Mixing Paradigms for More Comprehensible Models

Michael Westergaard[1,2]⋆ and Tijs Slaats[3,4]

[1] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
[2] National Research University Higher School of Economics,
Moscow, 101000, Russia
[3] IT University of Copenhagen
Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
[4] Exformatics A/S, Lautrupsgade 13, 2100 Copenhagen, Denmark
`m.westergaard@tue.nl`, `tslaats@itu.dk`

Abstract. Petri nets efficiently model both data- and control-flow. Control-flow is either modeled explicitly as flow of a specific kind of data, or implicit based on the data-flow. Explicit modeling of control-flow is useful for well-known and highly structured processes, but may make modeling of abstract features of models, or processes which are highly dynamic, overly complex. Declarative modeling, such as is supported by Declare and DCR graphs, focus on control-flow, but does not specify it explicitly; instead specifications come in the form of constraints on the order or appearance of tasks. In this paper we propose a combination of the two, using colored Petri nets instead of plain Petri nets to provide full data support. The combined approach makes it possible to add a focus on data to declarative languages, and to remove focus from the explicit control-flow from Petri nets for dynamic or abstract processes. In addition to enriching both procedural processes in the form of Petri nets and declarative processes, we also support a flow from modeling only abstract data- and control-flow of a model towards a more explicit control-flow model if so desired. We define our combined approach, and provide considerations necessary for enactment. Our approach has been implemented in CPN Tools 4.

1 Introduction

Petri nets provide a powerful formalism for specifying many real-life systems, including business processes. Petri nets excel by having a duality between data and events, yielding a very powerful tool for specifying how data flows though a system. Control-flow of a Petri net model is often modeled explicitly as flow of a specific kind of data, similar to a program counter in traditional programming. Alternatively, the control-flow is not modeled at all, and just manifests as a consequence of the data-flow. As such, we call a Petri net model a procedural model

⋆ Support from the Basic Research Program of the National Research University Higher School of Economics is gratefully acknowledged.

as the control-flow when disregarding data is close to procedural programming languages: modelers specify *how* to solve a problem. An example where such a language is useful, is classical filling of forms, such as a patient registration process at a hospital.

Declarative specification of processes is an emerging trend for specifying especially business processes, but it has not seen massive use in practice. Declarative models often focus primarily on flow of control, but instead of explicitly modeling control-flow as a program counter, constraints between the different events are described. Declarative languages resemble declarative programming languages: modelers specify what the *intention* of the control-flow is, but not how to achieve that. An example where such languages are useful, is a patient treatment process at a hospital; here, many tests need to be run and many treatments are possible. There is no strict order of tests and treatments, but some treatments are incompatible with each other, and some treatments need follow-up treatments.

Declarative processes are typically better at describing highly dynamic environments, where actions can take place in many different orders, or early in the design, where the exact order of events is unknown. On the other hand, Petri nets are far better at modeling data-flow, and the strict control-flow model makes it easier to model processes with a strict and well-understood control-flow, which also makes it much easier to extract experiences from the model to an eventual implementation [10]. In this paper, we propose to merge two declarative approaches, Declare [15,19] and DCR graphs [7,13], with a high-level Petri nets formalism, colored Petri nets [9], to obtain a formalism that offers the best of both worlds. We aim to do so in a manner that makes it possible to use all three formalisms completely independently of each other or to mix all three formalisms in a single model. This makes it also possible to initially construct a purely declarative model, optionally with data, and during refinement make it more procedural as applicable. If we consider a hospital, this allows us to make a single model comprising both patient registration, diagnosis, and treatment. The reason for using both DCR graphs and Declare for the declarative parts is that the languages have different focus areas: Declare provides higher level primitives, often resulting in more comprehensible models, but DCR graphs do not suffer from the computational overhead of detecting conflicts necessary to ensure correct execution of Declare models.

We introduce our combined approach in Sect. 2, including pointers on how to allow analysis of combined models and our implementation in CPN Tools 4 [17]. In Sect. 3, we sum up our conclusions and compare with related work.

2 Combined Models

In this section we informally introduce our hybrid model and its semantics, and provide analysis considerations important for implementation. Actual implementation details are deferred to the next section.

The idea behind the hybrid approach is to identify transitions of CP-nets with tasks of Declare models and events of DCR graphs and then allow places

and arcs (with annotations) from CP-nets, constraints from Declare models, and the relations from DCR graphs to be added to the model to constrain the possible executions.

The reason for including these three languages is that CP-nets is a widely used procedural formalism with a strong theoretical background. It provides great support for data flow, both theoretically and practically in the form of tool support. We also prefer to use both Declare and DCR graphs for specifying the declarative parts of the model. We choose these two languages because they are on the surface very similar, yet they have different focus areas.

Declare offers a large set of contraints which have been identified as commonly used in business processes, making it well-suited to the BPM domain. DCR Graphs on the other hand aim to provide a formal language for describing processes in general, containing only 4 basic constrains while still being formally more expressive than LTL.

By providing both languages, we can use pre-existing tools and techniques to analyze our combined models, automatically switching from one kind of analysis to the other as needed.

An execution is considered accepting if it is accepting for all three underlying models. In other words, the execution should be accepting for the CP-net that one gets when removing all Declare constraints and DCR Graph relations, it should be accepting for the Declare model one gets by removing all places, arcs and DCR Graph relations and it should also be accepting for the DCR Graph model that one gets by removing all places, arcs and Declare constraints. Formally, we can define the semantics of all three languages in terms of transition systems, and the semantics of the combined language is just the synchronization of the three transition systems we get from the individual semantics by projecting the combined model onto each of the three languages.

2.1 Analysis

We would like to provide a step-wise semantics for combined models. This is necessary for efficient simulation. For CP-nets isolated, this is easy because every state is accepting, so if a binding element sequence is enabled, the execution will inevitably end in an accepting state. For Declare models and DCR graphs, this is possible using a preprocessing step: we simply compute the prefix automaton, which is possible as they both have a semantics yielding finite automata, and only allow a transition if it leads to an accepting state in the prefix automaton. For the combined models, however, this is not in general decidable. While we can construct the transition system product of the 3 automata on the fly, we cannot employ any of the techniques to ensure we can end up in an accepting state: as not all states of Declare models and DCR graphs are accepting, not all states of the product are necessarily accepting, so we cannot just execute any enabled binding element sequence and be sure to end in an accepting state. As the transition system we get from a CP-net is not necessarily finite, neither is the product, so we cannot compute the prefix automaton. If either the CP-net model is bounded (yielding a finite state space), or the Declare model automaton

and the DCR graph automaton only have accepting states, we can use the fact that these properties are preserved by transition system product and use the appropriate technique. Otherwise, we must settle for weaker guarantees.

When talking about runtime verification of Declare models [12], each constraint can be in not just the two states *satisfied* and *violated*, but also in two weaker states, where a constraint is only temporarily satisfied or violated, but future execution may violate or satisfy it. Only when the execution is terminated, is it possible to collapse possible satisfied/violated constraints into their (permanently) satisfied/violated counterparts.

For DCR Graphs we do not keep track of the state of individual constraints, instead we have a current marking and on execution check that the executed event is enabled and calculate the new marking. If a marking contains no pending included responses, the DCR Graph is in an accepting state and the process can terminate. Feedback to the user consists of showing which events have occured before, are enabled and need to occcur.

Simple simulation. As demonstrated in [18], even if Declare is decidable, constructing the automaton for the full system can be very time and memory consuming – it is exponential in the number of constraints. To avoid this overhead, we can instead create an automaton for each individual constraint. If we do so, we can avoid ever violating individual constraints, while retaining fast simulation (we can update the model in the initial state in constant time). CPN Tools offers a mixed mode, where simulation and editing are interleaved. This is useful for testing and debugging, but requires that the simulation can resume very quickly, so performing an operation that is exponential in the size of the model may be undesirable (at least for large models). By constructing the individual automata, we can avoid ever (permanently) violating constraints, and for some constraints, e.g., init, this is sufficient. For other constraints, this provides a best-effort but fast simulation mode (we can update the model in constant time in the initial state). We call this the *simple simulation* approach. The simple simulation approach for DCR Graphs comes down to doing basic runtime verification as described previously: checking that an event is enabled in the current marking and calculating a new marking can be done in constant time.

Smart simulation. As shown in [11] some Declare constraints can be in a conflicted state: they are not violated, but also cannot all be (possibly) satisfied at the same time. We can only catch this if we construct the automaton for the full Declare model. By making the product explicit, we can compute the prefix automaton. Unfortunately, the product of the prefix automata is not sufficient. As demonstrated in [18], this can still be fairly fast for moderately-sized models (in the order of seconds for models with 30-50 constraints). We call this *smart simulation*: we avoid executing any transition that would lead to a conflicted state. We can also do *smart simulation* of the DCR Graph constraints by building the finite automaton that corresponds to the graph and only allowing the execution of events that can lead to an accepting state (i.e., the DCR Graph

does not contain any deadlocks). The efficiency of this approach has not been investigated structurally yet, but for the models that we have considered to date the state space tends to be modest. We can compute the product of the automata from the underlying Declare model and the underlying DCR Graph model to ensure that the two kinds of declarative constraints cannot conflict with other as well.

Data-aware simulation. When combining declarative models with CP-nets, we get an additional type of conflicts: a declarative constraint might require some task to be executed, while the the CP-net model blocks its execution (f.e. because of a missing token). Smart simulation cannot catch this on its own as it only looks at the declarative (and computable) parts of the model. To handle such situations we need constraints that yield automata which only have accepting states, which severely limits the usability, or that the state-space is finite. Thus, *data-aware simulation* is as hard as state space analysis.

For simple examples with small domains, we can just generate the state-space and perform the synchronization, typically in minutes or hours. If the state-space is larger but still finite, we can perform many simulations using smart simulation and discard any not ending in an accepting state, similar to how simulation is used for bug-finding until a final verification often is used. After computing the synchronization, it can be stored efficiently (often only few states are conflicted). If we deal with large domains, it is sufficient if we can generate an equivalence-reduced state-space. This is for example the case if all types are integers or reals, and we only compare all tokens with integers, similarly to region or zone reduction for timed automata.

2.2 Implementation

We have implemented our combined models in CPN Tools 4 [5,17]. CPN Tools 4 adds support for *simulator extensions* [17], a mechanism which makes it possible to extend CPN Tools using Java code. Each extension can add operations to CPN Tools and also modify existing operations. The integration comprises 4 parts: GUI extension, syntax check extension, enabling restriction, and analysis. In Fig. 1, we see how combined models look and are constructed in CPN Tools.

3 Conclusion

In this paper we have introduced a new approach to modelling workflows combining the procedural formalism colored Petri nets, and the two declarative formalisms, Declare and DCR graphs. The combined formalism can be seen as adding declarative control-flow to CP-nets or as adding data-flow to declarative formalisms. Declarative approaches are typically better for abstract descriptions or highly dynamic processes, where procedural approaches are better for well-known and structured processes. Combining the two allows us the best of both

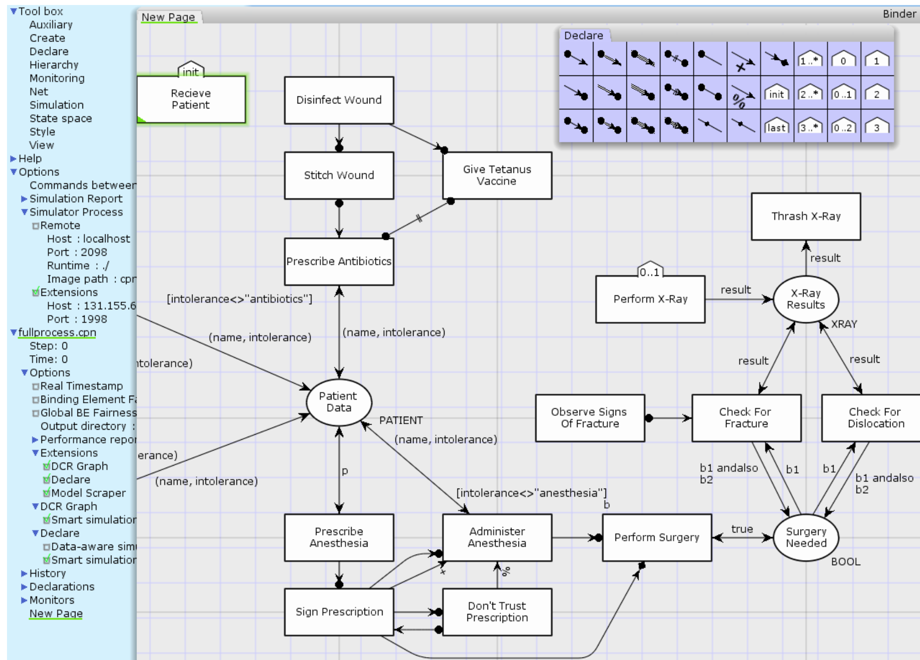

Fig. 1: Declare and DCR graphs in CPN Tools 4.

worlds and allows declarative processes to also deal with data. We have considered what is needed to provide simulation that avoids future conflicts in efficient ways, and introduce three modes of simulation: simple, smart, and data-aware, where the simple mode only avoids individual conflicts, smart avoids conflicts not related to data, and data-aware makes sure that even in the presence of data, all executions can terminate successfully. We have briefly introduced our implementation in CPN Tools.

3.1 Related Work

The *Guard-Stage-Milestone* (GSM) model [8] by Hull et al, which originated from the work on artifact-centric business processes [3], takes an approach to modelling business processes where a process consists of a number of (possibly nested) stages, which in turn contain a number of tasks. A stage also has guards and milestones; it is activated by satisfying its guards and through performing the tasks in the stage its milestones can become enabled, which can then in turn satisfy the guards of other stages. We see the GSM model as a hybrid model combining procedural and declarative structures in a single language, whereas our approach is based on combining existing procedural and declarative languages. In [14] the authors introduce a declarative version of the Computer-Interpretable Guidelines (CIG) language for modelling clinical guidelines, they

conclude that because both the procedural and declarative languages have their disadvantages it would be best to combine them into a single model, but leave this for future work. In [16] the authors have examined the understanding of procedural and declarative process models by users. In their conclusions they note that while it appears that procedural models are more comprehensible, it remains uncertain to what extent this is caused by participants being skewed towards procedural models because of their general acceptance and availability. They do not consider a hybrid approach using both procedural and declarative concepts. In [6] Fahland bridges the gap between declarative and procedural workflow approaches by proposing a general compositional mechanism for translating declarative workflow models to procedural workflow models. He exemplifies the general approach by giving a translation from Declare to Petri nets. The main difference to our approach is that while in [6] a declarative model is translated to a procedural version to facilitate using existing modeling, analysis and management techniques, we aim to combine the declarative and procedural approaches and provide tools and techniques for the hybrid approach.

3.2 Future Work

Here we have assumed that a user creates and refines a model. Another approach is to have the tool do that. For example, a precedence(A, B) constraint is trivially modeled using a single place of type boolean, initially marked by false. Then A changes the value indiscriminately to true and B checks that the value on the place is true. not co-existence(A, B) can be implemented using a place with three possible values: {A,B,UNDECIDED}. init(A) is less elegant, but can be implemented using, e.g., inhibitor arcs. This will not catch conflicts, but we can do that (and translate all constraints in a uniform way) by constructing the finite automaton either just equate the states of the automaton with new places or use a (not data-aware) process mining algorithm [1] or the theory of regions [4] to construct a Petri net for the control flow. Future work includes investigating the best way to make such a translation (semi-)automatically.

The current approach works as long as we describe a single run of a single process. That is, our example in the figures handles one treatment of one patient. It would be interesting to investigate multiple instances, which would essentially be a folding of the current model. We could also have multiple processes communicate, e.g., like Proclets [2], resulting in a more artifact-driven result. Using a translation from declarative constraints would essentially result in process-partitioned CP-nets in the sense of [10], which means it would be possible to automatically derive code from the resulting model. It would also be interesting to look into making a proper notion of a process in CP-net models, where instances cannot just terminate in any state. This would also include adding a notion of instances to the declarative languages.

References

1. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W., Barthelmess, P., Ellis, C.A., Wainer, J.: Workflow Modeling using Proclets. In: Proc. of CoopIS'00. pp. 198–209. LNCS, Springer (2000)
3. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Proc. of BPM'07. pp. 288–304 (2007)
4. Carmona, J., Cortadella, J., Kishinevsky, M.: A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In: Proc. of BPM. LNCS, Springer (2008)
5. CPN Tools webpage. Online: `cpntools.org`
6. Fahland, D.: Towards analyzing declarative workflows. In: Autonomous and Adaptive Web Services. Dagstuhl Seminar Proceedings, vol. 07061, p. 6. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI) (2007)
7. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Post-proc.of PLACES 2010 (2010)
8. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath, III, F.T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculin, R.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: Proc. of WS-FM'10. pp. 1–24. Springer-Verlag, Berlin, Heidelberg (2011)
9. Jensen, K., Kristensen, L.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
10. Kristensen, L., Westergaard, M.: Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In: Proc. of FMICS. pp. 215–230. LNCS, Springer (2010)
11. Maggi, F., Montali, M., Westergaard, M., Montali, M., van der Aalst, W.: Runtime Verification of LTL-Based Declarative Process Models. In: Proc. of RV. LNCS, vol. 7186, pp. 131–146. Springer (2011)
12. Maggi, F., Montali, M., Westergaard, M., van der Aalst, W.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Proc. of BPM. LNCS, vol. 6896, pp. 132–147. Springer (2011)
13. Mukkamala, R.R.: A Formal Model For Declarative Workflows - Dynamic Condition Response Graphs. Ph.D. thesis, IT University of Copenhagen (March 2012)
14. Mulyar, N., Pesic, M., van der Aalst, W.M.P., Peleg, M.: Declarative and procedural approaches for modelling clinical guidelines: Addressing flexibility issues. In: Proc. of BPM'07. pp. 335–346 (2007)
15. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
16. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: Proc. of ER-BPM '11. pp. 383–394 (2011)
17. Westergaard, M.: CPN Tools 4: Multi-formalism and Extensibility. Submitted to Petri Nets 2013
18. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: Proc. of BPM. LNCS, vol. 6896, pp. 83–98. Springer (2011)
19. Westergaard, M., Maggi, F.: Declare: A Tool Suite for Declarative Workflow Modeling and Enactment. In: Business Process Management Demonstration Track (BPMDemos 2011). CEUR Workshop Proceedings, vol. 820. CEUR-WS.org (2011)