Int j simul model 17 (2018) 1, 55-68
ISSN 1726-4529 Original scientific paper

MAPPING SPEM PROCESS SPECIFICATIONS TO ACTIVITY CYCLE DIAGRAMS

Pagliares, R. M.[*,**] & Hirata, C. M.[*]
[*] Computer Science Department - IEC - Instituto Tecnológico de Aeronáutica, ITA,
Pca. Mal. do Ar Eduardo Gomes, 50, 12228-900, São José dos Campos, SP, Brazil
[**] Universidade Federal de Alfenas, UNIFAL-MG, Alfenas, MG, Brazil
E-Mail: pagliares@bcc.unifal-mg.edu.br, hirata@ita.br

Abstract
The development of discrete event simulation models to simulate engineering processes is a complex endeavour because it requires a great deal of effort, time and expertise in different areas, in addition to knowledge of techniques of modelling and implementation. We aim to provide a model-driven approach to automate the generation of simulation models from process models. The approach consists of algorithms that map elements of SPEM (*Software & Systems Process Engineering Metamodel*) to XACDML (*eXtensible Activity Cycle Diagrams Markup Language*). SPEM is a notation used to model engineering processes. XACDML is a textual specification of ACD. ACD is graphical notation to represent discrete event simulation models. We extended XACDML to cope with SPEM hierarchical elements enabling the mappings between the models. We developed a tool and use it in an example in order to demonstrate the feasibility of our approach. The results indicate that the approach is feasible and reduces the burden associated with building simulation models of engineering processes.
(Received in June 2017, accepted in November 2017. This paper was with the authors 1 week for 2 revisions.)

Key Words: Software & Systems Process Engineering Metamodel (SPEM), Activity Cycle Diagrams (ACD), Automatic Model Generation, Discrete Event Simulation

1. INTRODUCTION

Process engineers and project managers are constantly monitoring and evaluating process performance in development projects in order to know its status and if the project goals will be achievable. The evaluation of process can give insights to potential process changes too. Proposing and evaluating process changes are generally complex tasks since they require knowledge of the process, inputs, available resources, used resources, and external factors that affect the project. Moreover, the task of evaluating process changes requires systematic approaches and techniques to support it. Discrete Event Simulation (DES) is one of the techniques. However, the development of DES models of engineering processes is a complex endeavour [1, 2]. It requires a great deal of effort, time and expertise in different areas, in addition to knowledge of techniques to model and implement simulators.

The goal is to provide a model-driven approach to automate the generation of DES models of SPEM process models. The approach uses SPEM [3] as notation for defining engineering processes and a textual representation of Activity Cycle Diagrams (ACD) [4] named XACDML [5] to represent DES models. An engineering process (process for short) modelled with SPEM is translated into an extended XACDML model containing the structure, behaviour and quantitative parameters of the simulation. The approach supports the translation by applying mapping algorithms. We extended XACDML to deal with modelling elements found in processes but not defined in XACDML models, such as *phases* and *milestones*.

Our approach is materialized in a tool that generates simulation models allowing tailoring the generation process to fit analysts' interests. The tool enables experimentation of the generated XACDML simulation model by transforming it, on-the-fly, into reusable *Java*

code. We employed the tool in a real project available in literature [6] as application of the approach.

The structure of this paper is as follows. In Section 2 we provide background on the notations used within our approach. Section 3 presents the related work. The approach uses mapping algorithms described in Section 4. In Section 5, we outline the tool support for the approach. Section 6 discusses the use of the tool with an example to demonstrate its applicability. Section 7 is devoted to discussions. We present concluding remarks in Section 8.

2. BACKGROUND

SPEM is a notation for modelling development processes. It is defined as a meta-model and as a UML profile [3] and has been used for authoring several processes used in engineering [3, 7]. SPEM provides building blocks representing methods, life cycles, roles, activities, tasks, and work products used in systems engineering processes. Processes modelled with SPEM are hierarchical and represented by a Work Breakdown Structure (WBS). The process WBS is constituted of distinct Work Breakdown Elements (WBEs) such as *phases, iterations, activities, milestones*, and *tasks*. SPEM is the most used language for modelling processes, despite of existence of other languages [7].

ACD is a graphical notation to model DES systems. Its power relies on its simplicity, what makes it a useful tool to enable stakeholders' communication. ACD describes the behaviour of *classes of entities* as a cycle, which spans from their creation to their death. Every cycle is composed of the alternation of two kinds of states (*active* and *dead*) that are connected by directed arcs that indicate the path the *entity* should follow. The *entity* leaves the system upon completion of the cycle. *Active states* perform the business logic in the model and act upon *entities*, creating, moving, and changing their attributes. Their duration can be determined in advance by the use of appropriate probability distributions. *Dead states* are passive repositories of *temporary entities* and *permanent entities* (*resources*). They can be thought as states in which *entities* wait for something to occur.

XACDML [5] is an ACD compliant XML specification for DES, designed to allow exchange of models among tools. XACDML is a well-formed and valid XML language. *Simulation analysts* can tailor a model represented with XACDML for different purposes by specifying parameters for the simulation model such as probability distributions for *active states*, and capacity and initialization of *dead states*. XACDML may also be used as an intermediary language, being a starting point for other DES modelling formalisms.

3. RELATED WORK

We found several works on the automatic generation of simulation models in the literature. Most of them are targeted to specific domains, although there are some domain agnostic approaches. There are also a growing number of works that frame the problem of generating simulation models in the context of Model Driven Development (MDD).

Çetinkaya et al. [8] present a MDD framework that addresses model continuity at the different stages of Modelling and Simulation (M&S) mirroring different models within a development process through transformation rules. The main goal of their work is to speed up the process of developing simulation models. The framework makes use of intermediate metamodels in addition to an executable simulation model. The authors suggest model transformations based on metamodels and the applicability of their framework is illustrated with a tool prototype.

Hollmann et al. [9] present CML-DEVS, a language that allows the representation of Discrete Event System Specification (DEVS) models in their abstract form in terms of

mathematical and logical expressions. CML-DEVS allows a *simulation analyst* to define a simulation model even if he/she does not have programming skills or knowledge of a particular modelling language of a specific M&S tool. CML-DEVS models are automatically transformed into the input models of different M&S tools through the use of transformation rules. The rules are implemented within a compiler in order to automate the transformations. The authors provide the transformation rules from DEVS to input models of DEVS-Suite [10] and PowerDEVS [11] tools.

Arias and Hirata [12] propose mappings of software models, annotated with performance information to a simulation model. Software models are based on UML deployment and state diagrams whereas the simulation model is represented with XACDML. The simulation model is translated to a simulation program for verification of performance requirements. The authors present an example in order to show the feasibility of the mappings.

To the best of our knowledge, the closest work to our current research is the work of Park et al. [2]. The authors propose an approach to derive a hybrid simulation model from a process model created with SPEM. The approach has steps that must be followed in order to derive a simulation model: mapping, modelling, and transforming. In the mapping step, the mapping between the elements of SPEM and DEVS-Hybrid is conducted. DEVS-Hybrid is an extension of DEVS formalism to the domain of hybrid software process simulation modelling. System Dynamics (SD) represents details concerning activity behaviours, and DES controls activity start/completion. In the modelling step, a process model is developed using a UML profile. The process model is transformed into a DEVS-Hybrid model by applying transformation rules in the transformation step. The authors also provide a tool to support their approach. In their approach, a *simulation analyst* must elaborate UML *use cases* to represent the WBS of a process and UML *activity diagrams* to represent the sequence of activities with input and output work products. Modelling these diagrams demands significant time from *simulation analysts* in order to transform the models, particularly in the absence of guidelines.

4. MAPPING SPEM MODELS TO EXTENDED XACDML

In this section, we describe how to map the elements of SPEM to elements of XACDML. SPEM provides some model elements that incorporate others (e.g. *phase*, *iteration*, and *activity*), allowing nesting of WBEs in a hierarchical fashion. Hierarchy cannot be modelled in XACDML, therefore, we extend it to artificially represent nested elements. Algorithms parameterized with quantitative data automate the mappings. Table I shows the algorithms' descriptions.

We apply the mapping algorithms in an example of a SPEM software process model that is translated into an extended XACDML simulation model to ease the understanding of the algorithms and XACDML extensions. Fig. 1 shows the process we use as example. It is a typical agile method for software development. The process has one *phase* named "*software production"* composed of several *releases.* Each *release* has the same structure. It includes a *planning and development activity* that starts with a *release planning* meeting where *user stories* are prepared to be implemented (prioritized, estimated, and split) and ends with several *iterations* of development in which *developers* perform the "*pair programming with test driven development" task*, before the next "*planning and development"* activity. For instance, the "*pair programming with test driven development" task* is performed by the *developer role* and has as input a *user story* that has been prepared during the "*release planning meeting"*. The *task* produces an implemented *user story* as output. A *milestone* named *"working software"* is reached at the end of the software production *phase*.

Table I: Mapping algorithms of SPEM elements to extended XACDML elements.

Algorithm	Description
#1	Maps SPEM *roles* to XACDML *permanent entities* and *resource dead states.*
#2	Maps SPEM *work products* to XACDML *temporary entities*, *dead states*, *generate active states*, and *destroy active states.*
#3	Maps SPEM *tasks* to XACDML *activities.* The mapping includes the new XML element named <*spem*> and the new attributes named, *spem_type*, *parent*, *dependency_type*, *condition_to_process*, *processing_quantity*, and *quantity_of_resources_needed*.
#4	Maps SPEM *iterations*, *activities*, *phases*, and *milestones* to *dummy active states* (i.e. *active states* that when performed do not advance the simulation clock). The mapping includes new attributes, such as *timebox* and *iteration_behaviour*.

We assume that processes may be quantifiable and data is available for adjusting the parameters of the XACDML model. XACDML models assume the specifications of the destinations of *entities* are made in an unequivocal manner. The SPEM process model of Fig. 1, after translation to XACDML, satisfies this assumption. We will refer back to this software process model in Section 6, where we experiment it with data taken from the literature.

Presentation Name	Index	Predecessors	Model Info	Type	Optional
Agile_process	0			DeliveryProcess	
Software production	1			Phase	
Release	2			Iteration	
Planning & development	3			Activity	
Release planning	4			TaskDescriptor	
User story			Mandatory Input		
User story			Output		
Customer			Primary Performer		
Developer			Primary Performer		
Iteration	5	4		Iteration	
Pair programming with test driven development	6			TaskDescriptor	
User story			Mandatory Input		
User story			Output		
Developer			Primary Performer		
Working software	7	1		Milestone	

Figure 1: SPEM process for an agile method for software development.

After applying the mapping algorithms described in Table I, an XACDML model is generated (Fig. 2). Notice that its contents are an excerpt of the final XACDML. Line 01 identifies the model. The value of the *id* attribute is derived of the name of root node in Fig. 1. The *acd* element is "closed" in line 44. The execution of the algorithms, discussed in what follows, generates the remaining contents in Fig. 2.

Fig. 3 shows the first mapping algorithm in pseudo code. It works by parsing the SPEM process model of Fig. 1. The results of applying this algorithm in the example can be seen in lines 2 and 4 in Fig. 2. Algorithm #1 receives as parameter an instance of *DeliveryProcess* (line 1). In SPEM, a *DeliveryProcess* represents the root node of the WBS representing the process. A SPEM *role* defines a set of related skills, competencies, and responsibilities of a set of individuals. For each SPEM *role* within the *DeliveryProcess*, an XACDML *permanent entity* is created and added to the output XACDML (lines 2-4). The statement at line 5 translates each SPEM *role* into its own *resource dead state*. Lines 6 and 7 of Fig. 3 allow *simulation analysts* to define the parameters for the *type* and *capacity* of the *dead state* to be generated and to include the *dead state* in the output XACDML.

```
<acd id="Agile process">
    <class id="Customer"/>  <class id="Developer"/>
    <class id="User story"/>
    <dead id="dq" class="Developer"> <type struct="QUEUE" size="10" init="4"/>  </dead>
    <dead id="q0" class="User story"> <type struct="QUEUE" size="100" init="29"/> </dead>
    <!--  BEGIN_Phase, Release, and Activity not shown. Similar to BEGIN_Iteration  -->
   <act id="Release_Planning" processing_quantity="BATCH">
      <stat type="CONST" parm1="480.0"/>
      <entity_class id="role1" prev="dq" next="dq"/>
      <entity_class id="temp ec" prev="q3" next="q4"/>
      <spem spem_type="TASK"
            parent="Planning & development"
            dependency_type="FINISH-TO-START"
            condition_to_process=" ALL-ENTITIES-AVAILABLE"
       />
   </act>
    <act id="BEGIN_Iteration" timebox="4320" processing_quantity="BATCH">
      <stat type="CONST" parm1="0.0"/>
      <entity_class id="temp ec" prev="q4" next="q5"/>
      <spem spem_type = "ITERATION"
            dependency_type = "FINISH-TO-START"
            condition_to_process = "ALL-ENTITIES-AVAILABLE"
       />
   </act>
   <act id="Pair programming with test driven development" processing_quantity="UNIT">
       <stat type="LOGNORMAL" parm1="1900" parm2="1200"/>
      <entity_class id="role1" prev="dq" next="dq" quantity_of_resources_needed="2"/>
      <entity_class id="temp ec" prev="q5" next="q6/>
       <spem spem_type = "TASK"
            parent = "BEGIN_Iteration"
            dependency_type = "FINISH-TO-START"
            condition_to_process = "ALL-ENTITIES-AVAILABLE"
      />
   </act>
   <act id="END_Iteration" processing_quantity="BATCH">
      <stat type="CONST" parm1="0.0"/>
       <entity_class id="temp ec" prev="q6" next="q7"/>
       <spem spem_type = "ITERATION"
           parent = "Planning & development"
           iteration_behaviour = "MOVE-BACK"
      />
   </act>
   <!--  END_Phase, Activity, and Release not shown. Similar to END_Iteration  -->
</acd>
```

Figure 2: Excerpt of the XACDML simulation model generated after applying the algorithms discussed in this section. Extensions made to XACDML are in boldface.

```
Algorithm#1 (DeliveryProcess deliveryProcess)
   for each SPEM  Role in deliveryProcess
      create an XACDML permanent entity;
      add the permanent entity to the ACD;
      create a resource dead state;
      configure resource dead state;
      add the resource dead state to the ACD;
```

Figure 3: Mapping algorithm #1 – "Maps SPEM *roles* to XACDML *permanent entities* and *resource dead states*".

Fig. 4 illustrates how SPEM *work products* are translated into XACDML elements. SPEM *work products* are artefacts consumed, produced, or modified by *tasks*. *Roles* perform *tasks* that use *work products* as input while producing other *work products* as output. The results of applying this algorithm can be seen in lines 3 and 5 in Fig. 2. The algorithm receives a *DeliveryProcess* as parameter (line 1). For each SPEM *work product* in the *deliveryProcess* (line 2), an XACDML *temporary entity* is created (line 3). Lines 4-6 are similar to the ones discussed in Fig. 3. We consider the initial *dead state* as the incoming one associated with the WBE of index "1" in the process WBS (*Software production*, in Fig. 1). The mapping algorithm of Fig. 4 identifies this *initial dead state* and creates a *generate active state* for it (lines 7-10). A *generate active state* creates *temporary entities* using a probabilistic distribution. In the process of Fig. 1, a *generate active state* that injects *user stories* into the *initial dead state* can be created. Not all systems need a *generate active state*. An alternative is configuring the initial quantity of *user stories* in the *initial dead state*. The tool we developed to support the mapping algorithms (see Section 5) allows the *simulation analyst* to include a *generate active state* in the XACDML model.

```
01  Algorithm#2 (DeliveryProcess deliveryProcess)
02      for each SPEM WorkProduct in deliveryProcess
03        create an XACDML temporary entity;
04        add the temporary entity to the ACD;
05        create a temporary entity dead state;
06        configure the temporary entity dead state;
07        firstWBE = deliveryProcess.getFirstWBE();
08        if (WorkProduct type equals "INPUT" && its associated WBE == firstWBE)
09           create a generate active state;
10           determine the probability distribution for the generate active state;
11         add the temporary entity dead state to the ACD;
12         if (temporary entity may be destroyed)
13            create a destroy active state for the temporary entity;
```

Figure 4: Mapping algorithm #2 – "Maps SPEM *work products* to XACDML *temporary entities*, *dead states*, *generate active states*, and *destroy active states*".

When a *temporary entity* is not used any longer in the simulation, it can be destroyed. This is indicated in lines 12 and 13 of Fig. 4. We determine whether a *temporary entity* is destroyed based on which *dead state* it is located. In our example, a *user story* can only be destroyed if it is already implemented. This means that the *temporary entity* named *User story* must be in the XACDML *dead state* that was generated for the output *work product* of the *"Pair programming with test driven development" task* of Fig. 1. Although the algorithm detects which *entities* may be destroyed, the *simulation analyst* can decide not to destroy them at all for reporting and debugging purposes. The XACDML excerpt shown in Fig. 2 illustrates the XACDML tags generated by applying the algorithm of Fig. 4 with the decisions of not creating an XACDML *generate active state* for *user stories* and not destroying implemented *user stories*.

The algorithm presented in Fig. 5 is used for SPEM *tasks* and generates the XACDML excerpt exhibited on Fig. 2 (lines 07-16, and 25-34). A SPEM *task* describes an assignable unit of work to a *role*. In XACDML, the equivalent of a SPEM *task* is the concept of *activity*. The algorithm statement *"create an XACDML activity;"* in line 2 of Fig. 5 is translated, for example, into the *<act id="Pair programming with test driven development">* tag on Fig. 2 (line 25). The value of the *id* attribute is derived from the *task* name.

We use quantitative data to configure the appropriate probability distribution and respective parameters for the duration of the *active states* (line 4 of Fig. 5). In our example,

the duration of the *activity* "*Pair programming with test driven development"* is sampled from a *lognormal* distribution with parameters values of 1900 minutes for *scale* and 1200 minutes for *shape*. This is reflected in the element *<stat type="LOGNORMAL" parm1="*1900*" parm2="*1200*"/>* of line 26 in Fig. 2.

1	Algorithm#3 (WorkbreakdownElement *task*)
2	create an XACDML *activity*;
3	configure the *entity classes* for the *activity*;
4	configure the probability distribution for the *activity*;
5	configure the extended XACDML elements and attributes for SPEM tasks
6	add the XACDML *activity* in the ACD;

Figure 5: Mapping algorithm #3 – "Maps SPEM *tasks* to XACDML *activities*".

An *activity* corresponds to an engagement of different type of *entities*, *permanent* and *temporary*. Both types carry information of their previous and next *dead states* as shown, for example, in lines 27 and 28 of Fig. 2. We created a new attribute for XACDML *permanent entity classes* named *quantity_of_resources_needed.* This attribute indicates the number of *resources* demanded to perform the *activity*. In our example, the "*pair programming with test driven development*" *activity* demands two idle *resources* to start servicing.

Line 5 of the mapping algorithm allows us to configure new *elements* and *attributes* of XACDML for SPEM support. The new element named *spem* (*<spem>* tag) consists of the mandatory attribute *spem_type* and the optional attributes *parent*, *dependency_type*, and *condition_to_process.* In addition to the new *spem* element, a new attribute named *processing_quantity* is used. When *processing_quantity* is set to "BATCH", it means all *temporary entities* in the incoming *dead state* are processed at once. If configured to "UNIT", only one is processed each time. The *spem_type* is used to configure the type of the WBE being mapped to XACDML (a *task*, in this case). The *parent* attribute allows identifying whether the *task* is nested inside other WBEs. Its value when set is used to discover the *timebox* of the parent WBE (we discuss the need for the *timebox* attribute in Section 5).

SPEM uses dependencies to define predecessor and successor relations among WBEs. The available dependency types are *Finish-to-Start*, *Finish-to-Finish*, *Start-to-Start, and Start-to-Finish*. The *Finish-to-Start* dependency type is by far the most common type of dependency [3] and the one we use in our implementation. Within our approach, the *dependency_type* and *condition_to_process* attributes are usually used together to provide better semantics for SPEM dependencies. When we set the *dependency_type* to "FINISH-TO-START", we indicate that a WBE may only start when its predecessor element has finished. In our approach, a predecessor finishes in one of two ways; when one *temporary entity* or when a *class of temporary entities* is served by a predecessor *active state*. A *class of temporary entities* represents a set containing all *temporary entities* of certain type. The attribute named *condition_to_process* is used to establish a condition that must be satisfied in order to the XACDML *active state* starts servicing. Its possible values are "SINGLE-ENTITY-AVAILABLE" and "ALL-ENTITIES-AVAILABLE". The former value is the default behaviour for XACDML. The latter is an option in the extended XACDML. In our example, we set 29 *temporary entities* in the *initial dead state* (see line 05 of Fig. 2.). This number represents the initial size of the *class of entities*. In that way, since the *active state* "*Pair programming with test driven development*" was configured with *dependency_type* = "FINISH-TO-START" and *condition_to_process* = "ALL-ENTITIES-AVAILABLE", it can only start servicing if its predecessor *active state* ("*Release Planning*") has already processed all the 29 *temporary entities*.

Fig. 6 illustrates the algorithm used for the creation of *dummy active states* (no simulation time is computed when running them). It is used to map SPEM *milestones*, *iteration*s, *activities*, and *phases* to the extended XACDML. For a SPEM *milestone*, this algorithm is called once. The algorithm is called twice for *phases*, *iterations*, and *activities*, generating a pair of XACDML *dummy active states* that delimitates them in time. The first *dummy active state* of the pair represents the BEGIN of a phase, *iteration*, or *activity*. The second *dummy active state* indicates its END. This mapping algorithm is rather similar to the algorithm #3 for *tasks*, with the difference that in addition to the *wbe* parameter of type *WorkBreakdownElement*, the algorithm also receives a parameter of type *String* that is used to name the *dummy active state* (line 1). Another difference resides in line 2 where only *temporary entity classes* are configured (we do not engage *permanent entities* in *dummy active states*). A third difference is the duration (line 4) that is set to zero, identifying it as an XACDML *dummy active state*. Line 6 adds the activity to the output XACDML.

```
1  Algorithm#4 (WorkBreakdownElement wbe, String name)
2    create an XACDML dummy active state and name it with the name parameter;
3    configure the temporary entity classes for the dummy active state;
4    configure the probability distribution for the activity to be CONST and equal to ZERO;
5    configure the extended XACDML elements for the dummy active state
6    add the XACDML wbe in the ACD;
```

Figure 6: Mapping algorithm #4 – "Maps SPEM *iteration*, *activities*, *phases*, and *milestones to dummy active states* with no simulation duration".

We use the recursive algorithm #5 (Fig. 7) to generate all XACDML *active states* from SPEM WBEs. In the tree-like WBS structure for processes in SPEM, *tasks* and *milestones* are leaves whereas *phases*, *iterations* and *activities* are internal nodes that may have children.

```
01  Algorithm#5 (WorkBreakdownElement wbe)
02    if (wbe type equals to "TASK")                    // leaf
03          Algorithm#3 (wbe);
04    else if (wbe type equals to "MILESTONE")          // leaf
05          Algorithm#4 (wbe);
06     else                                             // internal node {phase, iteration, or activity}
07          children =  wbe.getChildren();
08          Algorithm#4 (wbe, "BEGIN_"+ wbe.getName());        // BEGIN counterpart
09          for each child in children
10               Algorithm#5 (child);
11          Algorithm#4 (wbe, "END_"+ wbe.getName());          // END counterpart
```

Figure 7: Recursive algorithm used to translate *tasks*, *milestones*, *phases*, *iterations*, and *activities* to the extended XACDML *active states*.

The algorithm receives as parameter an object of type *WorkBreakDownElement* (line 01). If this object is of type *task* or *milestone* (leaf), the algorithm #3 or #4 is called (lines 02–05). Otherwise, the *wbe* object is a *phase*, *iteration*, or *activity* (internal nodes with children), and two *dummy active states* must be created, one for BEGIN and another for END. The BEGIN *dummy active state* and the END *dummy active state* are created in line 08 and in line 11, by calling algorithm #4. For each child in the *wbe*, a recursive call is made (line 10). When applied to our example, the execution of this algorithm generates lines 07-42 of the XACDML illustrated in Fig. 2.

5. TOOL SUPPORT

We developed a tool that supports our approach and implements the mapping algorithms. The tool provides graphical capabilities to the *simulation analyst* and is compliant with SPEM and XACDML. It takes as input a SPEM process and translates it to an output XACDML simulation model.

Fig. 8 shows a SPEM process imported into our tool. The process is shown on the right side of the figure. In addition to support the mapping algorithms, the tool also provides the functionality of transforming the generated XACDML model into *Java* code that can be reused on-the-fly by the tool itself, allowing out of the box experimentation, without human intervention. On-the-fly *Java* code is generated by the use of *eXtensible Stylesheet Transformation*, XSLT. Gil and Hirata [5] also used XSLT, however, we adapted their transformations to support the XACDML extensions and to make the output *Java* code more reusable. The generated *Java* code uses a component library for the development of DES models [13]. The component library includes an *executive* that supports the three-phase worldview. We extended it, implementing support for the new *elements* and *attributes* of the extended XACDML.

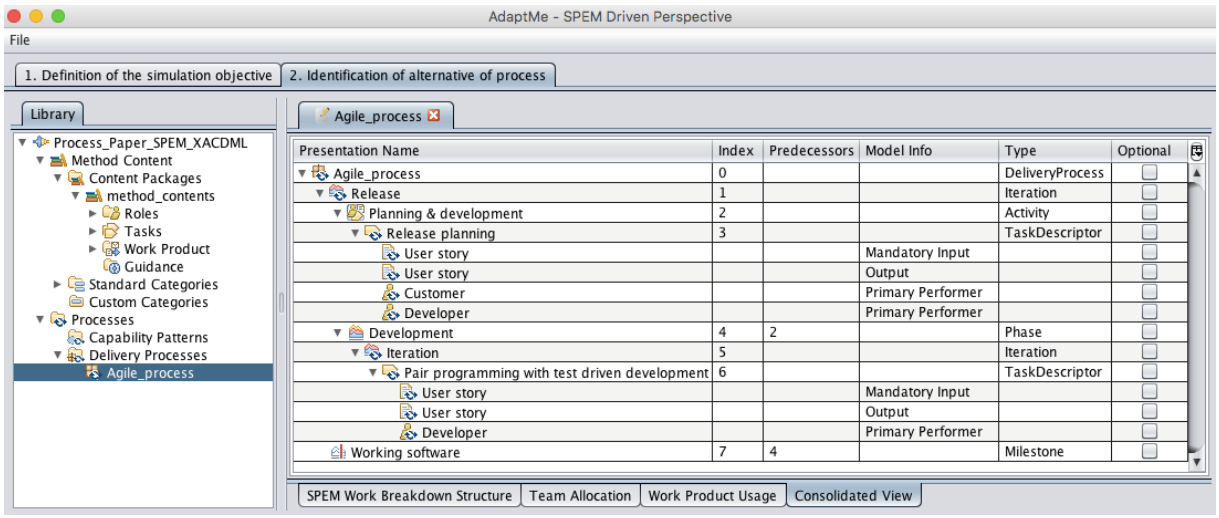

Figure 8: A SPEM process imported into our tool.

Fig. 9 shows a tool graphical support in which the duration of a SPEM *task* is configured with its corresponding probability distribution and parameters before generation of the extended XACDML model. In the lower part ("Extended XACDML attributes") we can use fields to configure all new XACDML *elements* and *attributes*. The fields are SPEM type sensitive. For instance, the field for "*quantity of resources needed to perform a task*" is only enabled when mapping SPEM *tasks*. The *timebox* field can only be configured for SPEM *activities* and *iterations*.

In what follows, we discuss the extensions to the simulation library [13]. In general they can be classified into extensions that establish conditions to the start of XACDML *active states* (evaluated at C-events) and extensions that implement the specific behaviour to be accomplished depending on the type of the SPEM WBE (executed at B-events).

One condition that must be evaluated before starting an *active state* is the condition configured with the attribute *condition_to_process*. As discussed, if this attribute is set to "ALL-ENTITIES-AVAILABLE", it specifies that the *active state* may only start when a *class of entities* has been produced (made available) by a predecessor *active state*. To make this possible, we define a *counter* for each object of type *ActiveState*. The *counter* is incremented

in its B when *temporary entities* are processed. The increment is based on the value of the XACDML *processing_quantity* attribute. If it is set to "UNIT", the *counter* is incremented by one. If it is set to "BATCH", the *counter* is incremented by the number of *entities* within the batch. At the C-event of an *active state*, the *counter* of the previous *active state* is compared with the *size* of the *class of entities*. If they are equal, the previous *active state* has finished processing all *entities* in the *class*, what satisfies the partial condition to the start of the *active state*.

Although no simulation time is spent in each *dummy active state* of the pair generated from SPEM *iterations* and *activities*, we use their *timebox* as parameter. This parameter determines the moment the END counterpart of the pair must be scheduled in the simulation calendar. We implement this correct schedule by keeping record of the time the BEGIN counterpart has started. The END counterpart is scheduled at the simulation time resultant of the time the BEGIN counterpart has started plus the value presented in the *timebox* attribute. For instance, if a BEGIN_iteration *dummy active state* has started at simulation clock 480 and the *iteration timebox* is configured to 4800, the END_iteration counterpart will be scheduled to 5280 (480+4800). It is important to note that no WBEs nested inside an *iteration*, or *activity*, may start servicing if the current clock plus estimated service time exceeds the current clock plus the *timebox* for its parent *active state*.

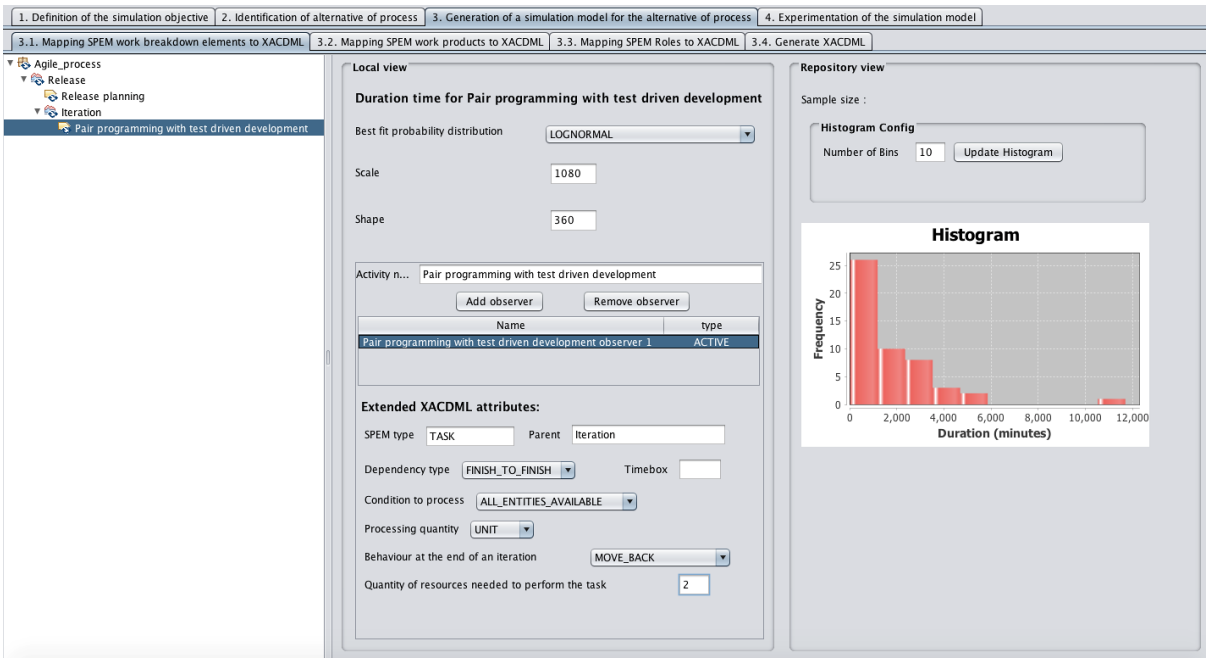

Figure 9: Graphical user interface developed to aid simulation analysts when mapping SPEM to extended XACDML.

In ACD, the default behaviour for an *active state* at its B event is to release *resources* and *entities* back to their respective *dead states*. Our approach adds additional behaviour depending on the type of the SPEM WBEs represented by the *spem_type* attribute. SPEM *activity*, *phase*, and *iteration* represent a period of time in a project with start and finish times. At the B events of the BEGIN and END active states representing these WBEs, we record the times they started and finished, respectively. We also increment a second *counter* that is used to know how many times these SPEM WBEs were performed in the simulated process.

At the B event of the *dummy active state* representing the end of an SPEM *iteration***,** in addition to the logic describe above, one of two other behaviours is possible: (i) move all remaining *entities* in intermediary *dead states* to the initial *dead state* or (ii) keep the current status of the *intermediary dead states*. These options can be configured by the use of the new

XACDML attribute named *iteration_behaviour*. The *initial dead state* of an *iteration* is the incoming one associated with the BEGIN *dummy active state* generated for the SPEM *iteration*. Intermediary *dead states are* all remaining ones presented within the limits of the BEGIN and END *dummy active states* for the *iteration*. As a side effect of executing the logic at the B event of an *iteration*, the *counter* presented in every single *active state* used to determine the number of *temporary entities* produced by the *active state* is set to *zero* and the number of *temporary entities* in the *class of entities* is set to the new quantity in the *initial dead state* of the *iteration*.

We decide to not consider SPEM *phase* as time boxed. Instead, its end is determined when a *class* of *temporary entities* is processed by the END counterpart *active state* (dummy) generated when mapping a SPEM *phase* to extended XACDML. A *milestone* represents a specific moment in time within the project. Within our approach, a *milestone* is reached when a *class* of *temporary entities* is processed. Hence, at a B-event of a *milestone*, we record the time it is reached.

6. EXAMPLE

In this section, we use an example of a software development project that enacts a typical agile method as the process defining the work to be accomplished by a team of developers. We use our tool in this example to demonstrate its applicability.

The process is the same presented in Fig. 1. This example was chosen since it is analogous to a case study that uses real project data presented by Melis et al. [6]. Their work does not use *phases* and *milestones* as in our example. We include these elements in order to cover all SPEM WBE elements during experimentation and verify if the mappings are correct. We argue that this does not affect the results, since *phase* and *milestone* are *dummy active states* and, hence, their executions do not contribute to the total time of the simulation run.

The quantitative model elements were taken from their work either directly or indirectly, i.e. derived from the results presented by the authors. The results presented in their work are compared with ours to validate the generated simulation model. We use the following parameters and values (in parentheses): *number of user stories* (29), *number of developers* (4), *typical release duration* (30 days), *time spent in release planning meeting* (1 day). Melis et al. use a hybrid simulation model with DES and SD in which *user stories* are implemented by incremental sessions of development. In their work, the duration of the development sessions is no longer than four hours. Within a development session, one of the following implementation tasks is performed: *pair programming with test-driven development, debugging*, or *refactoring*. Since we focus on DES, we had to derive the parameters for the duration of a *"Pair programming with test driven development"* task to complete one discrete unit of a *user story* (not a portion of it in an incremental way at the end of each development session). We estimate the duration distribution being *lognormal* with parameters of *mean* = 1900 days and *standard deviation* = 1200 days (the work of Melis et al. describes the effort to produce *user stories* as being lognormal, what helps us during this process of parameters estimation). We made several adjustments before getting to the values used in the example. For simplicity purpose, we assume that the duration to implement a *user story* by *"Pair programming with test driven development"* task includes the time spent in debugging and refactoring activities.

We, with the aid of our tool, performed a number of simulation runs using the input parameters discussed. We inspected the trace reports and log files generated during the simulation runs for verification purposes, attempting to identify logical flaws in the model. The simulation output was compared with those taken from the work of Melis et al. in order to validate our model. The results of the simulation runs are shown in Table II. As can be

seen, our results agree well with the results presented in the work of Melis et al. and the actual data of the reference system. We report here our results using the same number of simulation runs used in their work. The output analysis of the simulation results is out of our scope. Refer to their work for details.

Table II: Comparison between simulation results averaged over 200 runs of Melis et al. [6] and our work. Standard deviations are reported in parentheses.

Output measure	Real data	Simulation (Melis et al.)	Our model
Days	60	61.9 (13.5)	58.1 (5.4)
User stories	29	28.2 (1.6)	28.8 (0.3)
Number of releases	2	2.5 (0.6)	1.9 (0.5)
Iterations per release	3	2.7 (0.3)	2.9 (0.4)

7. DISCUSSIONS

In this work, we propose a model-driven approach to automate the generation of DES models of process models. To achieve our goal, we extended XACDML, the XSLT transformations from XACDML to *Java*, and a DES simulation library. All extensions were necessary in order to cope with process specific elements described in SPEM, including hierarchical structures that are not explicitly defined in ACD, and consequently, XACDML.

Our approach is materialised in a tool that allows generating simulation models and tailoring the generation process to fit analysts' interests. Instead of generating *Java* code from SPEM processes in a single step, we focus our attention in using an intermediate notation (XACDML) for verification purposes. We argue it is much easier to verify XML files (XACDML) representing ACD models than verifying code written in a programming language.

We cover all SPEM WBEs that can be used for the construction of the WBS that represents a SPEM process. In SPEM, these WBEs can be further configured by additional attributes (*isRepeatable*, *hasMultipleOccurrences*, *isOngoing*, *isEventDriven*, and *isOptional)*. We support all these attributes. The *isOptional* attribute is of particular interest since it allows creating conditional flows within a process. It indicates a WBE within the WBS whose inclusion is not mandatory when performing a project, since it depends on some domain specific condition(s). The *simulation analyst* can configure a WBE as optional and indicate the condition that must be satisfied for its execution during mapping of SPEM to XACDML. One such condition could indicate, for example, that a *deployment to production* task is only executed when a minimum number of *user stories* have been implemented. The conditions must be expressed by the use of some pre-defined variables representing the simulation clock, quantity of entities in the incoming dead state, quantity of entities produced by the previous activity, among others.

SPEM can be used to model any process of the systems engineering field [3, 7]. We verified this flexibility in our approach testing other examples not related to software development. In these examples we verified the generated XACDML simulation models looking for inconsistencies. In particular, we verified the correct mapping of all XACDML elements discussed in this paper, including conditional flows (testing several conditional branches). Likewise, we checked for each example, the XACDML model assumption that the specifications of the destinations of *entities* and *resources* were all made in an unequivocal manner. No problem was found during the verifications. We also conducted several runs with additional examples, verifying the simulation results and log files generated for each run in order to identify possible logical flaws. No error was found, what makes us to believe that our

approach with tool support is feasible and has the potential to support any process modelled with the SPEM specification.

The closest work to our current research [2] proposes an approach to derive a hybrid simulation model (DES and SD) from a SPEM 1.1 process model. We consider the hybrid mechanism as an advantage when contrasted to our work. By using their approach, the intervention of the stakeholders in developing the simulation model is reduced, but not as much when compared to our work. Since their work was made with an earlier specification of SPEM (version 1.1), it could not benefit from the main improvement in SPEM 2.0, that is, the clear separation of method content and its application in processes [3] – page 11. As a consequence of the lack of concern separation, their approach requires modelling using UML *use cases* and *activity diagrams* to model process work sequence. The need to modelling imposes some burden in order to transform the models. Our work is SPEM 2.0 compliant and hence, benefits from this separation of concerns, not requiring this additional and manual step of UML modelling. We only demand a *simulation analyst* with ACD knowledge to set the parameters during mapping of SPEM to XACDML.

We consider that our approach is effective and efficient. It is effective since it mitigates the effort needed to build simulation models for processes by automating the generation. It is efficient, since the time spent in conducting experiments is limited to the time a *simulation analyst* needs to configure the mappings of SPEM to XACDML.

Basic knowledge of ACD notations and XACDML is needed in order to use our approach. This is one drawback of our work. Although the mapping algorithms presented allow the automation of the translation process, a *simulation analysts* is needed to configure the mappings with appropriate parameters. For instance, the *simulation analyst* must decide whether it is necessary or not to include an XACDML *generate active state* during the mapping process.

MDD provides new capabilities for the development of simulation models, but during development of our approach, we found difficult adopting MDD, since it is restricted to particular modelling languages and requires a steep learning curve with respect to its principles and methods.

The decomposition of *work products* is an important feature of SPEM processes where one input *work product* of a WBE is decomposed into one or more output *work products* of the same or different types. XACDML should be extended to support this feature. The work of Hirata and Paul [14] can be used with this purpose. We intend to embrace this feature within our approach in a near future.

8. CONCLUSION

In this work, we proposed a model-driven approach to automate the generation of DES models of process models. We strived to reduce the need of human intervention. There is no need of knowledge of some programming language in order to use our approach.

The results obtained and presented in this paper indicate that transforming SPEM models to XACDML models is a feasible choice. However, we acknowledge that the approach still needs to be validated and improved with more examples and case studies in industry. We are interested in dynamic definition of engineering processes during enactment. As a future work, we will extend it to enable the analysis of the impact of process changes during project evolution. The idea is to support the decision-making of what process changes to conduct in an ongoing project. The decision is taken by evaluating several process alternatives with the use of simulation after evaluating the impact of these alternatives on output variables of interest in the project.

ACKNOWLEDGEMENT

CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) supported this work.

The work of the second author was supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico under the grant number Universal 01/2016 403921/2016-3 and the grant number PQ 303666/2015-3.

REFERENCES

[1] Ali, N. B.; Petersen, K.; Wohlin, C. (2014). A systematic literature review on the industrial use of software process simulation, *Journal of Systems and Software*, Vol. 97, 65-85, doi:10.1016/j.jss.2014.06.059

[2] Park, S.; Choi, K.; Yoon, K.; Bae, D.-H. (2007). Deriving software process simulation model from SPEM-based software process model, *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, 382-389

[3] Object Management Group. Software & Systems Process Engineering Meta-Model Specification, from *http://www.omg.org/spec/SPEM/*, accessed on 19-06-2017

[4] Pidd, M. (2004). *Computer Simulation in Management Science*, 5th edition, Wiley, New York

[5] Gil, J. N.; Hirata, C. M. (2003). XACDML – Extensible ACD Markup Language, *Proceedings of the 36th Annual Symposium on Simulation*, 343-350

[6] Melis, M.; Turnu, I.; Cau, A.; Concas, G. (2006). Evaluating the impact of test-first programming and pair programming through software process simulation, *Software Process: Improvement and Practice*, Vol. 11, No. 4, 345-360, doi:10.1002/spip.286

[7] Ruiz-Rube, I.; Dodero, J. M.; Palomo-Duarte, M.; Ruiz, M.; Gawn, D. (2013). Uses and applications of software & systems process engineering meta-model process models. A systematic mapping study, *Journal of Software: Evolution and Process*, Vol. 25, No. 9, 999-1025, doi:10.1002/smr.1594

[8] Çetinkaya, D.; Verbraeck, A.; Seck, M. D. (2015). Model continuity in discrete event simulation: A framework for model-driven development of simulation models, *ACM Transactions on Modeling and Computer Simulation*, Vol. 25, No. 3, Paper 17, 24 pages, doi:10.1145/2699714

[9] Hollmann, D. A.; Cristiá, M.; Frydman, C. (2015). CML-DEVS: A specification language for DEVS conceptual models, *Simulation Modelling Practice and Theory*, Vol. 57, 100-117, doi:10.1016/j.simpat.2015.06.007

[10] Kim, S.; Sarjoughian, H. S.; Elamvazhuthi, V. (2009). DEVS-suite: A simulator supporting visual experimentation design and behavior monitoring, *Proceedings of the Spring Simulation Multiconference (SpringSim 2009)*, Paper 161, 7 pages

[11] Bergero, F.; Kofman, E. (2010). PowerDEVS: a tool for hybrid system modeling and real-time simulation, *Simulation*, Vol. 87, No. 1-2, 113-132, doi:10.1177/0037549710368029

[12] Arias, R.; Hirata, C. M. (2011). Mapping of software model to simulation model for performance requirement specification, *Proceedings of the 44th Annual Simulation Symposium*, 142-150

[13] Araújo Filho, W. L.; Hirata, C. M. (2004). Translating activity cycle diagrams to Java simulation programs, *Proceedings of the 37th IEEE Annual Symposium on Simulation*, 157-164

[14] Hirata, C. M.; Paul, R. J. (1996). Object-oriented programming architecture for simulation modelling, *International Journal in Computer Simulation*, Vol. 6, No. 2, 269-287