Abstract

Workflow management tools may be used in many domains, to guide and direct processes, to support monitoring activities and to increase organizational efficiency. In safety critical applications such as healthcare, it is essential that the workflow is error-free, that is, for every run of the workflow, necessary requirements are satisfied and unwanted situations do not occur. However, most tools and frameworks which support workflow specification are not formal enough to allow automated verification and/or are not user-friendly enough for the domain experts to use. In this paper we discuss an extension to a model-driven engineering (MDE) based approach to workflow modelling. Our goals are to provide a framework that can model typical healthcare protocols, by means of a visual tool which can be easily understood by the users (usually clinicians), and to articulate and model check behavioural properties. With this tool, the user can input a workflow model and workflow properties which are defined diagrammatically; the model is automatically transformed to DVE code (the DiVinE model checker’s language) and the properties to LTL-formulae. If the workflow model is not valid wrt. a property, the tool provides a visual representation of a path which is a counter-example that can be easily analysed for debugging. The inherent agility of the MDE approach is especially useful in a healthcare setting because workflows, even for widely used clinical guidelines, generally need to be customized to local settings and updated frequently due to changing conditions, new medications or new research.

Keywords: Workflow modelling, User-friendly verification, Model checking, Model-driven engineering.

1. Introduction

Healthcare protocols are becoming more complex and are constantly updated due to new regulations and the introduction of new treatment methods, medications and technologies. Health procedures have traditionally been written as textual guidelines in natural languages, which can run to hundreds of pages, incorporating heavily annotated flow-type diagrams which use non-standard, and frequently confusing, routing notations [1]. Patient data is increasingly being stored electronically, methods such as workflow models are...
being used to formally describe treatment procedures and workflow management systems are used to guide the processes, providing both monitoring features and data for decision support. Healthcare, by nature, is a safety-critical process: it is essential to ensure the correctness of workflow protocols. While a number of initiatives have used model checking to verify healthcare processes, model checking was originally developed for use by computer scientists, and current software requires a great deal of programming expertise and familiarity with temporal logic[2]. To fully benefit from such formal methods, intuitive user interfaces are required that permit domain experts to interact with the software using their domain concepts.

In earlier work [3, 4, 5] we proposed a diagrammatic framework, based on model-driven engineering (MDE) [6, 7], which allowed us to automatically generate workflow simulation software from diagrammatic models. The diagrammatic models are easily understood by domain-experts, and the MDE approach which allows models to be easily customized to deal with specific settings, new treatment procedures, etc., provides the agility required for healthcare procedures. In this paper we extend our earlier language with a loop construct to permit cycles in workflow protocols (a feature of many health protocols) and focus on developing user-friendly tools for the model checking problem. We present a diagrammatic framework with which healthcare personnel, without being highly trained in logic or programming, can specify properties about their workflow; our tool model checks the workflow against the properties, and if it fails, the tool provides a user-friendly visualization for analysis.

In Section 2 we review our workflow modelling language. In Section 3 we discuss correctness of workflow models, and explain our user-friendly technique to define workflow properties and visualise counter-examples. Sections 4 and 5 present some related and future work and conclude the paper.

2. Workflow Modelling

Workflow models were originally introduced as means to optimize resources used in assembly line production environments; later, workflows were used to document and analyse complex work processes to ensure their correctness. In this paper we focus on the latter, with emphasis on how to specify workflow properties and visualize their counter examples. Our overall goal is to develop a framework which is user-friendly to domain experts. We start this section by giving an example of a workflow from the healthcare domain (see Fig. 1). The workflow illustrates a simplified scenario for cancer treatment. After an initial examination, the patient will have an MRI examination and a blood test. After an evaluation of the results of the two tests, the physician will decide which procedure the patient should follow (either Procedure A or Procedure B). After finishing this procedure, a second evaluation will occur to determine if the patient should continue with a drug treatment or if this workflow should end. If the drug treatment is chosen, once the drugs are finished, a blood test is done and an evaluation occurs to determine if the drug treatment should be repeated or if the workflow should end. Note that if the drug treatment is repeated, the blood test and the evaluation will also be repeated; i.e., the workflow will be in a loop. When it has been decided that the drug treatment should terminate, the workflow ends.

We want to stress the different purposes of the sample workflow presented above. The workflow could be used as a basis for formal reasoning done by experts in formal methods or model checking. The workflow could also be used as a working tool for practitioners in a domain, in our case, healthcare. These two

Figure 1: Sample workflow model
purposes may give rise to a conflict: as a basis for reasoning, one needs machine readable workflows with precise semantics, while healthcare practitioners need an intuitive and “easy to work with” language. Our aim is to propose a framework that is both intuitive enough to be used by healthcare practitioners and formal enough to be used to specify and verify interesting properties of healthcare workflows.

**Workflow Modelling Language.** We now give a short presentation of the workflow modelling language, for more details see [5]. As seen in Figure 1, our workflow models are diagrammatic models describing in which order specific work tasks should be executed. Each task is represented by a box. If there is an edge $T_1 \rightarrow T_2$ starting in task $T_1$ and ending in task $T_2$, then task $T_1$ must be performed before task $T_2$. Special binary constraints on forks (joins) specify splits (respectively, merges) of workflow branches. In fact, joins and forks could be extended in the standard way to arbitrary triples, quadruples, etc. We have three kinds of splits: [and_split], [or_split] or [xor_split], and three kinds of merges: [and_merge], [xor_merge] or [or_merge]. The meaning of these constraints are as usual: both branches have to be executed in an [and_split]; exactly one branch has to be executed in an [xor_split] and one or two branches have to be executed in an [or_split].

The syntax and semantics of the workflow modelling language is already given in [3, 4, 5]. Here we recall some of the most important details. The modelling language is defined using the Diagram Predicate Framework (DPF) [8] and implemented using the DPF Workbench [9]. In DPF, a modelling language is given by a metamodel and a diagrammatic predicate signature (see Fig. 2). The metamodel defines the types and the signature defines the predicates that are used to formulate constraints by the users. A model in DPF consists of an underlying graph, and a set of constraints. DPF supports a multi-level metamodelling hierarchy, in which a model at any level can be regarded the metamodel for models at the level below it. We say that a model conforms to (or is an instance of) a metamodel if the model’s underlying graph is typed by the metamodel’s underlying graph, and if the model satisfies the constraints defined in the metamodel. In DPF, the semantics of a (meta)model is given by the set of its instances. Both due to space limitations and to make our discussion easily comprehensible for those familiar with typical workflow jargon and unfamiliar with DPF, we now give a simple overview of our modelling hierarchy (for details of DPF see [8, 9, 3, 4, 5]).
In the design of our modelling language we have three modelling levels: M2, M1 and M0 (see Fig. 2). The metamodel of our workflow modelling language (which is at level M2) consists of a node Task and an arrow Flow. Simply put, this means that we can define a set of tasks together with the flow relations between these tasks. The signature $\Sigma_2$ of the workflow modelling language consists of a set of routing predicates such as [and_split], [and_merge], [xor_split, c], [xor_merge], etc. (see examples in Table 1).

We introduce one new predicate [NodeMult, n] in this paper, which will be used to restrict the number of instances ($n$) a task could have; i.e., it controls how many times a task could be performed, and can be used as an upper bound in a loop or cycle in the workflow model.

From the metamodel at level M2 and the signature $\Sigma_2$ with routing predicates, we can create a modelling language for the definition of “workflow models”. These workflow models, for example the model in Fig. 1, which conform to the metamodel at level M2, are located at level M1.

Given a specific workflow model at level M1 (like the one in Fig. 1) and the predicates $<E>$, $<R>$ and $<F>$ (where $<E>$, $<R>$, and $<F>$ denotes that a task instance is enabled, running, and finished, respectively) collected in a signature $\Sigma_1$ (see Table 2), we create another modelling language which we use to define “workflow states”. We refer to $<E>$, $<R>$ and $<F>$ as “task states”. These workflow states are located at level M0, and conform to the workflow model. Beginning with a state at level M0 that may be referred to as an instance of the workflow model we generate states by applying the so-called model transformation rules – which are referred to as rules for short (see Table 3). For example rule $r_1$ takes an instance of a task from $<E>$ to $<R>$.
Table 3: Two rules used for generating workflow states, the complete set of rules can be found in [5]

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$\mathcal{L}_0 \rightarrow \mathcal{L}_1$</th>
<th>$\mathcal{R}_0 \rightarrow \mathcal{R}_1$</th>
<th>$\mathcal{K}_0 \rightarrow \mathcal{K}_1$</th>
<th>$r_2$</th>
<th>$\mathcal{L}_0 \rightarrow \mathcal{L}_1$</th>
<th>$\mathcal{R}_0 \rightarrow \mathcal{R}_1$</th>
<th>$\mathcal{K}_0 \rightarrow \mathcal{K}_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>$R$</td>
<td></td>
<td>$R$</td>
<td></td>
<td>$E$</td>
<td></td>
</tr>
</tbody>
</table>

and rule $r_2$ takes an instance of a task from $<\mathcal{R}>$ to $<\mathcal{F}>$. A workflow run is represented by an execution path in the state space of the workflow model; i.e., by a sequence of rule applications.

Let us give a brief example, which uses our sample workflow model (see Fig. 1). The first (start) state, $\text{state}_0$, of this model, which is at abstraction level M0, would be $\text{InitialEvaluation}<E>$. This state consists of only one task instance. The colon notation in $\text{InitialEvaluation}<E>$ indicates the typing morphism and that we are talking about an instance of the task $\text{InitialEvaluation}$. The next state, $\text{state}_1$, obtained by applying the rule $r_1$ is $\text{InitialEvaluation}<R>$. The following state, $\text{state}_2$, obtained by applying the rule $r_2$ is then $\text{InitialEvaluation}<F>$. Then the states $\text{state}_3$ and $\text{state}_4$ will be, respectively,

$\text{InitialEvaluation}<F> \rightarrow \text{BloodTest1}<E>$ and $\text{InitialEvaluation}<F> \rightarrow \text{BloodTest1}<R>$.

We continue applying the rules in this way and will finally reach an "end state". We call a state an end state if 1. no more rules are applicable, and 2. at least one task instance without outgoing arrows is labeled $<\mathcal{F}>$. Note that if 1 is the case but not 2, we have a deadlock.

Thus the sequence of rule applications starting with the start state, has given us one possible execution path. Note that in this particular example, we have just a few possible paths. For example we could apply $r_2$ on $\text{state}_4$ and change the task state of BloodTest1 to $<\mathcal{F}>$, or apply $r_2$ and change the task state of MRI to $<\mathcal{R}>$. The order of applying these two rules would give us two different execution paths. However, in general, several different rules may be applicable at any state and choosing which one to apply will be determined (to some extent) by a controlling mechanism (details of such controlling mechanisms may be found in [3, 4, 5]).

**Loops.** When the user creates a cycle in the model, the modelling editor automatically identifies it as a loop. When a cycle is defined, the user must define the maximum number of iterations (upper bound). This number is defined as the parameter $n$ of the predicate $[\text{NodeMult}, n]$. The constraint formulated by this predicate will be automatically put on all nodes in the cycle. The predicate $[\text{NodeMult}, n]$ restricts the maximum number of instances which a task in the cycle could have. The $[\text{NodeMult}, n]$ predicate will be violated if the workflow loops more than $n$ times. We will restrict the minimum number by using looping conditions. If the looping condition is satisfied, we go through the loop, and can continue looping until either the multiplicity predicate $[\text{NodeMult}, n]$ on one of the nodes in the loop is violated, or until the looping condition is false, whichever comes first.

When loops are present, the workflow model will need to be augmented, depending on the situation (see Fig. 3 for some sample loop cases and how the model must be augmented). First, if there is an outgoing

![Figure 3: Sample loop cases](image-url)
arrow from a task in the loop to a task outside the loop, the arrow must be “XORed” with the looping arrow (exit point). That is, the flow must choose exactly one of the following: continue in the loop, or go out of the loop. Second, if there is an incoming arrow to a task in the loop from a task outside the loop (entry point), the arrow must be “ORed” with the looping arrow. That is, the task inside the loop will get enabled by either the flow from inside or the one from outside the loop. Currently our workflow language supports loops that can be expressed as regular expressions; in order to achieve this we shall apply the following restriction “The entry point and exit point of a loop must be the same task”.

3. Correctness of Workflow

Correctness is essential in safety critical domains such as healthcare. There are several different notions of correctness for workflow models. We say that the workflow is (i) type correct if it is correctly typed by its metamodel; (ii) valid if it does not violate basic generic properties such as: 1) the workflow must not have any deadlock, 2) the workflow must not have any livelock, 3) the workflow must terminate properly (see [10]); and (iii) verified if it satisfies all user-defined LTL properties.

In our framework, whether a workflow is type correct or not is ensured by the DPF Workbench at design time. Fig. 4(a) shows a type incorrect workflow since the flow F1 does not have any target, which is contrary to the meta-model. The validity of a workflow model is ensured by a model checker also at design time. The generic properties are verified dynamically at design time by our system using the DiVinE model checker. Fig. 4(b) shows a type correct but invalid workflow model as it has a deadlock (T4 will never be enabled). To ensure that the workflow satisfies the LTL properties specified by the user, we again use the DiVinE model checker. Fig. 4(c) shows an example of a type correct, valid but unverified workflow model: it did not satisfy the user-defined property G(T4.Running → (T2.Finished & T3.Finished)). In Fig. 4(c), the developer made a mistake while modelling the workflow; instead of using an [and_merge], she used an [xor_merge] which introduced a problem in satisfying the above mentioned LTL-formula. We now describe our user-friendly tool for verifying LTL-formulas and show how to visualize a counter-example to aid in debugging.

Transformation of Workflow Models to Model Checker Code. To ensure correctness of the workflow models that are specified in our modelling language we automatically transform the workflow models to the DiVinE model checker’s code DVE. The transformation is implemented as a code generation project with a template to generate DVE code supporting several constraints, such as tertiary and quaternary AND, OR, XOR, etc. The code is freely accessible under an EPL license from [11].

Property Definition. For defining specific properties of a workflow model we introduce a user-friendly editor for property definition where a user can see a workflow model and use patterns to define properties that must hold on every workflow execution path. Expert users may write properties as LTL-formulas at the bottom window (see Fig. 5) but we also provide GUI facilities to draw patterns for use by the non-expert. If we pay attention to the semantics of LTL operators we see that they are essentially patterns (see Fig. 6(a)). It may be hard for domain users to learn and/or write LTL-formulas but it should be a lot easier for them to identify or specify patterns in the property editor, as human minds are very good dealing with visual patterns. Using the property editor one may draw patterns and bind the propositions with workflow tasks. In Fig. 5, propositions P and Q are bound with ProcedureA and ProcedureB’s task states, respectively. A

Figure 4: Samples of incorrect workflow models: (a) type incorrect (b) deadlock (c) failure of LTL-property
complex proposition may be built using logical connectives (e.g., $\&\&$, $||$, $\rightarrow$). Patterns drawn by the user are translated to LTL-formulas and are checked against the workflow model by a model checker. Basic patterns (e.g., Must occur, Always occurs, Until, etc.) are also provided in the editor as templates which beginner users may use to specify properties. In future, we will incorporate an English translation of the patterns which the user can use to better understand the specified properties. Fig. 6(b) shows some sample patterns and their LTL-formulas. The dotted lines between states indicate an arbitrary number states without any specified properties.

**Visualisation of Counter-examples.** This section explains our approach to counter-example visualisation. As mentioned, our framework facilitates a user-friendly, diagrammatic specification of both workflow models, and temporal properties against which the models are checked. We generate DVE code from the models, and generate LTL-formulae (and then DVE code) from the properties. We run the generated DVE code together with the LTL-formulae on the DiVinE model checker. If the model checker detects any problems, i.e., if the model does not fulfil some properties, it provides a counter-example for each of them.

From the counter-example provided by DiVinE, we generate workflow states. We generate one workflow state for each state in the counter-example and use a 3D model viewer which shows the workflow model and its states. By using the 3D model viewer one can analyse a counter-example by traversing from one state to another (see Fig. 7). The workflow model is shown above the workflow state to better visualize the mapping between task instances and tasks. The 3D model viewer provides flexibility allowing the user to zoom in to the particular position she is investigating. In future we will also allow the user to visualize the metamodel using the same 3D model viewer, in order to check the typing morphism from the model to its metamodel.
4. Related Work

While formal verification is valuable for the evaluation of safety critical systems, model checking tools used for the specification of the models and their properties are not always designed for use by non-experts in software verification. Here we discuss some other efforts which address modelling, verification and usability issues.

Pérez et al. [12] have proposed a framework to enable authoring and verification of clinical guidelines. They have used MDE techniques to automatically process manually created guideline specifications and temporal-logic statements. The MDE-based tool chain semi-automatically processes the guidelines, generating the input model of a model checker from the text, and input from domain experts. The approach uses Dwyer patterns [13] to specify commonly occurring types of properties. Their approach for the property specification patterns differs from ours at the representation level; they have extracted requirements from natural language specification where we have provided a tool to define requirements specifications using paths with propositions. Our property editor tool can be enriched with more predefined pattern templates described by Pérez et al. In future we will use a CTL model checker in order to incorporate more expressive patterns such as the Existence, Possible existence patterns, etc.

In [14] the authors propose an approach to the verification of clinical guidelines, which is based on the integration of a computerized guidelines management system with a model-checker. Advanced Artificial Intelligence techniques are used to enhance verification of the guidelines. The approach is first presented as a general methodology and then instantiated by loosely coupling the guidelines management system GLARE [15] and the model checker SPIN [16]. Although the authors argue that they can rely on GLARE in order to present the output provided by SPIN in a format that is easily readable to clinicians, the development of a user-friendly, graphical interface for the definition of LTL-properties is left as a future work. A similar approach was presented by Rabbi et al. [17] to model compensable workflows using the Compensable Workflow Modelling Language (CWML) and its verification by an automated translator to the DiVinE model checker; but neither the workflow model nor the transformation were MDE based.

In [18] compliance checking of Business Process Models based on model checking technology and visualisation of compliance violation is discussed. BPMN-Q queries are used to express execution ordering compliance rules. For each query a set of anti-pattern queries is automatically derived and checked against the process models. When a violation (an anti-pattern) finds a match, the violating part of the process is shown to the user. Similar to our approach, the usage of patterns and anti-patterns and visualization of
the violations will enhance understanding the verification results. However, requiring the user to define properties and execution ordering rules in BPMN-Q may hamper domain experts who use this approach.

Alloy [19] is a structural modelling language based on first-order logic, for expressing complex structural constraints and behaviour. The Alloy Analyzer is a constraint solver translating Alloy specifications written in relational logic to a boolean satisfiability problem which is automatically evaluated by a SAT solver. For a given specification $F$, the Alloy Analyzer attempts to find an instance which satisfies $F$ if the translated formula is satisfiable. Otherwise, it will find counter-examples within a limited scope which violates the constraints of the system. The counter-examples are displayed graphically, and their appearance can be customized for the domain at hand. In contrast to Alloy, our modelling language and analyser does not require any knowledge of constraints and formal model checking from the users of the tool.

In [20] a method to improve the reliability and minimize the risk of failure of business process management systems from a compliance perspective is presented. Business process models expressed in the Business Process Execution Language (BPEL) are transformed into pi-calculus and then into finite state machines. Compliance rules captured in the graphical Business Property Specification Language (BPSL) are translated into linear temporal logic. Thus, process models can be verified against these compliance rules by means of model checking technology. As with our approach, a counter-example tracer is used to visualise the results of the model checking process. However, this approach requires some knowledge of expert languages such as BPEL and BPSL.

The authors in [21] present concepts to visualize violation of soundness of Petri nets and workflow nets in a user-friendly way. The proposed idea handles the difficulties encountered when a workflow designer tries to correct workflow models according to the output of soundness tests. As in our approach, the authors argue that these difficulties are mostly due to the fact that diagnostic messages are not directly linked to the graphical model and intuition of the error source is missing. Unlike our approach, the approach is restricted to the visualisation of violation of only five classes of soundness properties [10].

In [2] the development of a collection of tools around the SMV model checker is discussed. A state chart model of the system is specified and translated into an SMV model. Similar to our approach, it specifies the properties to be analysed using either Dwyer’s patterns [13] or a predefined list of templates for useful properties. The system is checked against the properties and the output is presented in a tabular view. These tools are intended to make model checking more accessible to software engineers and in particular to those concerned with the human interface issues in complex safety critical systems; in our approach, we focus on making the tool more accessible for domain experts with no software engineering background.

5. Conclusion and Future Work

In this paper, we have proposed a user-friendly approach to the definition and verification of healthcare workflow models. We build on our MDE-based workflow modelling language for the definition of diagrammatic workflow models and support verification of workflow models by a user-friendly technique for the definition of LTL-properties. The basic steps of our approach can be summarised as follows: healthcare workflows are defined using a diagrammatic editor; desired and undesired scenarios are defined as patterns using a diagrammatic LTL-property definition editor; the workflow models are transformed to DVE code, the language of the DiVinE model checker; the DVE code is then checked against the LTL-properties using the DiVinE model checker; if the model does not satisfy a property, we get a counter-example.

One of the main contributions of the paper is that we visualise the counter-examples in a syntax similar to the one used to define the workflow models. Our goal is to enable healthcare personnel to define, or at least fully understand, healthcare workflow models easily, and also enable them to define desired and undesired scenarios (or properties) and verify the models against these properties, and understand the results/outcomes of the verifications which they perform. The proof of concept prototype tool has been customised and is intended for use by clinicians; however, we believe the ideas are applicable to a wider field.

In future, we will extend our workflow modelling language to support data-awareness, i.e., support for declaration and value assignment of global and local variables. We will also implement a less restricted loop construct. That is, we will allow loops with incoming/outgoing flows from/to other tasks which are
outside the loop. In addition, we plan to implement a richer property definition module, supplemented with more predefined LTL-properties. Currently, our property definition tool does not directly support existential quantifiers since we are using an LTL model checker (though running the negation of a property and getting counter example gives a witness – i.e., a run where the property holds). We plan to adopt a CTL model checker in future in order to facilitate the definition of more expressive properties. Also, based on feedback from clinician we would like to improve the tool to make it even more user-friendly, and to enable users to define their own routing predicates.

Acknowledgements. This work is partially supported by The Natural Sciences and Engineering Research Council of Canada (NSERC), Atlantic Canada Opportunities Agency (ACOA), and The Research Council of Norway project 194521 (FORMGRID). The authors wish to thank Xiaoliang Wang for his contributions to the programming.

References


