Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

TRANSLATING ENGINEERING WORKFLOW MODELS TO DEVS FOR PERFORMANCE EVALUATION

István Dávid[1,2] Yentl Van Tendeloo[1,2] Hans Vangheluwe[1,2,3]

[1]University of Antwerp, Belgium
[2]Flanders Make vzw, Belgium
[3]McGill University, Montréal, Canada
{istvan.david, yentl.vantendeloo, hans.vangheluwe}@uantwerp.be

ABSTRACT

Engineering workflow models are frequently used to optimize an engineering endeavor for some well-defined performance metrics, such as time-to-market or monetary costs. Static workflow analysis is often insufficient due to the complex interleavings of different activities and the interplay with limited resources. Simulation-based techniques provide a feasible alternative to static analysis. In this paper, we provide an automated translation from engineering workflow models to DEVS models, useful for simulation and subsequent performance evaluation. Our translation supports the vast majority of the essential workflow control patterns, previously identified by van der Aalst et al. Thanks to the use of simulation, our approach is able to deal with stochastically varying activity execution times and workflow decisions, potentially evolving between subsequent iterations of the workflow. Our approach is implemented in the Modelverse, where the mapping to and simulation of DEVS models remains completely hidden from the process modeler.

1 INTRODUCTION

The increasing complexity of nowadays engineered systems necessitates the detailed modeling of the system before it is being realized. Multi-paradigm modeling (MPM) (Mosterman and Vangheluwe 2004) advocates modeling every aspect of the system, using the most appropriate formalisms, while also modeling the workflow of engineering the system. Workflow models, at the very least, can serve as documentation and specification of the engineering endeavor. In more advanced cases, they can be enacted (Dávid et al. 2017) and a significant part of the engineering work can be automated. Due to the enormous costs associated with the engineering of complex systems, it is desirable to optimize the workflow for a performance metric (e.g., transit time) before enacting it, thus increasing its efficiency.

Static workflow analysis is often insufficient due to the complex interleavings of different activities and the interplay with limited resources. Indeed, activities can execute concurrently only if some constraints are met, such as having sufficient resources available and that all activities it depends upon have finished. Some activities might even execute partially, such as a multi-instance activity, which might spawn three of its four instances while waiting for additional resource. Simulation-based techniques, therefore, provide a feasible alternative to static analysis.

In this paper, we provide an automated translation from engineering workflow models to a formalism appropriate for the simulation of performance characteristics of the workflow. To show that our approach is applicable to all possible workflow models, we base ourselves on the essential workflow patterns, previously identified by van der Aalst et al. (2003). Each of these patterns introduces essential concepts of a workflow model, thereby providing us with a list of elements that should be supported. While our approach is complete, only the most common constructs are presented here due to space restrictions.

Our mapping is furthermore specialized in two dimensions: (i) the stochastic nature of workflow execution; and (ii) the constraints imposed by the resource demands of the workflow.

Structure

The remainder of this paper is structured as follows. In Section 2, we present terminology and an overview of the approach. In Section 3, we present the steps required to transform workflow models to DEVS models. In Section 4, we discuss the stochastic aspect of the workflow models and show how to capture this during simulation. We provide a general mechanism for modeling resource constraints in Section 5, in order to make the workflow models more realistic. In Section 6, we evaluate the performance of the workflow, given some limited resources and distributions for the activities. In Section 7, we discuss the related work. Finally, Section 8 concludes our paper.

2 OVERVIEW

In this section, we give a brief overview of the aspects of our work.

Performance of workflows To evaluate the performance of a workflow model we assume a metric space (W,d) with W being the set of workflows and d being a metric on W, i.e. a function $d : W \times W \rightarrow \mathbb{R}$. (Choudhary 1992) Because of the domain of $\mathbb{R}$, the set of performance metrics constitutes an ordered set $(\mathbb{R},\leq)$, i.e. $\forall d_1,d_2 \in \mathbb{R} : d_1 \rightarrow d_2 \iff d_1 \leq d_2$. In this paper we use the simple metric of *transit time t* as a quantified performance metric. Given two transit time metrics t_1 and t_2, the better performance is associated with the lower value, following the intuitive interpretation of the time-based performance metrics. Apart from the transit time, typical performance metrics in business and engineering workflows include other time-based metrics (e.g., time-to-market, activity processing time, queueing time) or monetary value based metrics (e.g., monetary emerging costs); and more complex metrics (e.g., resource utilization).

Simulation by DEVS To obtain the performance metric of transit time, we translate workflows to DEVS (Zeigler et al. 2000). Given its orientation towards queueing systems and performance modeling (Zeigler et al. 2000), we determine DEVS to be an appropriate formalism for the performance analysis of workflow models. DEVS is a discrete-event formalism, consisting of Atomic DEVS models (defining behaviour) and Coupled DEVS models (defining structure). Atomic DEVS models are defined as an 8-tuple $\langle X,Y,S,q_{init},\delta_{int},\delta_{ext},\lambda,ta\rangle$ (Van Tendeloo and Vangheluwe 2017a; Van Tendeloo and Vangheluwe 2018). Coupled DEVS models are defined as a 7-tuple $\langle X_{self},Y_{self},D,MS,IS,ZS,select\rangle$. To keep our approach as general as possible, we do not tie ourselves to one particular workflow/process modeling formalims, but rather, we provide a mapping of each of the original 20 *van der Aalst* workflow patterns (van der Aalst et al. 2003) to DEVS models. Given these patterns, every workflow can be mapped on to DEVS and simulated.

Modeling the stochastic nature of workflows To obtain meaningful results, the workflow model must be calibrated with activity execution times. These execution times are, however, often based on distributions, as some variation will be present in this value. For this reason, the time taken by activities is best defined through a distribution, thereby having a further impact on the complexity of the time taken by the workflow. Additionally, this distribution likely evolves over time, depending on the number of iterations that have previously occurred. A typical example in model-driven engineering workflows is the activity of creating a model, which gradually takes less time as the workflow iterates through it, since the modeler does not have to restart from scratch.

Yet another stochastic element in the workflow is the decision: the selected outgoing branch depends on some condition that can only be evaluated at execution time. We therefore sample the choice from on a distribution, which is again likely to evolve over time. For example, the first time an engineering model is simulated for a property, it is likely that there are still some unsatisfied elements in the model, but this chance gradually decreases as the check is more often performed.

Modeling resource constraints An additional facet of workflows are the constraints imposed by the resources available for the specific activities. The execution of an activity demands specific amounts of specific resources types, such as a CPU core, an engineer, a software license, etc. Depending on the other (concurrently) execution activities, these resources might not be available, meaning that the activity cannot start execution yet. By introducing these limitations in our simulation, it becomes possible to determine

the effect of obtaining additional resources (e.g., hiring an additional employee), or releasing it. It thus becomes possible to optimize the available resources, while taking into account the impact on workflow enactment time.

Implementation Our approach is implemented in the Modelverse (Van Tendeloo and Vangheluwe 2017b), our MPM environment. In the context of this paper, the Modelverse presents the users with a workflow modeling environment, where the mapping to and simulation of DEVS models remains completely hidden.

Running Example

To illustrate our approach, we use the running example shown in Figure 1. The workflow depicts the simplified engineering scenario of selecting the appropriate configuration of a hardware module through modeling and hardware-in-the-loop (HIL) simulations. This is commonly done for example for Automated Guided Vehicles (AGVs) (Dávid et al. 2016). The workflow is augmented with the identifiers of the relevant *van der Aalst* workflow patterns at the appropriate locations.

First, the *requirements are defined*. Subsequently, the *Multi-body model is being created*, or updated, depending on the iteration. The workflow is then split into two parallel branches. In one branch, ten independent *HIL Simulation*s are carried out in a parallel way to measure the physical attributes of the system and obtain statistical characteristics, such as the mean and variance. In the other branch, the model of the system undergoes formal *model checking* to verify the satisfaction of the safety requirements. Both branches require a significant amount of CPU time for the involved computations. After the synchronization of the branches, results are evaluated. If requirements are met, the workflow finishes; otherwise another iteration of engineering work is enforced.

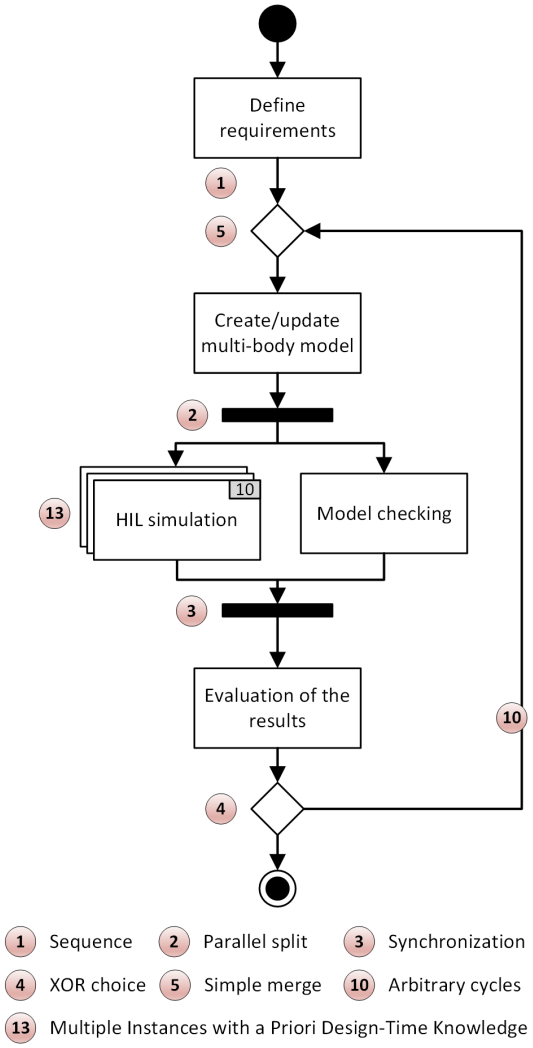

Figure 1: The workflow of selecting the optimal hardware component.

3 TRANSLATING WORKFLOWS TO DEVS

In order to simulate the enactment of the workflow model using DEVS, an automated translation between the two is necessary. Since DEVS consitutes Atomic and Coupled DEVS models, the translation consists of two parts.

The first part is the Atomic DEVS models, defining behaviour. As the behaviour of the workflow nodes is fixed, these can actually be handcoded in Atomic DEVS model. For example, the behaviour of a Parallel Split is independent of how it is used in the workflow. To tackle this part of the translation, we create a library of Atomic DEVS models of which the elements can be parameterized with runtime information (e.g., number of incoming links for Synchronization).

The second part is the Coupled DEVS model, defining structure. Given the set of Atomic DEVS building blocks, we must translate each workflow node to the equivalent Atomic DEVS model, and create

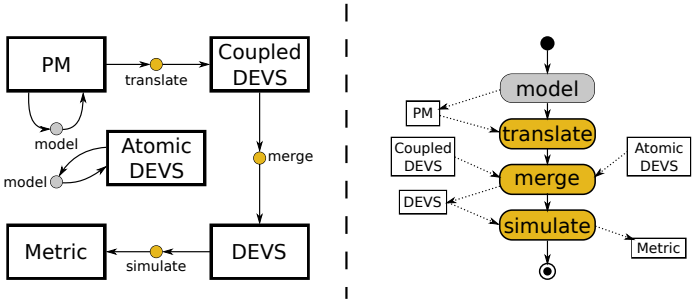

Figure 2: FTG+PM model of our approach.

the correct links between the two of them. For example, the sequence of two activities is translated to a connection in the Coupled DEVS model. Only this part of the translation is specific to the workflow model.

This is summarized in the FTG+PM (Lúcio et al. 2013) model shown in Figure 2. At the left-hand side, the Formalism Transformation Graph (FTG) presents the different formalisms and activities between them. At the right-hand side, the Process Model (PM) presents the workflow (process) as an instance of the FTG, showing the order in which activities are invoked and how the models are used.

3.1 DEVS Library

The first step is to create a library of Atomic DEVS models, one for each workflow node presented by van der Aalst et al. (2003). Due to space restrictions, we only present the patterns used in our running example.

All elements have a similar structure, with an input and output port termed "resource_in" and "resource_out", respectively. Over these ports, there is a control token that passes from node to node. As soon as a node receives the control token, it knows that the activity before it has terminated. Most workflow nodes subsequently pass on the token in one way or the other (e.g., split it or wait for multiple). Further details are given next for each type of workflow node.

Activity The activity takes in the control token and passes it on after some time. While not a pattern, it is the basic building block of a workflow, and could be considered as the Sequence pattern (Pattern 1). In DEVS, this can simply be modeled as shown in Equation 1. Essentially, the atomic DEVS model is either active (i.e., currently executing) or inactive (i.e., waiting for control). When the model is active, it outputs the token after some time $t_{process}$, otherwise it passivates.

Parallel Split The parallel split (Pattern 2) merely duplicates a token instantaneously, thereby effectively starting concurrent branches. It is therefore identical to the activity, but instead of taking $t_{process}$ time to process the token, it happens instantaneously. The control token is put on the output port, to which all subsequent nodes are connected. As such, the splitting of the token happens automatically. The DEVS specification is not repeated, as it is identical to Equation 1, but with $t_{process}$ set to zero.

Synchronization Synchronization (Pattern 3) has to wait for all incoming control flows to send a token. From the patterns definition, we know that only a single control token is sent on each branch (as the input places are guaranteed to be safe). Therefore, we can simply keep a counter, counting how many incoming tokens we need to receive, before we forward it ourselves. This counter is initialized with the number of incoming branches, which is specific to the workflow. This is shown in Equation 2.

$$Activity = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (1)$$
$$X = \{ControlToken\}$$
$$Y = \{ControlToken\}$$
$$S = \{active, inactive\}$$
$$q_{init} = (inactive, 0.0)$$
$$\delta_{int} = \{active \to inactive\}$$
$$\delta_{ext} = \{((inactive, \_), ControlToken) \to active\}$$
$$\lambda = \{active \to ControlToken\}$$
$$ta = \{active \to t_{process}, inactive \to \infty\}$$

$$Synchronization = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (2)$$
$$X = \{ControlToken\}$$
$$Y = \{ControlToken\}$$
$$S = \mathbb{N} \cup \{inactive\}$$
$$q_{init} = (inactive, 0.0)$$
$$\delta_{int} = \{0 \to inactive\}$$
$$\delta_{ext} = \{((i \in \mathbb{N}, \_), ControlToken) \to i - 1\}$$
$$\lambda = \{0 \to ControlToken\}$$
$$ta = \{0 \to 0.0, \mathbb{N} \setminus \{0\} \to \infty, inactive \to \infty\}$$

Exclusive Choice The exclusive choice (Pattern 4) makes a decision depending on some condition, thereby passing the control token to only a single branch. In the context of our workflow, we need to make it such that the decision varies between different invocations, as otherwise the workflow might become stuck in a loop. Later in this paper, we make this choice probabilistic, thereby circumventing these problems.

Simple Merge The simple merge (Pattern 5) merges different branches again as soon as a single one of them proceeds. This is commonly used after an exclusive choice, when the conditional part is finished, but can also be used in other scenarios where branches must merge. In contrast to synchronization, the simple merge continues for each token that comes in and does not wait for any other input. This definition is again similar to that of an activity, as it essentially takes in a control token and immediately passes it on to the next node. As such, the definition is identical to Equation 1, but with $t_{process}$ set to zero.

Multiple Instances with *a priori* Design-Time Knowledge The multiple instance workflow node (Pattern 13) is a more complex node, which spawns several instances of a single activity concurrently. The number of instances is defined at design time, making it possible to know beforehand how many instances there will be. Nonetheless, this information is still unknown in our Atomic DEVS model, as the library blocks are generic, and must therefore be taken in as a parameter. Control only progresses as soon as all spawned threads are finished, meaning that we operate synchronously. For the moment, this definition is again the same as the activity, shown in Equation 1, as nothing prevents us from spawning these elements concurrently immediately. Indeed, given that each instance takes $t_{process}$ amount of time to workflow, but they can all run concurrently, the total time taken is again $t_{process}$.

Initial While not an actual pattern, each workflow model needs exactly one initial node, which is where the workflow starts. This has its own specific DEVS model, which starts with the control token and immediately hands it over to the next node. The DEVS model is shown in Equation 3. Essentially, the model immediately sends out the token and then passivates in the inactive state.

$$Initial = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (3)$$
$$X = \{\}$$
$$Y = \{ControlToken\}$$
$$S = \{active, inactive\}$$
$$q_{init} = (active, 0.0)$$
$$\delta_{int} = \{0 \to inactive\}$$
$$\delta_{ext} = \{\}$$
$$\lambda = \{active \to ControlToken\}$$
$$ta = \{active \to 0.0, inactive \to \infty\}$$

$$Finish = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (4)$$
$$X = \{ControlToken\}$$
$$Y = \{\}$$
$$S = \mathbb{R} \cup \{inactive\}$$
$$q_{init} = (inactive, 0.0)$$
$$\delta_{int} = \{\}$$
$$\delta_{ext} = \{((inactive, e), ControlToken) \to e\}$$
$$\lambda = \{\}$$
$$ta = \{* \to \infty\}$$

Finish Similar to the initial node, the finish node is not actually a pattern, although it is required to indicate that the workflow has finished. This again has its own DEVS model, which starts in a passivated

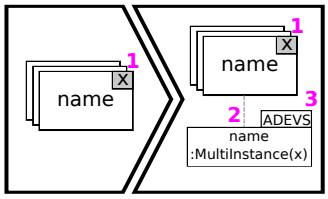

Figure 3: Transformation rule for activities.

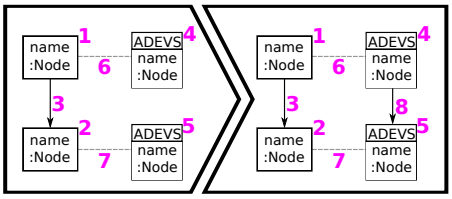

Figure 4: Transformation rule for control flow links.

state. As soon as the control token arrives, the elapsed time is stored in the state, as it will be used in the simulation termination condition later on. This elapsed time is the time taken by the workflow, as no other events ever arrive in this model. The DEVS model is shown in Equation 4.

3.2 Mapping through Model Transformations

With all Atomic DEVS models defined, the next step is to replicate the original workflow semantics with them. For this, we will make instances of the DEVS library elements for each of the workflow nodes, and copy the relevant connections between them.

Model transformations are ideally suited for this operation, as they allow for pattern detection and replacement. A model transformation defines a left-hand side (LHS), right-hand side (RHS), and negative application condition (NAC). For each application of the transformation, a match for the pattern in the LHS is searched in the model. When found, the match is replaced with the RHS if simultaneously the NAC is not matched. Model transformations generate traceability links, which store the origin of a specific construct in the target. This can be used subsequently for matching, or later on for debugging. For example, each process node is linked to an instance of the activity Atomic DEVS model through such a traceability link. These links are later on used to find translated elements: with all nodes matched and traceability information stored, linking the control flow edges becomes trivial.

An example transformation rule for an activity is shown in Figure 3. Here, the Multi-Instance node is translated to the instantiation of an Atomic DEVS model. Subsequently, a transformation rule for the edges is shown in Figure 4. Here, we match on an abstract superclass (*Node*) and the related DEVS Atomic DEVS models, through the traceability links (links 6 and 7). Due to space restrictions, the complete set of rules and the schedule is not shown, although these are similar to the previous examples.

3.3 Implementation

The approach was implemented in the Modelverse (Van Tendeloo and Vangheluwe 2017b), our Multi-Paradigm Modeling (MPM) environment. When enacting the FTG+PM model shown before, the Modelverse automatically loads the appropriate formalism (Process modeling) and allows users to model with it. When users are content with the workflow model, enactment continues with the (automated) mapping to a Coupled DEVS model. This is then combined with the Atomic DEVS model library created beforehand to create a complete DEVS model. This DEVS model is serialized and sent to PythonPDEVS (Van Tendeloo and Vangheluwe 2016), our DEVS simulator, through the use of external services (Van Mierlo et al. 2018). PythonPDEVS simulates the model and returns the simulation results, which get stored in the Modelverse.

4 CALIBRATION OF THE WORKFLOW MODELS

To evaluate the performance of the worklow, we augment its structure with multiple stochastic parameters, capturing the dynamics of the workflow. As Figure 5 shows, each activity is augmented with an *estimated execution time* (blue), and the characterization of the *evolution of the execution time* (red) over multiple iterations. For clarity, we only show the evolution function for *create/update multi-body model*, as this is a constant function for all other activities. Finally, the outgoing branches of decision nodes are also augmented with a quantitative *decision function* (green).

The characteristics of the estimations and the resulting functions can be determined by looking into historical data, for example. At this point, our example values rely on rules of thumb taken from industrial partners. We now elaborate on the above parameters in detail.

4.1 Estimated execution time of the activities

The execution time of the activities is estimated by a normal (Gaussian) distribution. The distribution is set so that (i) its expected value represents the guess of the estimator, and (ii) its variance yields 80% of the estimations within a 20% error range. The latter characteristic is achieved by setting the variance σ relative to mean μ, resulting in the function of $t(a) = N(\mu, 0.15625\mu)$.

4.2 Evolution of the execution time

The previous estimates of the activity execution times might evolve during the subsequent iterations of the workflow. For example, an activity might cache previous results (e.g., in simulation) or use them as a basis (e.g., modeling from scratch vs. updating a model). To capture this aspect, we use an aggressive exponential function $e^{-1/0.7i}$, where i denotes the number of iterations. This factor is used to scale down the original estimates as follows: $t(a,i) = t(a) * e^{-1/0.7i}$. The function results the following scale factors for the first few iterations: 1.0, 0.2397, 0.05743, 0.01376, 0.00329, 0.0008. As shown in Figure 5, there is only one activity (*Create/update multi-body model*) with a decreasing execution time over the various iterations. Of course, the execution time might also remain constant or even increase.

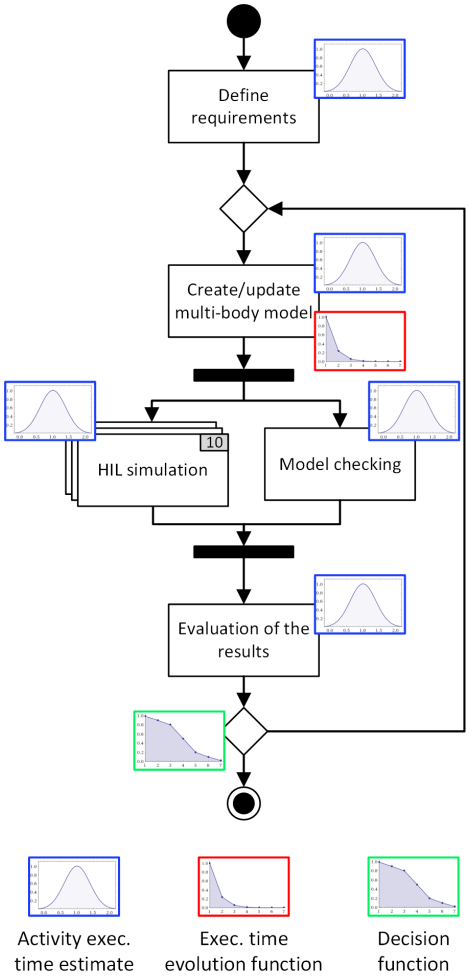

Figure 5: The augmented workflow.

4.3 Decision function

Deciding whether or not to re-iterate over a part of the workflow also happens with a given probability. A strong assumption against the workflow is its convergence towards a final solution. This also means that the chance of having to re-iterate over previously carried out activities has to decrease over time. We model this in a similar vein to the evolution of the execution time, again keeping track of the number of iterations. The function results the following probabilities of having to reiterate for the first few executions of the workflow: 0.99, 0.9, 0.8, 0.5, 0.2, 0.1.

4.4 Incorporating the stochastic parameters in the DEVS models

There are only three Atomic DEVS models that are influenced by these extensions. The first two, Activity and Multi-Instance, are actually similar, as they merely have to update their $t_{process}$ from a constant to a sample from a random distribution. To cope with the evolution of the execution time, both types of workflow nodes keep a counter of how many times they have been executed. This is trivial to incorporate in the DEVS model and is therefore not shown explicitly. The third workflow node, the Decision node, similarly gets augmented with a counter which counts how many times it was executed. This counter is passed on to the distribution function when sampling for which output port to use.

5 MODELING RESOURCE CONSTRAINTS

Another workflow constraint is resource utilization. There are several types of resources, such as recurring costs (e.g., employees), one-time costs (e.g., software licenses), or a combination of both (e.g., CPUs). The constraints imposed by these resource requirements have a significant effect on the performance of the workflow. Indeed, if multiple activities are scheduled in parallel, but only enough resources are available for one (e.g., only one employee), these activities are effectively sequentially. As such, the inclusion of resources is an important consideration. For simplicity, we consider a generic "resource" in further discussions, although this could be any relevant resource.

5.1 Implications of Resources

In our running example, the use of resources becomes important in the concurrent part, where HIL simulation and model checking are done simultaneously. Both activities can be done automatically, meaning that relevant resources are for example software licenses (e.g., for the tool performing the simulation or model checking) and CPU cores (to execute the activities on). As only activity and multi-instance workflow nodes consume a non-trivial amount of resources, we focus on them next.

Activity The first workflow node that is influenced, is the activity. Assuming that an activity has some resource requirements, which it needs throughout the complete duration of the activity. Should this not be the case, the activity can be split up in multiple distinct activities. As such, before starting the execution of the activity, we request all required resources. Only as soon as all resources are acquired, will the activity start executing. Upon termination of the activity, resources are released again.

Multi-Instance The multi-instance node is similar to the activity node, except that multiple instances are spawned as soon as possible. Each instance is independent of the others, meaning that some instances could already be spawned, depending on resource availability. Instead of requesting all resources, the resources for a single instance are requested atomically. For example, if five instances have to be spawned, we make five individual requests, each requesting all resources required by the instance. If for example only two requests can be granted at that time, at least these two instances already acquire the resources and can start execution. After an instance has finished, its resources are already released. As soon as resources become available, additional instances are spawned.

5.2 Incorporating resources in the DEVS models

We now include these concepts in the Atomic DEVS models discussed before. As only two types of nodes need resources, we only consider these two. Additionally, the resource handler, being the entity which is responsible for the acquisition and release of resources, is modeled as well. To handle the communication between these nodes and the resource handler, all relevant nodes are augmented with a *resouce_out* and *resource_in* port, all connected to the single resource handler.

Activity For the activity, we merely add two new states to make a request for the resource and wait for the reply. Only upon a positive reply does the activity start its execution and its $t_{process}$ starts to count down. When activity execution is finished, resources are released in the same output function where the control token was passed on. A DEVS specification is given in Equation 5.

Multi-Instance The Multi-Instance node is a bit more difficult, as it considers multiple instances, all requiring the same resources. When the node gets the control token, it immediately requests all resources for all instances to be spawned. Upon receiving a positive reply from the resource handler, the $t_{process}$ for one instance is sampled from the distribution and the countdown starts. These timers run in parallel, modeling the concurrent execution of a multi-instance node. Note that each instance samples from the same distribution, although the actual times might vary. Instances that acquired their resources at the same time, might thus finish at different times. Only when all instances have finished execution will the model passivate again. A full DEVS specification is not presented in this paper due to space restrictions, although it can be found in our implementation.

Resource Handler The resource handler is responsible for keeping track of resources. Incoming requests are put in a queue, and, when resources are available, the request is processed. Processing a resource means that the resources are acquired in the resource handler (i.e., a counter is decremented), and a reply is sent to the requester. When a request for a release is sent, the resource is immediately released (i.e., a counter is incremented).

The logic of the request handler can be complex, such as request prioritization, different resource types, and deadlock resolution. In our case, we only present a minimal example resource handler, managing a single type of resource, for which each request operates on exactly one resource instance. This Atomic DEVS model is given in Equation 6.

$$
\begin{aligned}
Activity &= \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (5)\\
X &= \{ControlToken, Resources\}\\
Y &= \{ControlToken, Request, Release\}\\
S &= \{active, inactive, request, waiting\}\\
q_{init} &= (inactive, 0.0)\\
\delta_{int} &= \{active \to inactive, request \to waiting\}\\
\delta_{ext} &= \{((inactive, \_), ControlToken) \to request,\\
&\quad ((waiting, \_), Resources) \to active\}\\
\lambda &= \{active \to [ControlToken, Release],\\
&\quad request \to [Request]\}\\
ta &= \{active \to t_{workflow}, inactive \to \infty,\\
&\quad request \to 0, waiting \to \infty\}
\end{aligned}
$$

$$
\begin{aligned}
ResourceHandler &= \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \quad (6)\\
X &= \{Request, Release\}\\
Y &= \{Resources\}\\
S &= [Request] \times \mathbb{N}\\
q_{init} &= (([], resources), 0.0)\\
\delta_{int} &= \{([R, r], i) \to ([R], i-1)\}\\
\delta_{ext} &= \{((([R], i),), Request) \to ([R, Request], i),\\
&\quad ((([R], i),), Release) \to ([R], i+1)\}\\
\lambda &= \{([R, r], i) \to Resources(r)\}\\
ta &= \{([R], i > 0) \to 0,\\
&\quad ([], i) \to \infty,\\
&\quad ([R], 0) \to \infty\}
\end{aligned}
$$

6 PERFORMANCE EVALUATION

After mapping the workflow model to the DEVS model and simulating it, the performance metric of transit time is obtained. Figure 6 shows the performance results of our running example, charted against the number of available resources. The chart shows the standard box plot features (i.e. minimum, first quartile, median, third quartile, maximum, and outliers) of the single simulation cases. For each scenario, 1000 simulations have been carried out.

The sharp decrease of the transit time is due to the more and more parallel threads spawned from the *HIL Simulation* activity (Figure 1). We spawn at most 11 concurrent activities (10 simulators and 1 model checker). This explains why with 6 available resources, time doesn't decrease further: we need at least two "phases" of executing the activities. Subsequently, when 11 resources are available, we see another minor decrease in time, since then all activities can then execute concurrently. More than 11 resources are not shown, as then we just have spare resources which are never used.

Note that there are several "bands" in which results are clustered. Indeed, the majority of the execution runs are clustered in the boxplots, which for the most likely scenario. However, we see that there are scenario's that cluster together: these values depend on the decision that was made at the end of the process. If we immediately decide that results are fine, the lowest "band" of results is achieved. Similarly, the topmost results indicate that the decision was made to redo the modeling for several times. Of course, results vary depending on the model and used parameters.

Such results can be used in various ways for workflow optimization. Given an appropriate cost function of resources, one can identify the appropriate amount of resources to be added to the workflow for an optimal configuration. The chart shows no relevant decrease of the transit time after case 5, which (in the absence of a more complex cost function) may denote the optimum of the amount of added resources. The

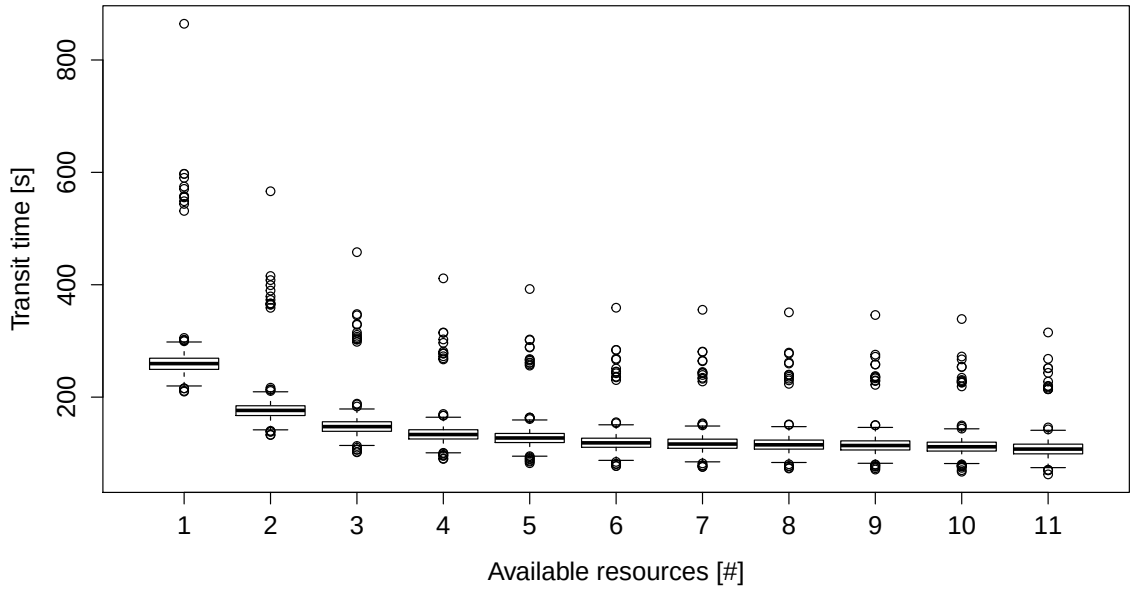

Figure 6: Results of the simulations.

chart also shows a large variance in transit times, shown by the circles denoting the outliers. This is an indicator of brittle workflows with a hidden hazard factor, which should be investigated.

7 RELATED WORK

Due to the semantic variety of different workflow formalisms, there are is no single solution for the performance evaluation of workflow models in general.

Alshareef et al. (2016) map workflows to parallel DEVS models with the intent of behavior specification. The authors base themselves on UML Activity diagrams. In our work, we aim at a more general set of workflow formalisms and provide mappings for every *van der Aalst* pattern. UML Activity diagrams do not include all these patterns. Additionally, our intent of mapping to DEVS is the performance evaluation of stochastic workflows in combination with resources.

The idea of using DEVS for performance evaluation already popped up in the work of Cohen et al. (1985). The authors view discrete event systems as linear algebras, in which the periodical behavior (i.e., repeatedly performed activities) can be characterized by solving an eigenvalue and eigenvector equation. Numeric algorithms are used to solve the equations and link the results to performance metrics of manufacturing systems. In our approach, instead of modeling with DEVS, we allow the high-level, more appropriate workflow formalisms to be used and support mapping to DEVS in the background.

The linchpin of assessing workflow performance is the calibration of the workflow as realistically as possible. Automating this step is a big step towards realistic simulation results. The most natural way to do so is by processing data from the previous runs of the workflow. Li et al. (2004) extend the WF-net formalism with timing information and provide a formal framework for assessing the lower bound of average turnaround time of the workflow. The timing information, however, is not modeled by statistical distributions, but scalar metrics. Xiao et al. (2006) focus more on the problem of the competition for the limited resources within the workflow. They propose a method based on queuing theory in order to analyze the time performance of a workflow. Our DEVS-based approach can be also extended to an explicit queuing formalism. Miu and Missier (2012) use learning algorithms over historical data to predict execution times of single activities based on the characteristics (instances and attributes) of their input data. Although in our running example we only focused on the control flow of the process, the approach presented in this paper fits well with our previously presented approach, based on the FTG+PM formalism. (Lúcio et al. 2013)

The rich semantics of the FTG+PM formalism allow reasoning not just over the syntactic characteristics of input data, but also the semantic characteristics, resources and costs.

Our approach solves the dual problem of the resource constrained project scheduling problem (RCPSP) (Artigues et al. 2007). In the RCPSP, instead of the workflow performance, it is the scheduling of activities that has to be determined in the presence of resource constraints and timing information. Our approach aligns with this problem nicely, and can be trivially extended to cover such problems as well.

8 CONCLUSIONS

In this paper, we presented an approach for simulating performance metrics of engineering workflow models. Our approach relies on the automated translation of engineering workflow models to DEVS models, and the capability of the latter to simulate stochastic performance metrics, such as transit time. Our approach supports each of the *van der Aalst* control patterns (van der Aalst et al. 2003). This means that our approach is compatible with the vast majority of workflow and process modeling formalisms. Our proof-of-concept implementation integrates with the FTG+PM framework seamlessly. We presented how stochastic behavior and resource constraints can be modeled in order to gain more realistic performance metrics.

In future work, we plan to further investigate DEVS simulations for different performance metrics (such as queuing time, resource utilization), and feeding back the results to actually improve the process. The approach presented in this paper integrates naturally with our inconsistency management framework (Dávid et al. 2016), in which the inconsistencies and their quantitative metrics (Dávid et al. 2016) serve as performance metrics to the workflow.

ACKNOWLEDGEMENTS

This research has been partially funded by a BOF DOCPRO-1 grant of the University of Antwerp, Belgium; and by a PhD fellowship from the Research Foundation - Flanders (FWO).

REFERENCES

Alshareef, A., H. S. Sarjoughian, and B. Zarrin. 2016. "An Approach for Activity-based DEVS Model Specification". In *Proceedings of the Symposium on Theory of Modeling & Simulation*, TMS-DEVS '16, 25:1–25:8. San Diego, CA, USA: Society for Computer Simulation International.

Artigues, C., S. Demassey, and E. Neron. 2007. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE.

Choudhary, B. 1992. *The Elements of Complex Analysis*. New Age International. ISBN 978-81-224-0399-2.

Cohen, G., D. Dubois, J. Quadrat, and M. Viot. 1985, Mar. "A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing". *IEEE Transactions on Automatic Control* 30(3):210–220.

Dávid, I., J. Denil, K. Gadeyne, and H. Vangheluwe. 2016. "Engineering Process Transformation to Manage (In)consistency". In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016)*, 7–16. http://ceur-ws.org/Vol-1717/.

Dávid, I., B. Meyers, K. Vanherpen, Y. Van Tendeloo, K. Berx, and H. Vangheluwe. 2017. "Modeling and Enactment Support for Early Detection of Inconsistencies in Engineering Processes". In *2nd International Workshop on Collaborative Modelling in MDE*.

Dávid, I., E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, and K. Vanherpen. 2016. "Towards Inconsistency Tolerance by Quantification of Semantic Inconsistencies.". In *COMMitMDE@ MoDELS*, 35–44.

Li, J., Y. Fan, and M. Zhou. 2004, March. "Performance modeling and analysis of workflow". *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 34(2):229–242.

Lúcio, L., S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukšs. 2013. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". *Lecture Notes in Computer Science* 7916:182–202.

Miu, T., and P. Missier. 2012, Nov. "Predicting the Execution Time of Workflow Activities Based on Their Input Features". In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 64–72.

Mosterman, P. J., and H. Vangheluwe. 2004, September. "Computer Automated Multi-Paradigm Modeling: An Introduction". *Simulation* 80(9):433–450.

van der Aalst, W., A. ter Hofstede, B. Kiepuszewski, and A. Barros. 2003, Jul. "Workflow Patterns". *Distributed and Parallel Databases* 14(1):5–51.

Van Mierlo, S., Y. Van Tendeloo, I. Dávid, B. Meyers, A. Gebremichael, and H. Vangheluwe. 2018. "A Multi-Paradigm Approach for Modelling Service Interactions in Model-Driven Engineering Processes". In *Proceedings of Mod4Sim*, Mod4Sim, part of the Spring Simulation Multi-Conference, 565–576.

Van Tendeloo, Y., and H. Vangheluwe. 2016, April. "An Overview of PythonPDEVS". In *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, edited by C. W. RED, 59 – 66: Éditions Cépaduès.

Van Tendeloo, Y., and H. Vangheluwe. 2017a, December. "Classic DEVS Modelling and Simulation". In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, 644 – 656: IEEE.

Van Tendeloo, Y., and H. Vangheluwe. 2017b, December. "The Modelverse: a Tool for Multi-Paradigm Modelling and Simulation". In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, 944 – 955: IEEE.

Van Tendeloo, Y., and H. Vangheluwe. 2018. "Extending the DEVS Formalism with Initialization Information". *ArXiv e-prints*.

Xiao, Z.-j., H.-y. Chang, and Y. Yi. 2006. "Method of workflow time performance analysis". *COMPUTER INTEGRATED MANUFACTURING SYSTEMS-BEIJING-* 12(8):1284.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. second ed. Academic Press.

AUTHOR BIOGRAPHIES

HANS VANGHELUWE is a Professor in the department of Mathematics and Computer Science at the University of Antwerp (Belgium) and an Adjunct Professor in the School of Computer Science at McGill University (Canada). He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results.

ISTVÁN DÁVID is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). He obtained his Master's degrees from the Budapest University of Technology and Economics. His research interests include process modeling, inconsistency management in multi-model/multi-paradigm settings, complex event processing and language engineering.

YENTL VAN TENDELOO is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). In his Master's thesis, he worked on MDSL's PythonPDEVS simulator, a simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS, grafted on the Python programming language. The topic of his PhD is the conceptualization, development, and distributed realization of a new (meta-)modelling framework and model management system called the Modelverse.