

A Formal Description of the ITIL Change Management Process Using Abstract State Machines

Jan Kubovy (Author)
Institute of Applied Knowledge Processing
Johannes Kepler University
Linz, Austria
jkubovy@faw.jku.at

Verena Geist, Felix Kossak (Author)
Software Competence Center Hagenberg
(SCCH)
Hagenberg, Austria
verena.geist@scch.at, felix.kossak@scch.at

Abstract—We suggest formalising Information Technology Infrastructure Library (ITIL)[1] processes using the Business Process Model and Notation (BPMN)[2] and the Abstract State Machine (ASM) method. We describe the benefits of our approach as well as the necessary prerequisites. We argue that such a formalisation will lead to a clearer understanding of the process and a reduction of ambiguity.

Keywords—asm; bpmn; formalizing; itil;

I. INTRODUCTION

The best practices described by ITIL provide a good start to model a company's IT processes. However, when concrete, custom processes are defined for a particular company, an unambiguous modelling of those processes is required in order for them to be reliably implemented, supervised, maintained, and changed when required.

We propose modelling IT processes with as much rigour and formality as required, depending on the given circumstances such as complexity. As a first step towards enhanced rigour as compared with the ITIL templates, we propose to model specific processes with the Business Process Model and Notation (BPMN) (see [2]).

However, also BPMN does not have a *formally* defined semantics, leading to possible ambiguities and differences in the behaviour of specific tools. In order to overcome these problems, if required, we propose further to formally define the semantics of BPMN constructs using the ASM method (see e.g. [3]). We do not see this as a *replacement* for BPMN, however, because first, BPMN is a widely used and basically well-understood standard, and secondly, ASMs cannot be a substitute for the intuitive, graphical representation of BPMN.

We demonstrate this three-step approach (ITIL templates - BPMN - ASMs) with a small example taken from ITIL, concerning change management. Thereby we draw from our ongoing work in which we define an unambiguous semantics for the BPMN 2.0 standard.

II. THE SEMANTICS OF BPMN

We cannot repeat the semantics of BPMN in this place and refer the reader to the standard in [2] (see also e.g. [4]

for a formalisation of a beta version of BPMN 2.0 using ASMs). We will present an example of a small part of the change management process from ITIL to show the formalisation process and to discuss the advantages of such a representation.

We will make use of ASMs (see e.g. [5], [3]) to model semantics. We trust that ASMs are easily readable, but we will briefly explain a few constructs to clarify their meaning for those who are not as yet familiar with the method.

A. A Few Notes on Abstract State Machines (ASMs)

ASMs can be seen as “a rather intuitive form of abstract pseudo-code”, though based on a precise but minimal mathematical theory of algorithms, but also as “Virtual Machine programs working on abstract data” ([3, p. 2, 5]).

The main constructs we will use are *rules* and *derived functions*. A *rule* describes a *state transition* of the machine (automaton). A *state* is determined by particular values of arbitrary data structures. Those *data structures* are described by *functions*, e.g. `tokenType`. If a *function* is given specific parameters, we speak of a *location* - e.g. `tokenType(t)`, where *t* is a particular token. If the value of at least one *location* is changed (*updated*), the *state* of the machine changes. Such *updates* are described by *rules*, of which several may execute in parallel.

Whether a *rule* fires in a particular *state* or not is determined by a *guard* or condition in the form of an `if ... then ... statement`. This *guard* will typically query the value of one or more functions. If the query is more complex, or we want to leave it abstract for the time being, we can use *derived functions*, which combine the values of other functions through an arbitrary statement. An example is `controlCondition(flowNode)`, which determines whether there are enough tokens on the incoming *sequence flows* of a given *flow node* for it to fire. *Derived functions* have no effect on the *state* of the machine.

For clarity, we will mark *rule* definitions and *derived functions* with the keywords `rule` and `derived`, respectively.

Monitored functions are controlled by the environment; they can e.g. be used to model user input.

Nesting of expressions will be expressed by indentation only.

Refinement of abstract *rules* will often be indicated by the keyword *where*, after which *subrules* and *derived functions* previously left abstract will be defined.

A *choose* statement models an *arbitrary* choice of an element of a given set. We trust that all other keywords and constructs are easy to grasp.

B. General Semantics of BPMN Models

[4] model the semantics of a BPMN flow node as shown in Figure 1.

```

1 rule WorkflowTransition(flowNode) =
2   if eventCondition(flowNode) and
3     controlCondition(flowNode) and
4     dataCondition(flowNode) and
5     resourceCondition(flowNode) then
6     EventOperation(flowNode)
7     ControlOperation(flowNode)
8     DataOperation(flowNode)
9     ResourceOperation(flowNode)

```

Figure 1. Workflow Transition

That is, in order for a *flow node* (e.g. an *activity* or a *gateway*) to “fire” and thereby to do something, it may have to wait for an event to happen, for a *token* to arrive (controlCondition), for a data-based condition to be true or for data to be available in the first place, and/or for resources to be available. Then it may itself fire events, pass on or create *tokens*, manipulate data, and occupy and/or release resources. (Note that we use the convention to capitalise the operations, which refer to (abstract) *subrules*, while we write the conditions – (abstract) *derived functions* – with small first letters.)

In this paper, we only consider *sequence flows*, i.e. controlCondition and ControlOperation.

C. The Semantics of Gateways

We will only skirt the semantics of *gateways* as much as we need it for the purposes of this paper.

We differentiate between *splitting gateways* and *merging gateways*. According to the BPMN standard, one *gateway* can be both splitting and merging, but with the exception of *complex gateways*, which we do not treat here, a *gateway* which is both splitting and merging can always be simulated by separate splitting and merging *gateways*. That way the semantics is easier to understand.

We start with *splitting gateways* (see Figure 2).

Based on the *where* statement, we say that WorkflowCondition from Figure 1 is refined by specifying controlCondition as well as ControlOperation in Figure 2 and Figure 4.

Parallel gateways, *exclusive gateways* and *inclusive gateways* only differ in the way they choose outgoing *sequence flows* for passing on *tokens*, i.e. in the definition

```

1 rule SplitTransition(gateway) =
2   choose incomingSequenceFlow in
3     incomingSequenceFlows(gateway) do
4     // assert: there is only one element
5     // in incomingSequenceFlows(gateway),
6     // thus incomingSequenceFlow is chosen
7     // deterministically
8   WorkflowTransition(gateway) where
9     derived controlCondition(gateway) =
10      tokens(incomingSequenceFlow) != {}
11   rule ControlOperation(gateway) =
12     let chosenSequenceFlows =
13       selectOutgoingSequenceFlows(
14         gateway) in
15     choose token in
16       tokens(incomingSequenceFlow) do
17       Consume(incomingSequenceFlow,
18         token)
19     forall outgoingSequenceFlow in
20       chosenSequenceFlows(
21         gateway) do
22       Produce(outgoingSequenceFlow,
23         NewToken(tokenType(token)))

```

Figure 2. SplitTransition

of selectOutgoingSequenceFlows. We only need *exclusive gateways* for this paper, where we take firingCondition to be a *monitored function* (as we do not treat data-based conditions in this paper). A set with only one element is returned (see Figure 3).

```

1 rule ExclusiveSplitTransition(gateway) =
2   WorkflowTransitionSplit where
3   derived selectSequenceFlows(gateway) =
4     { choose sequenceFlow in
5       outgoingSequenceFlows(gateway) with
6       firingCondition(sequenceFlow) = true
7     }

```

Figure 3. ExclusiveSplitTransition

```

1 rule ExclusiveMergeTransition(gateway) =
2   choose incomingSequenceFlow
3   in incomingSequenceFlows(gateway)
4   with tokens(incomingSequenceFlow) != {}
5   do
6   choose token in
7     tokens(incomingSequenceFlow) do
8     WorkflowTransition(gateway) where
9     derived controlCondition(gateway) =
10      token != undef
11   rule ControlOperation(gateway) =
12     choose outgoingSequenceFlow
13     in outgoingSequenceFlows(
14       gateway) do
15     Consume(incomingSequenceFlow,
16       token)
17     Produce(outgoingSequenceFlow,
18       NewToken(tokenType(token)))

```

Figure 4. ExclusiveMergeTransition

D. The Semantics of Activities

For the semantics of BPMN activities (with our present simplifications), we can use `WorkflowTransition` with the constraints that

- 1) there is exactly one incoming and one outgoing *sequence flow* and
- 2) when the *activity* has been performed, a *token* is passed on to the outgoing *sequence flow*; however, here we abstract from the act and duration of the actual performance (see Figure 5).

```

1 rule ActivityTransition (flowNode) =
2   let enablingTokenInstances =
3     enablingTokens (flowNode) in
4     WorkflowTransition (flowNode) where
5   derived controlCondition (flowNode) =
6     enablingTokenInstances != {}
7   rule ControlOperation (flowNode) =
8     choose outgoingSequenceFlow ,
9       incomingSequenceFlow in
10      outgoingSequenceFlows (gateway) do
11        Consume (incomingSequenceFlow , token)
12        // actual performance
13        Produce (outgoingSequenceFlow ,
14          NewToken (tokenType (token))

```

Figure 5. ActivityTransition

For the purpose of this paper, we will not further refine activities to e.g. tasks or sub-processes, although for real application, this will have to be done.

III. PROCESS DESCRIPTION

We will use only one part of the whole ITIL change management process[6] to clarify the concept for the reader. The graphical representation of that part modelled using BPMN can be seen in Figure 6.

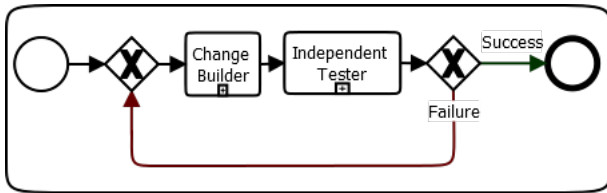


Figure 6. Part of the change management process

A. Transformation to ASM code

For the representation by ASMs we first need to define the *flow nodes* in the process and their types (see Figure 7).

The next task is to define the *sequence flows* and connect the defined *flow nodes* (see Figure 8).

We will use the following naming convention: every *flow node* has a number placed in the comment in Figure 7. The name of a *sequence flow* will start with a capital “A” (as in “Arc”) followed by two numbers, where the first number is

```

1 nodes := {
2   "Start",           // node 1
3   "Merge",           // node 2
4   "ChangeBuilder",   // node 3
5   "IndependentTester", // node 4
6   "Split",           // node 5
7   "End"              // node 6
8 }
9
10 nodeType("Start") := StartEvent
11 nodeType("Merge") := ExclusiveMergeGateway
12 nodeType("ChangeBuilder") := SubProcess
13 nodeType("IndependentTester") := SubProcess
14 nodeType("Split") := ExclusiveSplitGateway
15 nodeType("End") := EndEvent

```

Figure 7. Flow Nodes

the number of the source *flow node* and the second number is the number of the target *flow node*.

```

1 sequenceFlows :=
2   { "A12", "A23", "A34", "A45", "A52", "A56" }
3
4 source("A12") := node("Start")
5 target("A12") := node("Merge")
6 source("A23") := node("Merge")
7 target("A23") := node("ChangeBuilder")
8 source("A34") := node("ChangeBuilder")
9 target("A34") := node("IndependentTester")
10 source("A45") := node("IndependentTester")
11 target("A45") := node("Split")
12 source("A52") := node("Split")
13 target("A52") := node("Merge")
14 source("A56") := node("Split")
15 target("A56") := node("End")

```

Figure 8. Sequence Flows

B. Running Environment

To be able to run the defined process, a *main rule* (see Figure 9) is defined on top of the `WorkflowTransition`. This rule continuously checks if there are some *top level processes* and calls the rule `WorkflowTransitionInterpreter` (see Figure 10) for each of them. For simplicity, we left out some implementation parts of the `WorkflowTransitionInterpreter`.

```

1 main rule RunTopLevelProcesses =
2   if topLevelProcesses != undef then
3     forall process in topLevelProcesses do
4       WorkflowTransitionInterpreter (process)

```

Figure 9. Main Rule

The `WorkflowTransitionInterpreter` checks for all active *process* instances which became inactive and removes them from the list of active *process* instances. It triggers all *start events* of all new *process* instances requested by the environment and, for each *flow node*

in a *process*, it calls the rule `WorkflowTransition` described in Figure 1.

```

1 rule WorkflowTransitionInterpreter(process) =
2   if abortedByEnvironment = false then
3     forall node in flowNodes(process) do
4       WorkflowTransition(node)

```

Figure 10. Workflow Transition Interpreter

IV. BENEFITS

The way in which ITIL processes are depicted does not appear to be standardised, nor does it appear to have a clear, unambiguous semantics. According to the experience of the authors, once it gets into details, the precise meaning of such diagrams is very often not as clear as it may appear at first sight.

In order to improve the modelling of ITIL processes in this light, we first replace the given flowcharts by BPMN diagrams. The BPMN core elements, which are similar to those of UML Activity Diagrams (and similar to those used by ITIL), are already well understood by many people, and many tools exist for modelling and simulating on the basis of BPMN. At the same time, the intended semantics of BPMN diagrams is extensively documented in the standard.

However, the BPMN standard defines the semantics in natural language, and in different, overlapping chapters apparently written by different authors. It turns out that there are cases where questions about the behaviour of the system are left open – that is, the BPMN 2.0 standard remains ambiguous, and, in certain cases, even exhibits contradictions. (Several of these ambiguities have been pinpointed e.g. by [4], and more have been detected by us in yet unpublished work.)

Consequently, in a second step, we define the semantics of BPMN constructs formally using Abstract State Machines (ASMs). Building on previous work by [7] and [4], we have been building a flexible ground model of the BPMN 2.0 standard which unambiguously describes the intended behaviour of each BPMN element. This ground model can easily be tweaked and adapted, e.g. to an update of the standard.

This way, IT processes not only can gain a clear, *unambiguous meaning*, but also a wide range of *tool support* becomes available, by which those processes cannot only be documented and maintained, but can also be *simulated* and thereby be validated in a sandbox. Moreover, using the formally defined semantics, also certain properties (like deadlock freedom or liveness) can be checked, either manually, or automatically (e.g. using model checkers).

But maybe most importantly, the process of formalising the intended behaviour alone already exhibits problems and weaknesses of initial sketches, which are not as easily detected in informal descriptions and diagrams without formal

semantics. (This benefit of formal methods has long been known, see e.g. [8].) This is further supported by the ability to simulate the models.

V. CONCLUSION

We believe that using a unified, unambiguous and clear modelling notation along with formal semantics will lead to a steeper learning curve and greater understanding of ITIL processes and of particular processes inside a company.

Since there are some ambiguities in widely-used flowchart notations and even in the current Business Process Model and Notation (BPMN) 2.0 specification, we are using an own specification which is based on BPMN 2.0 but where we tackle the ambiguous and unclear parts by means of Abstract State Machines (ASMs).

Following the three-step approach proposed in this paper, the user will first choose an ITIL template and then draw a BPMN diagram (or use an existing one if it already exists). This graphical representation of a process can be easily converted into a formal yet intelligible ASM code. At this level the user can define more detailed information about the process, constraints, and dependencies needed to deterministically run the code.

VI. FUTURE WORK

One possible application could be a creation of a simple and abstract framework by which an abstract description of a change management process can be acquired and refined for the specific needs of the company using it. This would allow the company using such a framework to lower their costs for specifying a new change management process and to assess the impact of changes. Thereby also risks such as unauthorised changes can be lowered.

Also other processes beside ITIL processes can be described using the proposed method to ensure that they will not break after changes. The refinement method described in [3] allows one to make such changes and to improve the process without breaking it.

Furthermore, since the conversion process between BPMN diagrams and ASM code should be a deterministic, easy, and simple routine job (except from purely graphical information), such conversions should be automated in the future.

ACKNOWLEDGMENT

This work was supported in part by the Austrian Science Fund (FWF) under grant no. TRP 223-N23.

The project *Vertical Model Integration (VMI)* is supported within the program “Regionale Wettbewerbsfähigkeit OÖ 2007-2013” by the European Fund for Regional Development as well as the State of Upper Austria.

REFERENCES

- [1] A. Cartledge, M. Lillycrop, A. Hanna, C. Rudd, I. Macfarlane, J. Windebank, and S. Rance, *An Introductory Overview of ITIL V3*, ser. The IT Infrastructure Library. ITSMF, 2009.
- [2] OMG, *Business Process Model and Notation (BPMN) 2.0*, www.omg.org/spec/BPMN/2.0, Object Management Group Std., 2011, accessed August 2011.
- [3] E. Börger and R. Stärk, *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
- [4] E. Börger and O. Sörensen, “BPMN core modeling concepts: Inheritance-based execution semantics,” in *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*, D. Embley and B. Thalheim, Eds. Springer Verlag, 2011.
- [5] Y. Gurevich, “Sequential abstract state machines capture sequential algorithms,” *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 77–111, 2000.
- [6] Office Of Government Commerce, *ITIL V3 Service Transition*. The Stationery Office, 2007.
- [7] E. Börger and B. Thalheim, “A method for verifiable and validatable business process modeling,” *Advances in Software Engineering, LNCS*, vol. 5316, pp. 59–115, 2008.
- [8] B. Meyer, “On formalism in specifications,” *IEEE Software*, vol. 2, pp. 6–26, 1985.