Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 37 (2014) 481 – 488

Procedia Computer Science

1st International Workshop on Metamodelling for Healthcare Systems (MMHS–2014)

Model Checking Healthcare Workflows using Alloy

Xiaoliang Wang[a], Adrian Rutle[a,*]

[a]*Bergen University College, Bergen, Norway*

Abstract

Workflows are used to organize business processes, and workflow management tools are used to guide users in which order these processes should be performed. These tools increase organizational efficiency and enable users to focus on the tasks and activities rather than complex processes. Workflow models represent real life workflows and consist mainly of a graph-based structure where nodes represent tasks and arrows represent the flows between these tasks. From workflow models, one can use model transformations to generate workflow software. The correctness of the software is dependent on the correctness of the models, hence verification of the models against certain properties like termination, liveness and absence of deadlock are crucial in safety critical domains like healthcare. In previous works we presented a formal diagrammatic framework for workflow modelling and verification which uses principles from model-driven engineering. The framework uses a metamodelling approach for the specification of workflow models, and a transformation module which creates DiVinE code used for verification of model properties. In this paper, in order to improve the scalability and efficiency of the verification, we introduce a new encoding of the workflow models using the Alloy specification language, and we present a bounded verification approach for workflow models based on relational logic. We automatically translate the workflow metamodel into a model transformation specification in Alloy. Properties of the workflow can then be verified against the specification; especially, we can verify properties about loops. We use a running example to explain the metamodelling approach and the encoding to Alloy.

Peer-review under responsibility of the Program Chairs of EUSPN-2014 and ICTH 2014.

Keywords: Workflow modelling, Efficient verification, Alloy, Model checking, Model-driven engineering.

1. Introduction

Healthcare is the domain which cost states and local governments a considerable portion of their budgets. Furthermore, mistakes in almost any aspect of a healthcare-related system may cause severe damages. This has lead to an increasing pressure on making processes and procedures in healthcare safer and more effective. Clinical guidelines, dictating how processes should be organized, have been provided by health authorities to guide and unify healthcare processes across institutions. These guidelines are in constant changes due to updates in regulations and advances in treatment methods and medications. Unfortunately, the guidelines are traditionally written in natural languages, which can run to hundreds of pages, incorporating heavily annotated diagrams which use non-standard and confusing notations [1].

* Corresponding author. Tel.: +47-5558-7791 ; fax: +47-5558-7789.
E-mail addresses: xwa@hib.no (Xiaoliang Wang)., aru@hib.no (Adrian Rutle).

Peer-review under responsibility of the Program Chairs of EUSPN-2014 and ICTH 2014.
doi:10.1016/j.procs.2014.08.072

Workflow models may be used to formally structure clinical guidelines. A workflow model consists mainly of a graph-based structure where nodes represent tasks and arrows represent the flows between these tasks. In earlier work[2,3,4] we proposed a diagrammatic framework (called DERF) for the specification of workflow models using model-driven engineering (MDE)[5,6] techniques. The diagrammatic models are easily understood by domain-experts, and the metamodelling approach allows models to be easily customized to deal with new treatment procedures and other changes in clinical guidelines.

From workflow models, one can use model transformations to generate workflow software. Workflow software are used to guide users in which order these processes should be performed, and to resolve dependencies between tasks. These tools improve organizational efficiency and enable users to focus on the tasks and activities rather than complex processes. The correctness of the software is dependent on the correctness of the models, hence verification of the models against certain properties like termination, liveness and absence of deadlock are crucial in safety critical domains like healthcare. In[7] we proposed a verification approach for models specified in DERF, in which the workflow models were transformed to DVE, the language of the DiVinE model checker. The approach also incorporated a user-friendly editor for specification of model properties, as well as a module for visualization of counter-examples in case some properties did not hold. In this paper, we extend upon our earlier work, and introduce a new, efficient encoding of the workflow models using the Alloy specification language. Furthermore, we present a bounded verification approach for workflow models based on relational logic. We automatically translate the workflow metamodel into a model transformation specification in Alloy. Properties of the workflow can then be verified against the specification; especially, we can verify properties about loops. In case a property does not hold, a counter-example is generated automatically by the Alloy and visualized as a graph. We use a running example (adopted from[7]) to explain the metamodelling approach and the encoding to Alloy.

In Section 2 we review our workflow modelling language. In Section 4 we discuss correctness of workflow models, explain our encoding to the Alloy specification language, and visualize counter-examples. Sections 5 and 6 present some related and future work and conclude the paper.

2. Metamodeling for Healthcare Workflows

Workflow models may be used to document and analyse complex work processes in clinical guidelines and to ensure their formal correctness. In previous work, we presented a diagrammatic modelling framework used for workflow modelling[2,3,4,7]. A design goal of the framework has been to make the modelling tools intuitive enough to be used by healthcare practitioners and formal enough to be used to specify and verify interesting properties of healthcare workflows. Here, we only present the most important details of the framework, the details can be found in the references above. This short presentation of the modelling language and the running example are adopted from[7].

The workflows are represented as graph-based structures describing in which order specific tasks should be executed. Each task is represented by a node. If there is an arrow $T_1 \xrightarrow{e} T_2$ from a task T_1 to a 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 can be extended in the standard way to arbitrary triples, quadruples, etc. The most used splits (e.g. `[and_split]`, `[or_split]` or `[xor_split]`) and merges (e.g. `[and_merge]`, `[xor_merge]` or `[or_merge]`) are formulated as predicates in our framework. 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]`.

Fig. 1 shows a sample of a workflow from the healthcare domain. The workflow illustrates a simplified scenario for cancer treatment. After an initial examination, the patient will have an MRI examination *and* a blood test. According to the results of the two tests, the physician will decide which procedure the patient should follow (*either* Procedure A *or* Procedure B). After finishing the chosen procedure, the result shall be evaluated to determine whether the patient should use drug treatment or not. If drug treatment is chosen, then when the drugs are finished a blood test is taken and the result is evaluated to determine whether the patient should be given further drug treatment or not. Hence if the drug treatment is repeated, the blood test and the evaluation will be repeated as well; i.e., the workflow will be in a loop. The workflow ends when the evaluation shows that the drug treatment should terminate.

The syntax and semantics of the workflow modelling language is given in[2,3,4,7]; here we only recall some of the 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

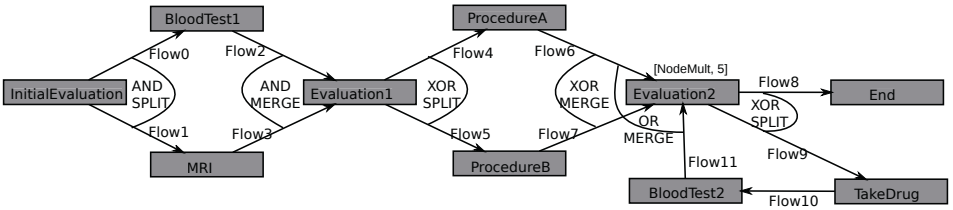


Figure 1: Sample workflow model. Adopted from[7].

(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. In DERF, we have three modelling levels: M2, M1 and M0. The metamodel of our workflow modelling language (which is at level M2) consists of a node Task and an arrow Flow. This means that we can define a set of tasks together with the flows between these tasks. The signature Σ_2 of the workflow modelling language consists of a set of routing predicates such as `[and_split]`, `[and_merge]`, `[xor_merge]`, etc. Tasks which are involved in a cycle in the workflow are marked with a predicate `[NodeMult,n]` where n specifies how many instances that task can have at most. We call these tasks "loop tasks", and we call flows within a loop for "loop flows".

From the metamodel at level M2 and the signature Σ_2 with routing predicates, we can create a modelling language for the definition of "workflow models". These workflow models, 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 Σ_1 (see Fig. 2) We refer to <E>, <R> and <F> as "task states". Note that in an earlier version of the language[2,3] we had 4 states, <D>, <E>, <R> and <F>, thereof the name DERF. 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 model transformation rules (see Tables 1 and 2). For example rule t_1 takes an instance of a task from <E> to <R> and rule t_2 takes an instance of a task from <R> to <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. The state space which can be generated by the transformation rules comprises the dynamic semantics of the workflow.

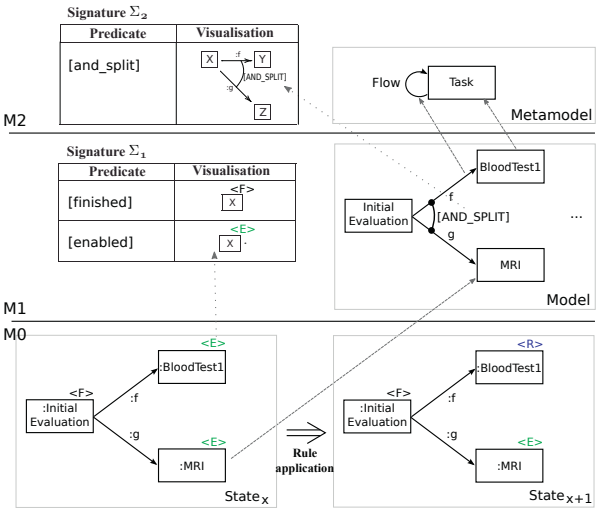


Figure 2: Workflow modelling hierarchy: dashed arrows indicate types of some model elements, dotted arrows indicate relations between signatures and models. Adopted from[7].

3. Encoding of workflow model

In this section, we will cover how to encode a workflow model and its corresponding transition system as an Alloy specification. The specification represents a model transformation system which simulates the dynamic semantics (each task can change from a state to another). However, the state information is not represented in the generated specification. The encoding procedure is adapted based on our encoding of model transformation systems detailed in[10]. It is implemented as a code generation module in DPF and can derive the Alloy specification automatically from a workflow model and the coupled transformation rules. Before presenting the encoding procedure, we give a brief introduction to Alloy.

Table 1: The coupled transformation rules t_1 and t_2 of our transition system

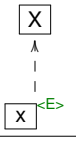
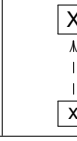
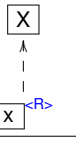
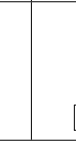
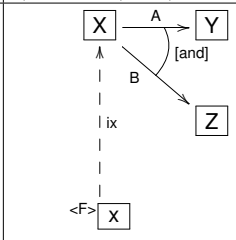
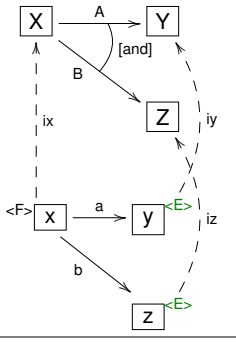
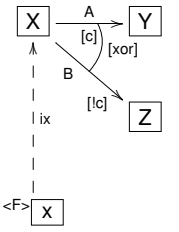
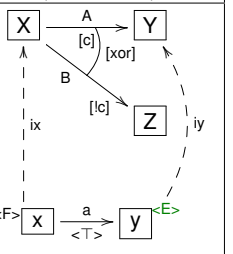
t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$	$(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$	t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$	$(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$
t_1				t_2			

Table 2: Some coupled transformation rules for the transition system, adopted from [4]

t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) = (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$	t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) = (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$
t_3			t_4		

Alloy[11] is a structural modelling language, based on first-order logic, for expressing complex structure and constraints. 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 or find a counterexamples which violates F by running run or $check$ command within a use-defined scope. The instance or counter-example is displayed graphically, and their appearance can be customized for the domain at hand.

3.1. Encoding of the metamodel at M2 level

Recall that each model in DPF (and also in DERF) consists of an underlying graph and a set of constraints. Given a workflow model, for the underlying graph, each task $t{:}Task$ is encoded as a task signature S_t; each flow $f{:}Flow$ is encoded as a flow signature S_f with two fields src and trg denoting the source task and the target task of the flow. The encoding procedure handles the loop tasks specially. In order to count how many times the task is performed, a field $count$ is added to the loop task's signature. Thus the workflow model can be encoded as a graph signature S_G containing two fields: the field $nodes$ denoting the tasks; the field $arrows$ denoting the flows. Since the structure is a graph, it should satisfy that if a flow is contained by a graph g, its source and target tasks should also be contained by g. The structure encoding is shown in the following listing: (assuming the structure contains m tasks and n flows.)

```
1 sig S_{t_i}{count:one Int//The field is optional depending if the task is a loop task or
      within a loop.
2 }//For each task t_i, i ∈ {1..m}
3 sig S_{f_j}{src:one S^s_{f_j}, trg:one S^t_{f_j}}{//For each flow f_j, j ∈ {1..n}, S^s_{f_j}/S^t_{f_j} is the flow's
    source/target task
4 sig S_G{nodes:set S_{t_1}+...+S_{t_m},edges:set S_{f_1}+...+S_{f_n}}
5 fact{all g:S_G|all e:g.edges|(e.src in g.nodes and e.trg in g.nodes)}
```

Besides the structural information, the workflow model contains also constraints restricting the set of valid instances. The constraints are of two types:

General Constraints These constraints are implicitly contained in each workflow model and must be satisfied by all workflow states. In DPF, we specify these constraints using universal constraints[8].

1. Each task instance may enable at most one instance of the same subsequent task. This is forced by a multiplicity constraint `mult[0..1]` on each flow in workflow models. Similarly, two instances of the same task cannot enable the same instance of a subsequent task. This is forced by injective constraint `[inj]` on each flow.
2. A task instance cannot be enabled before its preceding task is finished. To specify this constraint, when a task has only one incoming flow, the flow will be constrained with surjective constraint `[surj]`. However, if the task has multiple incoming flows and the model designer has not put any routing constraint on these, the constraint `[or_merge]` is put on the flows.
3. If a task has incoming flows mixing loop flows and ordinary flows, two separate `[or_merge]` (or `[sur]` if the sets contain only one) are put on each of these two sets.

Specific Constraints These constraints are specified in a workflow model explicitly by designers. These constraints are formulated using predicates from Σ_2. Since there is a limited number of predicates for the workflow modelling language, these predicates are hard-coded in the implementation and used to formulate different constraints in the models. For example, the `[xor_split,c]` constraints in Fig. 1 are encoded as:

```
pred fact_E1_xor_split[g:Graph]{//For Evaluation1
    all n:NE1&g.nodes|not ((some e:AE1_PB&g.arrows|e.src=n) and (some e:AE1_PA&g.
        arrows|e.src=n))
}
pred fact_E2_xor_split[g:Graph]{//For Evaluation2
    all n:NE1&g.nodes|not ((some e:AE1_PB&g.arrows|e.src=n) and (some e:AE1_PA&g.
        arrows|e.src=n))
}
```

3.2. Encoding of model transformation

In DERF, we use coupled transformation rules to define the dynamic semantics of workflow models. We adopt a variant of the encoding procedure for transformation rules detailed in[10]. First, we derive the graph transformation rules by finding the matching of each coupled transformation rule. For example, for the rule t_4 in Table 2 defining the semantics of `[xor_split,c]`, two matches are found: one on Evaluation1 and one on Evaluation2 (See rules $E1^1_{xs}, E1^2_{xs}, E2^1_{xs}, E2^2_{xs}$ in Table 3). Note that this step of deriving the graph transformation rules is performed implicitly in the encoding procedure. Then each derived rule r is encoded as a predicate $pred\ apply\_r[tran{:}Trans]$ as in[10] stating that a transformation applies the rule. The signature $Trans$, as in[10], encodes the direct model transformations which contains 7 fields: the rule applied $rule$, the source workflow $source$, the target workflow $target$, and, the deleted and added elements during the transformation $dnodes$, $anodes$, $darrows$, $aarrows$. Assuming there are nr derived rules, the following *fact* asserts that every transformation should apply exactly one of the derived rules.

```
fact {all t:Trans | apply_r1[t] or ... or apply_rnr[r]}
```

Since in the workflow modelling language loops are represented as tasks with predicate `[MultNode,n]`, the loop tasks can be repeated a finite number n of times. That is, the loop tasks may have up to n instances. Therefore, when deriving the graph transformation rules for this case, several points should be considered:

- For the incoming flows of a loop task which are not loop flows, the rule creates a new instance of the loop task with $count = 0$ (see rules $E2^1_{xm}$ and $E2^2_{xm}$ in Table 3).
- For the flow loops which are not coming into a loop task, the rule creates a new instance of the flow's target task with $count$ equals to the flow's source task. In addition, for the flow coming out of a loop task, a precondition should check if its count is less than the upper limit n in `[MultNode,n]` (see rule $E2^2_{xs}$ in Table 3).
- For the loop flows coming into a loop task, the rule shall create a new instance of the loop task with $count = count' + 1$, where $count'$ is the count of the flow's source task (see rule $Flow11$ in Table 3).

Table 3: Derived graph transformation rules for t_4 in Table 2

Rule	L	K	R
$E1^1_{xs}$	:E1	:E1	:E1 → :PA
$E1^2_{xs}$	:E1	:E1	:E1 → :PB
$E2^1_{xs}$	:E2	:E2	:E2 → :End
$E2^2_{xs}$	c<5 :E2	c<5 :E2	c<5 :E2 → :TD
$E2^1_{xm}$	:PA	:PA	:PA → :E2 c=0
$E2^2_{xm}$	:PB	:PB	:PB → :E2 c=0
Flow11	c :BT2	c :BT2	c :BT2 → :E2 c+1
Flow10	c :TD	c :TD	c :TD → :BT2 c

4. Verification of Healthcare Workflow

After a workflow is encoded as an Alloy specification, the Alloy Analyzer can be used to verify its properties. In this work, we want to verify whether the workflow model satisfies *generic properties* such as: 1) absence of deadlocks, and, 2) termination (when loops are present). The Alloy Analyzer performs a bounded check and can prove whether the workflow system is without error w.r.t. the properties within a user-defined scope. Hence, the approach can find bugs in a workflow model efficiently. In addition, the Alloy Analyzer can visualize the counterexamples if they exist.

Before verifying these generic properties, we firstly verify a pre-property that the encoded Alloy specification correctly stimulates the dynamic semantics of the workflow model. It means that every instance of the *Trans* encodes a transition between states in the state space of the workflow model. Note that the pre-property implies that each instance of the S_G encodes a valid state of the workflow model. If this is verified correct, then we can examine the derived Alloy specification to verify other properties of the workflow model; in addition, it means that each workflow instance contains path information; i.e., there exists a sequence of transformations applied on the start state to get such an instance. Otherwise, it means that the modeling of the workflow is not correct and we need to revises the workflow model or the rules to fix the problem.

To verify the pre-property, we check the *Direct Condition*[10] to show that each transformation from a valid source state can produce a valid target state. In addition, a similar condition should also be verified: if the target of a transformation is a valid state that the source is also a valid state. Similar to the verification method in[10], these two properties are verified by running the commands in the following listing. The scope we use is *for* 10 *but exactly* 1 *Trans*, *exactly* 2 *Graph*. It means that in each workflow instance, at most 10 instances of each task (such as Evaluation1 and Evaluation2) are present.

```
check{all trans:Trans|valid[trans.source] and not valid[trans.target]} for 10 but
    exactly 1 Trans, exactly 2 Graph
check{all trans:Trans|not valid[trans.source] and valid[trans.target] and not
    isStart[trans.target]} for 10 but exactly 1 Trans, exactly 2 Graph
```

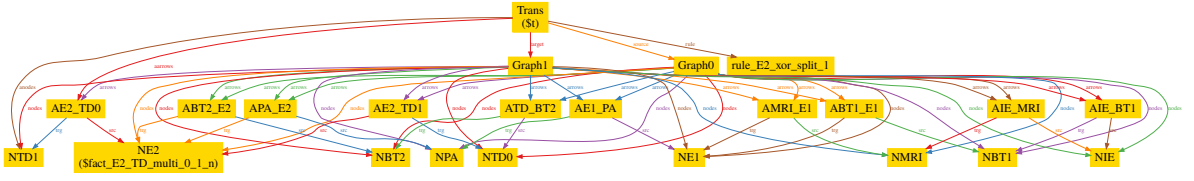


Figure 3: Counterexample of `xor_split`

The verification result shows several counterexamples; e.g. the `[xor_split,c]` constraint is violated. One violation is shown in Fig. 3. To correct this problem the rule for `[xor_split,c]` should use the two split branches as NAC to avoid reapplying the rules multiple times (see Table 3). The errors and counterexamples disappear after

that some rules are revised. This means that the encoded Alloy specification correctly simulates the dynamics of the workflow model.

Now we can prove the properties like absence of deadlock or termination for loops. To verify the absence of deadlock property, we try to find a transformation where the source state is valid $valid[trans.source]$, the target state is not in finished state $not\ finished[trans.target]$ (which means the workflow terminates,) and no rule can be applied on the target model $not\ rules\_applicable[trans.target]$. If such transformation is found, it means there is deadlock in the workflow model. The Alloy Analyzer finds an instance by the command in the following listing.

```
run{all trans:Trans|valid[trans.source] and not finished[trans.target] and not
    rules_applicable[trans.target]} for 10 but exactly 1 Trans, exactly 2 Graph
```

We can verify that a workflow will terminate although it contains a loop. It means each time a workflow enters a loop, it will terminate in the future. We can use the Alloy Analyzer to find counterexamples. That is, a workflow has entered a loop but have not finished or have further applicable rule. Actually, this is a special case of deadlock verification. The result shows there is no deadlock or loop without termination for the workflow model.

```
run{all trans:Trans|has_enter_loop[trans.source] and valid[trans.source] and not
    finished[trans.target] and not rules_applicable[trans.target]} for 10 but
    exactly 1 Trans, exactly 2 Graph
```

5. Related Work

We shortly present some efforts using model checking for verification of safety critical systems. Pérez et al.[12] use MDE-based tool chain semi-automatically to process manually created clinical guideline specifications and generate the input model of a model checker from the specifications. The approach uses Dwyer patterns[13] to specify commonly occurring types of properties. 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]. 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. In[18] a method to 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.

Most of these works use model checking to verify the workflow system while we use Alloy, based on relation logic and a satisfiability solver. These works are complete since the model checker work on the whole state space. However, our approach is bounded and incomplete, i.e., the properties verified is only valid in some scope. But our approach can find bugs in the system more efficiently. In addition, the above mentioned works have their own patterns and languages to specify the properties and verify different kinds of properties, while in our work, we only verify those mentioned properties if they are expressed in first-order logic. Furthermore, we can also derive the model checker input file (semi-)automatically.

6. Conclusion and Future Work

In this paper, we apply a bounded verification approach based on Alloy to the verification of healthcare workflow models. We build on our MDE-based workflow modelling language for the definition of diagrammatic workflow models. In order to verify a workflow, the dynamic semantic of the workflow is simulated as a model transformation system, encoded as a specification in Alloy. Then the Alloy Analyzer is used to verify general properties of the workflow by finding counterexamples. If such counterexamples are found, they are visualized by the Alloy Analyzer showing how a property is violated.

One of the main contributions of the paper is that we use a new technique to verify workflow models. Comparing with other approaches with model checking techniques, the approach is bounded and incomplete. But the approach enable the designer quickly find the bugs in the models and correct them with the feedback from the verification result. In[10], the verification approach based on Alloy encounter a scalability problem when the relations in metamodel or transformations rules are too complex. But as we can see from the workflow metamodel and the derived transformation rules, this may not be a problem; because the arity of the relations in the coupled model transformations are at most 2.

In this work, we only applied the approach to one workflow model. In the future, larger models will be used to study the performance of the approach. Right now, limited properties are verified with the approach. More study should be continued to see whether other properties can be verified. In[7] we used a user-friendly editor for the specification of properties. We plan on translating properties defined in this editor so that they can be verified against the Alloy specifications using Alloy Analyzer. Furthermore, we abstract out the state information in the encoding procedure. Actually, some flows, like `TakeDrug` to `Evaluation2`, can be also omitted. We will check if any systematical approach can make the encoding result simpler.

References

1. D. Méry, N. Singh, Medical protocol diagnosis using formal methods, in: Z. Liu, A. Wassyng (Eds.), Foundations of Health Informatics Engineering and Systems, Vol. 7151 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 1–20. doi:10.1007/978-3-642-32355-3_1.
2. H. Wang, A. Rutle, W. MacCaull, A Formal Diagrammatic Approach to Timed Workflow Modelling, in: Proceedings of TASE 2012: 6^{th} International Conference on Theoretical Aspects of Software Engineering, Vol. 0, IEEE Computer Society, 2012, pp. 167–174.
3. A. Rutle, H. Wang, W. MacCaull, A Formal Diagrammatic Approach to Compensable Workflow Modelling, in: Z. Liu, A. Wassyng (Eds.), Foundations of Health Informatics Engineering and Systems, Vol. 7789 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 194–212.
4. A. Rutle, W. MacCaull, H. Wang, Y. Lamo, A Metamodelling Approach to Behavioural Modelling, in: Proceedings of BM-FA 2012: 4^{th} Workshop on Behavioural Modelling: Foundations and Applications, ACM, 2012, pp. 5:1–5:10.
5. B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25. doi:10.1109/MS.2003.1231146.
6. T. Stahl, M. Völter, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2006.
7. A. Rutle, F. Rabbi, W. MacCaull, Y. Lamo, A user-friendly tool for model checking healthcare workflows, Procedia Computer Science 21 (0) (2013) 317 – 326, the 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2013) and the 3rd International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH). **Best paper award**. doi:/10.1016/j.procs.2013.09.042.
8. A. Rutle, Diagram Predicate Framework: A Formal Approach to MDE, Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010).
9. Y. Lamo, X. Wang, F. Mantz, W. MacCaull, A. Rutle, DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment, in: R. Lee (Ed.), Computer and Information Science 2012, Vol. 429 of Studies in Computational Intelligence, Springer Berlin Heidelberg, 2012, pp. 37–52. doi:10.1007/978-3-642-30454-5_3.
10. X. Wang, Y. Lamo, F. Büttner, Verification of graph-based model transformation using alloy, in: In: Proc. of GTVMT, 2014.
11. Alloy, Project Web Site, http://alloy.mit.edu/community/.
12. B. Pérez, I. Porres, Authoring and verification of clinical guidelines: A model driven approach, Journal of Biomedical Informatics 43 (4) (2010) 520–536.
13. M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st international conference on Software engineering, ICSE '99, ACM, New York, NY, USA, 1999, pp. 411–420.
14. A. Bottrighi, L. Giordano, G. Molino, S. Montani, P. Terenziani, M. Torchio, Adopting model checking techniques for clinical guidelines verification, Artificial Intelligence in Medicine 48 (1) (2010) 1 – 19. doi:10.1016/j.artmed.2009.09.003.
15. L. Anselma, A. Bottrighi, G. Molino, S. Montani, P. Terenziani, M. Torchio, Supporting knowledge-based decision making in the medical context: The glare approach, IJKBO 1 (1) (2011) 42–60.
16. SPIN, Project Web Site, http://spinroot.com/.
17. F. Rabbi, H. Wang, W. MacCaull, Compensable Workflow Nets, in: Proceedings of ICFEM 2010: 12^{th} International Conference on Formal Engineering Methods, Vol. 6447 of Lecture Notes in Computer Science, Springer, 2010, pp. 122–137. doi:/10.1007/978-3-642-16901-4_10.
18. Y. Liu, S. Müller, K. Xu, A static compliance-checking framework for business process models, IBM Syst. J. 46 (2) (2007) 335–361.