A Property Specification Language for Runtime Verification of Executable Models

Fernando Macias, Adrian Rutle, and Volker Stolz

Bergen University College
first.last@hib.no

The definition of workflows is a complex task, comprising aspects such as time constraints, failure detection and recovery. Executable modelling is a promising concept for the simplification of workflow modelling, verification and execution. Hence, the verification of executable models, especially using runtime verification and monitoring, is required. We present a metamodelling approach to the combined modelling of workflows and the temporal properties used to define requirements and constraints on them. Here, a model at a certain level describes a modelling language which can be used to specify models in the level below. First, we give an overview of a generic workflow modelling framework for executable models. Then, we highlight the design and implementation of the property specification language. Both the workflows and their temporal properties are specified as graph-based structures. For the temporal properties, we draw on the well-known Linear Time Logic (LTL), which has already been used successfully in checking whether an (execution) path satisfies a given property [3]. A key point of the proposed language is its direct applicability on the running instances of the workflows, instead of the execution logs as it is usually done. Because of this, the atomic propositions of our language are graphs which are matched against the actual running instances while monitoring, and hence translating *match/no match* to *true/false* respectively. In our approach we exploit facilities from deep metamodelling [4] to establish links between the running instances of the workflows and the atomics propositions, using the *typing* relationship. Deep metamodelling enables us to define types for other levels in the hierarchy than the one directly below. In addition, we will borrow the concept of *linguistic extension* and *double typing* of model elements [4].

Model execution and verification

This work is framed in a bigger proposal for the creation of a metamodelling framework which allows for the definition of executable modelling languages. To achieve this goal, we have outlined a deep metamodelling hierarchy where the two topmost levels are fixed (see Fig. 1). In them, the common elements for any executable language are defined. With these two levels as a starting point, the user can define her own modelling language using as many levels as required. The bottommost level will contain the running instances that the execution runtime can interpret and modify in every time step through model transformations. Note that the modelling levels M_2 through M_4 are displayed only to be used as examples and are inspired by the workflow modelling language defined in [6, 5], while M_0 and M_1 are fixed. The syntax of the language defined at level M_2 is out of the scope of this paper; the interested reader may consult the references above. It is worth pointing out, however, the meaning of the dashed arrows: they represent the *typing* relationship, which indicates the type of the element according to the metamodel in the level above. Formally, this relationship is defined as a graph homomorphism from lower level models to upper level models. We can also indicate this relation with the colon notation ":", e.g. Task:Executable. Note that in Fig. 1 we have used the concrete syntax for the models M_2 through M_4, meaning that, e.g. a Flow instance, which should be a node, is actually represented as an arrow.

The same applies to the [and] and [xor] constraints in M_3. For the sake of clarity, not all the typing relationships are displayed. Next, we focus on the property definition language which is defined as a linguistic extension (i.e. a parallel metamodel) on the left-hand side. This technique consists of a separate model aside the vertical hierarchy, whose elements can be instantiated at any level of the hierarchy. In our work, this technique allows to apply such properties in any level using double typing. Further explanations of this can be found in the next section.

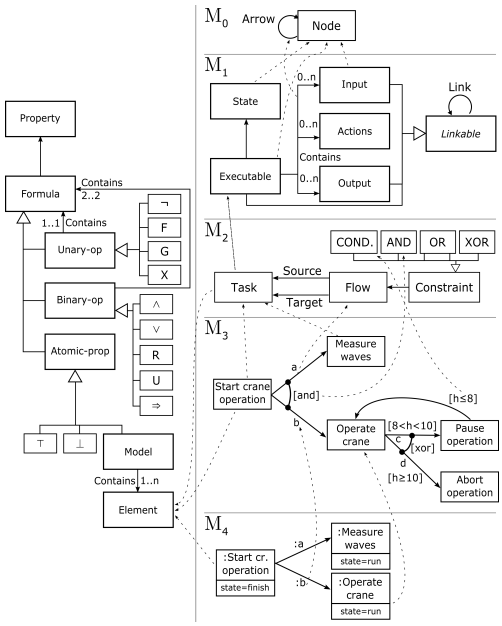

Figure 1: Global modelling hierarchy

As a consequence of the choice of models for the representation of both the workflows and the temporal properties, we propose two disjoint sets of model transformations. The first one defines the semantics for the evolution of the running workflow instances in every time step (see [5, 2]). The second one defines the expansion of the temporal properties in the same stepwise manner, and determines how the properties are checked against the running workflow instances. This second set of model transformations is briefly introduced at the end of the next section. We discuss the design space of this construction and show its applicability to runtime verification and monitoring, where the properties are checked against the running system, as opposed to model checking, where the whole state space of the model is explored. In the case of the violation of a temporal property some actions could be taken, e.g. generating a warning.

Property Specification Language

The language, as inspired by LTL, contains the the main temporal operators: **F** (eventually), **G** (always), **U** (Until), **R** (release) and **X** (next). It also contains the well-known boolean operators $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ and the terminal symbols $\top$ and $\bot$.

In order to create a consistent syntax for the language, all these operators inherit from one of the abstract classes UnaryOperator, BinaryOperator or AtomicProposition (see Fig. 1). All these, in turn, inherit from Formula, and can contain instances of Formula elements at the same time. This allows, in a grammar-like style, for the nesting of operators with the correct cardinality. Hence, a Property contains a single operator, which then contains the remaining operators in a tree structure; i.e. the models represent abstract syntax trees.

The two remaining elements in the language are Model and Element. In these elements lies the expressive power of our language: any element in any model in any level can be typed by Element (hence the redundancy) in addition to a type in their corresponding metamodel. One of the examples of this double typing shown in Fig. 1 is Start crane operation, which is both typed by Task and Element. This allows us to specify properties *on any* modelling level, or even *across* modelling levels, i.e. the same property can use Model instances which contain Element instances at different levels of the hierarchy. We call these *cross-level* properties.

In the sample hierarchy (levels M_2-M_4) in Fig. 1, properties specified using elements from M_3 as well as properties connecting elements from M_3 and M_4 can be understood as requirements specifications. In general, this consideration can be applied to the second-to-last level, where

the workflow itself is designed, right above the running instance. To illustrate our proposal, we show a possible temporal property *liveness* for the preceding workflow (Fig. 1). Note that this property is deliberately convoluted for the sake of exemplification.

Fig. 2 shows the model version of the liveness property which reads as follows in natural language: "For every X:Task, if we find a :X with the state *enable*, we should eventually find a :X with the state *run* and eventually after that a :X with the state *finish*". The main highlights are (1) the expressiveness of the language, which allows for the definition of a strict requirement which includes the sequence of states that the element must go through; and more importantly, (2) the inclusion of a variable element, X:Task, of the level above the running instances, M_3.

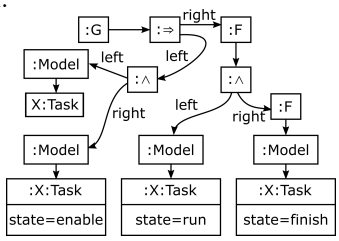

Figure 2: Liveness property

Finally, the semantics of the language is specified using model transformation rules. These rules check in a stepwise manner the specified properties against the running instance of the workflow. In order to do that, we use LTL expansion rules [ref], e.g. $G(f) = f \wedge X(G(f))$, for any formula f. This expansion is necessary to decompose the rules and extract atomic properties that can be checked in the current point in time in the workflow execution. Checking an atomic proposition in a workflow instance means finding a graph homomorphism (i.e. a match) from the underlying graph of the proposition to the underlying graph of the workflow instance, and hence translating *match/no match* to *true/false* respectively. We have already implemented a prototype with this semantics using EMF [1] and ATL [1].

Conclusions and future work

We have presented a modelling language for the specification of temporal properties. While the idea of applying LTL monitoring to workflows is not new, cf. [8], our focus here is on the integration into a generic modelling framework for the specification and execution of workflows. We have also showed its two main characteristics: (1) the focus on the specification of temporal properties that have to be checked by means of runtime verification, instead of model checking; and (2) the flexibility it offers for the specification of *cross-level* properties. In this context, we plan to address in the immediate future what we call *multi-instance* properties, i.e. the ones that are checked against more than one workflow instance running in parallel. For this, it is required to extend the language with a means of quantification over instances [7].

References

[1] Eclipse Modeling Framwork. *Web site*. `http://www.eclipse.org/modeling/emf`.

[2] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *IJSTTT*, 14(1):15–40, February 2012.

[3] M. Leucker et al. A brief account of runtime verification. *JLAP*, 78(5):293–303, 2009.

[4] A. Rossini, J. De Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014.

[5] A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *BM-FA in conjunction with ECMFA*, Kgs. Lyngby, Denmark, July 2012. ACM.

[6] A. Rutle, H. Wang, and W. MacCaull. A Formal Diagrammatic Approach to Compensable Workflow Modelling. In Z. Liu and A. Wassyng, editors, *FHIES*, volume 7789 of *LNCS*, pages 194–212. Springer, 2013.

[7] V. Stolz. Temporal assertions with parametrized propositions. *JLC*, 20(3):743–757, 2010.

[8] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. OTM'05, pages 130–147. Springer, 2005.