interrupt project (a9) is performed, negotiate with sales (a2) must be performed afterwards.</td>
</tr>
<tr>
<td>R6</td>
<td>graphic design (a4), send sample (a5), negotiate with manufacturing (a6) and deliver design (a7) can be performed after structural design (a3) has been performed.</td>
</tr>
<tr>
<td>R7</td>
<td>deliver project (a7) or cancel project (a8) must be performed.</td>
</tr>
<tr>
<td>R8</td>
<td>deliver project (a7) and cancel project (a8) are mutually exclusive.</td>
</tr>
<tr>
<td>R9</td>
<td>deliver project (a7) or cancel project (a8) must be the last to be performed, when one of two has been executed no activity can be performed afterwards.</td>
</tr>
</tbody>
</table>

Figure 1 – Not modeled constraints necessary for the process.

3.1 Declare / Linear Temporal Logic

DECLARE is developed as a constraint-based system and uses a declarative language grounded in Linear Temporal Logic (LTL) for the development and execution of process models [5]. LTL can be used to define properties of sequences of tasks. Every trace represents an executed alternative where the tasks in the trace occur exactly in the order in which they appear. In the field of model checking, LTL is extensively used to check whether a system satisfies properties specified by LTL formulae. The business rules are formally described using LTL, allowing verification algorithms to identify if such rules are obeyed [5,6]. DECLARE allows for customized specification of relation types frequently needed to model constraints between tasks.

Considering LTL formulas can be difficult to understand by nonexperts, DECLARE provides a graphical representation of constraints that hides the associated LTL formulas from users of declarative workflows. This graphical representation is called constraints templates. Each template has a name, an LTL formula and a graphical representation. Templates can be used to create actual constraints for a specific process [2,5,6]. Declare offer four sets of constraints: existence, relation, negation and choice. Existence models specify how often or when a task can be performed. Relation models define some relation between two (or more) tasks. Negation models define a negative relation between tasks. Choice models are used to specify that one must choose between two or more tasks. The Figure 2 shows an example of modeling in Declare, the 1of3 constraint defines that one of three activities must be performed, this constraint is the choice type [6], the precedence constraint defines that activity 5 can be performed only after...
activity 4 to be performed, this constraint is the relation type, and the existence constraint defines that activity 6 must be performed at least once, this constraint is the existence type.

Figure 2 – 1of3 constraint modeled in Declare

Figure 3 shows the two constraints R5 and R8 modeled through Declare and it presents the build-time and runtime of each constraint. The R5 build-time presents the using of response operator to define whenever a9 is performed a2 must be performed afterwards. The R5 runtime presents a condition in which a9 was performed and consequently a2 performing is pending and the constraint is temporarily violated. The R8 build-time shows the using of the exclusive choice operator to define if a7 is completed so a8 will never be completed and vice-versa. The R8 runtime presents a condition in which neither of a7 and a8 is completed and the constraint is temporarily violated.

Declare/LTL divide each activity in three events: started, completed and canceled and the modeling is done on these events. This is not perceptible at build time but only at runtime, this condition provides a greater power to the user when process is running. Also during the runtime the status of each constraint is flagged by colors, when the constraint is satisfied the color of operator is green, whenever a constraint is temporarily violated the color of operator becomes orange and when the constraint is met again the color of the operator becomes green again.

Figure 3 - Constraints R5 and R8 modeled by Declare

3.2 Dynamic Condition Response Graphs

The idea of Dynamic Condition Response Graphs (DCRGraphs) emerged from the use of LTL notation [2,5,6]. LTL notation in the representation of constraints tends to be quite complex, making the use by the end user hard [6,17,18]. This difficulty has motivated the development of a new notation that has a high power of representation associated with easily understood by the end user [17,18]. To achieve this goal the authors decided to adopt a smaller library of relationships between elements, but through this it would be possible to express all necessary relations in modeling the constraints of a process. Thus, the result was a fairly simple model on many elements which made it the easier model to be used and understood.
The main components of the DCRGraphs are presented in Table 3. The main components of the DCRGraphs are presented in Table 3. There are four basic relations in the table, but there are still another two relations not mentioned: the milestone relation and the nested activities. Any other different relation involving one or more activities must be constructed from these six structures [17,18].

The Figure 4 show the two constraints R5 and R8 modeled through Dynamic Condition Response Graphs language and presents the build-time and runtime of each constraint. The R5 build-time presents the using of response operator (blue arrow) to define whenever a9 is performed a2 must be performed afterwards. The R5 runtime presents a condition in which a9 was performed (green “V” on the activity) and consequently a2 performing is pending (red “!” on the activity).

The R8 build-time shows the using of the exclusion operator (red arrow) to define if a7 is completed so a8 will never be completed and vice-versa. The R8 runtime presents a condition in which a7 is completed (green “V” on the activity) and the a8 is disabled (dashed line on activity).

The Table 3 – Main elements of DCRGraphs

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/diagram.png" alt="Diagram" /></td>
<td>Event/Activity: similar to others notations represents atomic unit of working in the process. The solid line in the A1 activity means that it is included and the dashed line in the A2 activity means that it is not included.</td>
</tr>
<tr>
<td><img src="https://example.com/diagram.png" alt="Diagram" /></td>
<td>Condition relation: specifies that the A1 activity is a condition for A2 activity to be performed. The red symbol in the A2 means that A2 cannot be performed and it remains until A1 is performed.</td>
</tr>
<tr>
<td><img src="https://example.com/diagram.png" alt="Diagram" /></td>
<td>Response relation: specifies that if the A1 activity is performed then A2 activity must be performed afterwards.</td>
</tr>
<tr>
<td><img src="https://example.com/diagram.png" alt="Diagram" /></td>
<td>Inclusion relation: specifies that each time A1 activity is performed and the A2 activity is not included then A2 activity must be included.</td>
</tr>
<tr>
<td><img src="https://example.com/diagram.png" alt="Diagram" /></td>
<td>Exclusion relation: specifies if A1 activity is performed then A2 activity must be excluded.</td>
</tr>
</tbody>
</table>
Figure 4 - Constraints R5 and R8 modeled by DCRGraphs language.

3.3 Supervisory Control Theory (SCT)

The Supervisory Control Theory (SCT) was developed as a formal methodology for automatic synthesis of optimal controllers for discrete event systems (DES). In this theory, it is assumed that a set of activities has an uncontrollable behavior that might violate some required properties (e.g. security system events). This behavior must be modified through an agent (supervisor) in order to achieve a set of specifications (or that certain restrictions are not violated) [9,10]. To do so, the supervisor acts on the set of activities by preventing the generation of some events and allowing others. The Local Modular Control (LMC) approach, unlike the classical SCT approach, provides a framework for decentralized control, in which local monitors act on parts of the overall plant [14,15].

In order to apply the SCT for constraints-based processes it is necessary to obtain two models: (1) the model of the system under control and (2) the constraint model. To each task is assigned an automaton that represents its behavior. We assume that each task is modeled as an automaton with two states: (0) an initial state where the task is not running (one instance was not started) and (1) other state where the instance is being processed. With the start event (ti), the task is started (state 1 is reached.) When the task ends, signaled by the occurrence of the event complete (ci), or cancel (xi), it returns to state 0 [11-13]. Figure 5 shows the automaton that represents a task ti.

![Figure 5 - The automaton that represents a task ti](image)

Considering using the LMC approach, the first step for the local synthesis of supervisors is to get the set of activities for each constraint. Using the SCT algorithms is possible to obtain local supervisors, each guaranteeing the constraint expressed by the corresponding automaton. The synthesis of the local supervisor is established considering corresponding constraint and your set of activities [14,15]. Figure 6 show a constraint and its local supervisor, this constraint specifies that when a1 activity is performed so a2 activity must be performed afterwards and it is called precedence constraint [6,13]. There are more constraints than this but because of the pages limit we introduce only this one.

![Figure 6 - Constraint and its local supervisor](image)

Figure 7 show the automata of the two constraints R5 and R8 and their respective supervisors synthesized according to SCT. R5 must be modeled as \( \text{response}(a9,a2) \) and R8 as \( \text{not}\_\text{coexistence}(a7,a8) \). The automata of supervisors indicate all the sequences of events that can occur without violating the constraints. The supervisor for \( \text{response}(a9,a2) \) ensures whenever \( a9 \) is completed (c9 occurs) it is obligated \( a2 \) to be completed (c2 occurs) to reach a marked state. The supervisor for \( \text{not}\_\text{coexistence}(a7,a8) \) ensures if \( a7 \) is completed (c7 occurs) so \( a8 \) cannot be completed (c8 does not occur) and if \( a8 \) is completed (c8 occurs) so \( a7 \) cannot be completed (c7 does not occur).
4. Discussion

As mentioned before, highly structured processes are better modeled by imperative languages as BPMN and Petri Net. However for modeling constraint-based processes it is suitable to use declarative languages as Declare/LTL, DCRGraphs, SCT and CMMN approach.

Considering the running example shown in Section 5 this paper presented a process to developing customized corrugated packaging. Due the characteristics of the process presented in Section 3, it is classified as a constraint-based process and it is more adequately modeled by a declarative language. We chose the \textit{R5 and R8} constraints (the \textit{response} and \textit{not coexistence} constraints, respectively) to model using three declarative languages: SCT approach, Declare/LTL and DCR Graphs. First, we attempt to model the two constraints using BPMN, but we show that it is very hard, or even impossible, build an imperative model which describes such constraints. This happens because it
is impractical to unforeseen all the possible sequences of activities considering the constraints to be imposed. We can say that BPMN does not provide the proper resources to model a constraint-based process.

Declare/LTL has a user interface which enables to handle activities as atomic unit and constraints patterns as relations between them. The Declare/LTL approach offers large amount of constraints patterns in its modeling environment, this can be a vantage as a single pattern can comprise various logical constructs and the modeling can be done with a reduced quantity of operators and symbols, this can mean a leaner and comprehensible modeling. Declare/LTL is a language create precisely to constraints-based processes enabling the user to model each constraint and its corresponding activities in an isolated manner, essential feature to treat such processes. Thus we were able to model R5 and R8 constraints and we found that Declare/LTL is language that can provide looseness flexibility to a process and it can be used to model based-constraint processes [1-6]. Besides Declare/LTL to offer a large set of constraints it also hides the LTL complexity from user, this indeed is a great vantage because the user does not need to know LTL specifications.

DCRGraphs approach offers reduced amount of logical constructs when compared to Declare. There are a minimum set of the logical operators that meet the basic logical relations as response, precedence, inclusion and exclusion and from these the user must be able to model any process. When compared to the Declare, this smaller amount of elements can offer the vantage to the user not worry about to get knowledge of a great number of patterns but only a few of them. However, the user may build more complicated constraints from this small group of operators and the modeling can be more complicated than in the Declare [17,18]. Like Declare, DCRGraphs hides the complexity of logical specifications from user which does not need to know them to model a process.

In the SCT approach, we worked directly with automata to model activities, constraints, supervisors, and this enabled us to envision accurately the event sequences that the process can track. SCT approach permits to visualize, in advance, which entire sequences are possible to happen. SCT approach is also a declarative language that provides a degree of looseness flexibility when modeling a constraint-based process [9-13].

The Supervisory Control Theory allows automatic synthesis of supervisors so that restrictions are not violated. One advantage of this approach is that the solution is optimal in the sense that it is minimally restrictive (disabling is performed only when necessary) and nonblocking (there is always a sequence of events that leads to the completion of an activity). This approach also allows supervisors to be designed quickly and automatically when, for example, these new restrictions or modifications are necessary. The approach guarantees that the solution obtained is correct by construction.

5. Conclusion
We discussed the main differences among highly structured and intensive-knowledge processes and the importance to comprehend this in order to choose their correct modeling languages. More precisely, we seek to study Declare/LTL, DCRGraphs and SCT declarative languages to model a constraint-based processes model. In this type of process is more difficult to envision all possible sequences of events as the process evolves according to the user knowledge at runtime. These processes require the looseness flexibility, in other words, the declarative language must provide looseness flexibility enough to model adequately these processes. Declare/LTL, DCRGraphs and SCT approaches are indeed declarative languages and consequently they allow to model a constraint-based process but each provide a different degree of looseness flexibility to the process and a future study may be undertaken to try to assess what the consequences this difference may bring to a process. In order to begin the investigation of the flexibility degree of each approach, our next work will be a comparison between the automaton generated by each of them, so we expect obtain more information that allow to show more precisely if there is some equivalence between the languages generated by each approach.

References