Verification of ArchiMate Behavioral Elements by Model Checking

Piotr Szwed

Verification of ArchiMate behavioral elements by model checking

Piotr Szwed

AGH University of Science and Technology
pszwed@agh.edu.pl

Abstract. In this paper we investigate the problem of verification of business processes specified with ArchiMate language. The proposed solution employs model checking techniques. As a verification platform the state of the art symbolic model checker NuSMV is used. We describe a method of fully automated translation of behavioral elements embedded in ArchiMate models into a representation in NuSMV language, which is then submitted to verification with respect to requirements expressed in CTL. The requirements specification can be entered by user, but we also propose to derive some of them automatically, based on analysis of control flows within business processes. The solution was implemented as a plugin to Archi, a popular ArchiMate modeling tool. Application of the method is presented on an example of a small business process.

Keywords: formal verification, model checking, business process, ArchiMate, NuSMV

1 Introduction

In this paper we investigate an application of model checking techniques to automated verification of behavioral description embedded within ArchiMate models. ArchiMate is a lightweight language providing a uniform representation of enterprise architecture [**?**]. The language comprises elements of various types, however, constructs allowing to model behavior can be found only in the *Business* layer. They include events, processes (also understood as activities), interactions, collaborations and several types of junctions. Therefore, the verification of ArchiMate behavioral elements falls into a wide domain of business process verification [**?**].

Business models can comprise a large number of processes. For clarity reasons they are often depicted in form of several *views*, that cover only selected parts of the model. In consequence, behaviors embedded in the model are distributed among the views, what often makes them difficult to track. Although modeling tools offer support for local syntax checking, e.g. correct use of links between elements of the graphical language, some structural errors remain undetected, especially those resulting from incorrect use of synchronization mechanisms [**?**]. Partial analysis of model behavior can be performed by simulation techniques,

however, only application of formal methods can give unequivocal answer that the verified model exhibits desired properties.

Formal system verification can by done either by deductive reasoning or model checking [**?**]. Deductive reasoning consists in formulating theorems specifying desired system properties and proving or falsifying them using manual or automated techniques. The advantage of deductive reasoning methods is their ability to verify systems with infinite domains (number of states). However, they give very little information on causes, if the verified property does not hold.

Model checking allows to verify a concurrent system modeled as a finite state transition graph against a set of specifications expressed in a propositional temporal logic. It employs efficient internal representations and quick search procedures to determine automatically, whether the specifications are satisfied along the computational paths. Moreover, if a specification is not met, the procedure delivers a counterexample that can be used to analyze the source of the error. The main problem faced by model checking is the state explosion [**?**]. At the very beginning only small examples could have been processed. A significant progress in this technique was achieved with application of ordered binary decision diagrams (OBBD) [**?**] allowing to model systems consisting of millions of states and transitions.

Although formal tools reached state of the art, they are not commonly used in engineering practice. According to Huuck [**?**] three factors decide on successful application of formal tools: they should be simple to use, the time spent on model preparation and verification should be comparable with other user activities, and, finally, a tool should provide a real value, i.e. deliver information that was previously not available.

The goal of our work was to develop a software tool that fully automatically translates behavioral elements of a business model expressed in ArchiMate language to a corresponding finite-state graph required by a model checker. We were also attracted by an idea of deriving automatically requirements specifications based on control flows within business processes.

As a verification platform the state of the art symbolic model checker NuSMV [**?**] is used. NuSMV allows to enter a model being a set of communicating finite state machines (FSM) and automatically check its properties specified as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL) formulas. For a given temporal logic formula $\mathcal{F}$, NuSMV provides the answer that $\mathcal{F}$ is satisfied by the model or it delivers a counterexample falsifying it.

The concept of verification system is presented in Fig. 1. The business model is defined within Archi [**?**], a well known ArchiMate modeling tool. We have developed an Archi plugin that extracts a subgraph of ArchiMate behavioral elements and transforms it into NuSMV model descriptions.

As a specification language CTL is used. Basically, a specification of system properties is entered by a user. This is a manual task, that requires a certain insight into the business process, as well a knowledge of mapping of its elements onto NuSMV model. However, a part of the specification is generated automatically by an analysis of the process structure.

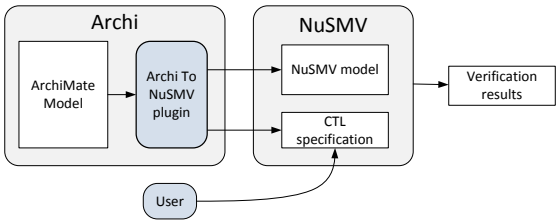


Fig. 1. The concept of the verification system

The paper is organized as follows: next Section 2 discusses various approaches to data verification of business models. It is followed by Section 3, which presents ArchiMate language. In Section 4 the translation procedure is described. Application of the presented approach to a business process verification is presented in Section 5. Section 6 provides concluding remarks.

2 Related works

Application of formal methods to verification of business processes was surveyed by Morimoto [**?**]. Author distinguished three prevalent approaches: based on automata, Petri nets and process algebras. The first approach consists in translating the process description into a set of communicating automata (state machines) and performing model checking with such tools, as SPIN or UPPAAL In analysis of Petri net models basically simulation techniques are used, especially in case of more expressive colored Petri nets.

Model checking has an established position in verification of business processes. I was applied in [**?**] to BPMN models extended with temporal and resource constraints. In [**?**] verification of of e-business processes was achieved by translation to CSP language and checking refinement between two specifications. In [**?**] authors implemented a system that translates BPEL specification into NuSMV language and then allows to check properties defined as CTL formulas. Three types of correctness properties were analyzed: invariants, properties of final states and temporal relations between activities. The first two can be classified as *safeness*, the last as the *liveness* property. Similarly, in work by Fu et al. [**?**] CTL was applied to the verification of e-services and workflows with both bounded and unbounded number of process instances. Work [**?**] discusses verification of data-centric business processes. The correctness problem was expressed in the LTL-FO, an extension to the Linear Temporal Logic, in which propositions were replaced by First Order statements about data objects.

In our previous works [**?**,**?**] we proposed a method for verification of ArchiMate behavioral specifications based on deductive reasoning. The described approach consisted in transforming ArchiMate model into a set of LTL formulas, then extending it with formulas defining desired system properties and formally proving them using semantic tableaux method.

NuSMV [?] is a state of the art model checker that has been succesfully used for various verification tasks including formal protocol analysis [?], verification of requirements specification [?] or planning tasks [?]. The package uses a special language (named also NuSMV) to define the verified model as a set of linked finite state machines, as well as its specification in form of temporal logic formulas. The model submitted to the verification tool must be manually coded in NuSMV language or generated from another language amenable to state transition system, e.g a state charts [?] or reachability graphs of Petri nets [?].

3 Archimate

ArchiMate [?] is a contemporary, open and independent language intended for description of enterprise architectures. It comprises five main modeling layers shortly characterized below. The *Business* layer includes business processes and objects, functions, events, roles and services. The *Application* layer contains components, interfaces, application services and data objects. The *Technology* layer gathers such elements as artifacts, nodes, software, devices, communication channels and networks. Elements of the *Motivation* layer allow to express business drivers, goals, requirements and principles. Finally, *Implementation&Migration* layer contains such elements, as work package, deliverable and gap.

ArchiMate allows to present an architecture in the form of views which, depending on the needs, can include only items in one layer or can show vertical relations between layers, e.g.: a relationship between a business process and a function of the component software.

ArchiMate was built in opposition to UML [?], which can be seen as a collection of unrelated diagrams, and Business Process Modeling Notation BPMN [?] which covers mainly behavioral aspect of enterprise architecture. The definition of a language has been accompanied by an assumption, that in order to build an expressive business model, it is necessary to use the relationships between completely different areas, starting from business motivation to business processes, services and infrastructure.

Archimate provides a small set of constructs that can be used to model behavior. It includes *Business Processes*, *Functions*, *Interactions*, *Events* and various connectors (*Junctions*), which can be attributed with a logical operator specifying, how inputs should be combined or output produced. According to language specification casual or temporal relationships between behavioral elements are expressed with use of *triggering* relation. On the other hand, Archimate models frequently use *composition* and *aggregation* relations, e.g. to show that a process is built from smaller behavioral elements (subprocesses or functions).

Although the set of behavioral elements seems to be very limited when compared with BPMN [?], after adopting a certain modeling convention its expressiveness can be similar [?]. An advantage of the language is that in allows to comprise in a single model a broad context of business processes including roles,

services, processed business objects and elements of lower layers responsible for implementation and deployment.

4 Model generation

This section discusses language patterns that can be used to model ArchiMate elements in NuSMV, as well as details of the translation procedure.

4.1 ArchiMate model

The internal structure of an Archimate model constitutes a graph of nodes linked by directed edges. Both nodes and edges are attributed with information indicating a type of element or relation. Generating NuSMV code describing behavioral aspects of Archimate model we focus on components of the *Business layer*: processes (interactions, functions), events and various junctions.

It should be noted that Archimate behavioral constructs have no precisely defined semantics. In fact, translation from Archimate specification to NuSMV assigns a semantics, which, although arbitrarily selected, follows a certain intuition, e.g. how to interpret an activity or an event.

Definition 1 (Archimate model). *ArchiMate model AM is a tuple* $\langle V, E, C, R, v, e\rangle$*, where* V *is a set of vertices,* $E \subset V \times V$ *is a set of edges,* C *is a set of ArchiMate element types,* R *is a set of relations,* $vt\colon V \to C$ *is a function that assigns element types to graph vertices* $et\colon E \to R$ *assigns relation types to edges.*

As we focus on business layer elements that are used to specify behavior, it is assumed that $C = \{$*Process*, *Function*, *Interaction*, *Event*,*Junction*, *AndJunction*, *OrJunction*, *Other*$\}$ and $R = \{$*triggering*,*association*,*composition*, *other*$\}$.

4.2 NuSMV model

The basic structural unit in NuSMV language is *module* understood as a set of variables and statements that assign to them initial values and define a transition relation. Depending on the module definition, we may distinguish input variables corresponding to stimuli, internal state variables and output variables (actions).

Definition of a module introduces a new type that can be instantiated. Hence, it is possible to declare a variable of a module type and bind it during declaration resembling a constructor call to a number of input variables.Subsequent variables definitions may reference outputs of other modules instances as their inputs. This allows to define a system of communicating state machines of desired complexity, which propagates input stimuli to its components causing subsequent state changes and generation of output signals. Typically, the model integration is achieved within the special *main* module, however, it can be distributed among lower level modules, which are referenced from *main*.

After an analysis of components used to describe ArchiMate processes the following basic modules were identified and implemented:

- $atomicProcess_n$: n-ary atomic process has exactly one input, one primary output and n additional outputs, which can be activated if one of n exceptions occurs. The exception should be modeled in ArchiMate as an event linked with the process by an association relation.
- *event*: has only one input and one output (a boolean flag). Multiple recipients may use this flag as trigger.
- $andFork$: used to model AndJunction in Archmate. The module construction is analogous to event.
- $andJoin_n$: n-ary andJoin produces output signal, if all n inputs are set to TRUE.
- $xorFork_n$: n-ary xor-fork have one input and n outputs. Upon module activation, only one from outputs will be triggered.
- $xorJoin_n$: n-ary xor-join has n inputs and sets the output flag if any of them is set. Moreover it tracks the number of inputs, e.g. if two from n inputs are activated, the output flag will be set twice.

Fig. 2 shows the state diagram of the module `atomicProcess1`. The number 1 indicates the number of additional outputs activated as a result of exception occurrence. The process is activated by the input signal *trigger*. Upon signal arrival it makes the state transition from *idle* to *started*. Then a choice can be made between the states *finished* and *interrrupted1*. Synchronously, the corresponding output variable is set: either *outflag* or *exccptflag1* to *TRUE*. The output variable, whichever is set, will be cleared during the transition to *idle* state.

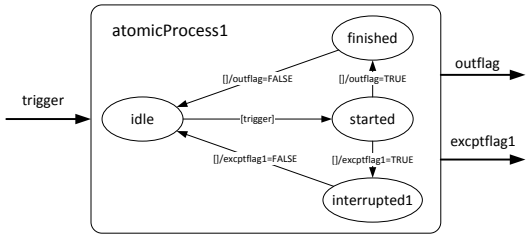


Fig. 2. State machine modeling an atomic process

The NuSMV code for the module is given in Fig. 3. It should be mentioned, that n exceptional outputs, we generate module `atomicProcess_n` with states *interrupted1*,..., *interrruptedn* and n output flags *exceptflag1*,..., *exceptflagn*.

4.3 Generation procedure

The generation procedure consists of the following stages:

1. *Refactoring.* With relation to the numbers of inputs and outputs, it is expected that elements fall into one of two classes: $1 : m$ (one input and m

```
MODULE atomicProcess1(trigger)
VAR
   state : {idle,started,finished,interrupted1};
   outflag : boolean;
   excptflag1 : boolean;
ASSIGN
   init(state) := idle;
   next(state) :=
      case
         state = idle & trigger: {started};
         state = started : {finished,interrupted1};
         state = finished & !outflag : idle;
         state = interrupted1 & !excptflag1 : idle;
         TRUE : state;
      esac;
   init(outflag) := FALSE;
   next(outflag) :=
      case
         state = finished : TRUE;
         state = idle : FALSE;
         TRUE : outflag;
      esac;
   init(excptflag1) := FALSE;
   next(excptflag1) :=
      case
         state = interrupted1 : TRUE;
         state = idle : FALSE;
         TRUE : excptflag1;
      esac;
SPEC
   AG (trigger = TRUE -> AF (outflag = TRUE | excptflag1 = TRUE))
```

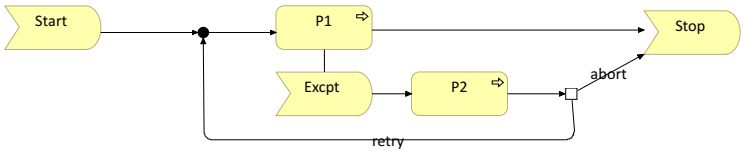
Fig. 3. NuSMV code of the module `atomicProcess1` (the number 1 indicates number of exceptional outputs)

outputs) or $n : 1$ (m inputs and one output). Hence elements with the arity $n : m$ are replaced by two two elements: the first is an appropriate xorJoin or andJoin of arity $n : 1$. The second is an atomic process, event or fork of arity $1 : n$.

2. *Assigning representation.* For each element, based on its type and numbers of inputs/outputs, an appropriate NuSMV module type is selected and configured. Only required modules are generated. E.g. if the specification uses only processes with one and three exceptional outputs, only modules defining `atomicProcess1()` and `atomicProcess3()` will be generated.
3. *Main module generation.* This step comprises declaration of variables and linking them. For roots (modules without inputs) appropriate initial variables and transitions are added as well.
4. *Specification generation.* The implemented procedure analyses the graph of elements and generates CTL specifications. See Section 4.5.

4.4 Small example

We will discuss the effects of the generation procedure on a small process example presented in Fig. 4. The whole process is activated upon occurrence of the event *Start*. Then the subprocess *P1* is launched. If *P1* terminates correctly, the event *Stop* is produced. However, *P1* execution can be interrupted by the event *Excpt*, which triggers the subprocess *P2*. After finishing *P2* a decision is made, whether the whole process should terminate (*abort*) or *P1* should be launched once again (*retry*).




```
MODULE main
VAR
   Excpt : event(P1.excptflag1);
   P1 : atomicProcess1(Junction.output);
   P2 : atomicProcess(Excpt.outflag);
   Start : event(Start_trigger);
   Stop : event(Junction_Before_Stop.output);
   Junction : orJoin(Start.outflag,Or_Junction.outflag2);
   Or_Junction : orFork(P2.outflag);
   Junction_Before_Stop : orJoin(Or_Junction.outflag1,P1.outflag);
   Start_trigger : boolean;
ASSIGN
   init(Start_trigger) := FALSE;
   next(Start_trigger) := {FALSE,TRUE};
```

Fig. 4. Sample ArchiMate specification and corresponding NuSMV main module code

The generated NuSMV code for the *main* module is presented in Fig. 4 below the ArchiMate diagram. It can be noticed, that variables definition are unordered and the code contains forward references, e.g. the output variable `P1.excptflag1` is referenced before `P1` definition. The event *Stop* has two inputs. As the result of model refactoring an OrJunction (variable `Junction_Before_Stop`) was introduced into the model. For the event *Start* constituting a root element, the boolean variable `Start_trigger` with corresponding transition was added.

4.5 Generation of specification

As a specification language we use CTL, which allows to formulate properties applying to a tree of computations (paths) starting from a given state. As the tree defines a set imaginable futures, CTL is called the branching time logic. CTL formulas are combinations of two types of operators *path quantifiers* and *linear-time operators*. The path quantifiers are: A (for every path in a tree) and E (there exists a path in a tree). Temporal operators include: G ($G\,p$ means that p holds true globally in the future) and F ($F\,p$ means that p holds true sometime in the future).

Typically a specification formally describing requirements is entered by a user. However, we tried to derive some *liveness* requirements based on control flows within ArchiMate model (see Definition 1). The implemented procedure generating a set of specifications comprises the following steps:

1. Build a set of paths $\Pi = \{\pi_i\}$ within the Archimate model,
2. Restrict elements in π_i to events only (elements from the set Evt)
3. Build a partial mapping R: $Evt \rightarrow 2^{Evt}$
4. Generate the specification for each pair $(e_i, R(e_i))$ in R

In the first step (1) a depth-first search starting from *roots* (ArchiMate elements having no predecessors) is performed. It returns a set of paths $\Pi =$

$\{\pi_i\}$ comprising ArchiMate elements linked by control flow relation. For a path $\pi_i = (e_{ib}, \ldots, e_{ie})$, its last element e_{ie} is either a final element in the model (without successors) or a branching element (already present in π_i). The set of obtained paths reflects only topological relations within the process model. The procedure does not attempt to interpret the model according to any behavioral semantics. This is left to the verification tool.

In the step (2) the paths from Π are restricted to ArchiMate elements being events. We decided to focus in requirements specification on elements of *Event* type, because in business process definitions they are typically used to mark important process states (e.g. initial, final and intermediate events).

In the next step (3) a partial mapping $R\colon Evt \to 2^{Evt}$ is built. The mapping R assigns all (potentially) reachable events to first events appearing in paths from Π

Finally, in the step (4) for each event $e \in \operatorname{dom} R$, a pair $(e, R(e))$ is converted into a set of specifications taking the form of (1), where $\mathcal{G} = \{AG, EG\}$, $\mathcal{F} = \{AF, EF\}$ and $\mathcal{O} = \{\bigvee, \bigwedge\}$.

$$\mathcal{G}((f \to \mathcal{F}(\underset{l_i \in R(f)}{\mathcal{O}} l_i))) \tag{1}$$

For the process presented in Fig. 4 an example of generated CTL specification is: `AG (Start.outflag -> EF (Stop.outflag | Excpt.outflag))`. It is equivalent to the statement: *for every path, starting with Start event, it is possible to reach a state, where Stop or Excpt events occur*. This requirement is obviously true. Another generated specification: `AG( Start.outflag -> AF( Stop.outflag & Excpt.outflag ))` is false, as justified by a counterexample path comprising 14 elements produced by NuSMV.

5 Business process example

In this section we present a more realistic example of ArchiMate specification describing a process of selling a product (a service) to a client. The process is divided into two stages: preparation presented in Fig. 5 and finalization (Fig. 6) separated by the event *Contract Prepared*. The finalization phase is far more complex. During execution of the *Acceptance* subprocess two events: *Timeout* and *Rejection* may occur and in consequence loop back the whole process to a previous stage. Contract signing by both parties, as well as *Implementation* and *Signed contract scanning* are placed between ArchiMate AndJunctions (forks and synchronization joins.)

Based on this specification the NuSMV model was generated. During refactoring phase second AndJoin in Fig. 6 was split into two (serving as join and fork). Another join was added before the interaction *Terms negotiation*. The *main* module of NuSMV comprised 22 finite state machines, whereas the flattened model consisted of 47 state variables. Considering their ranges, the whole state space comprised $3^9 \cdot 2^{37} \cdot 5 = 1.35 \cdot 10^{16}$ states, whereas the number of

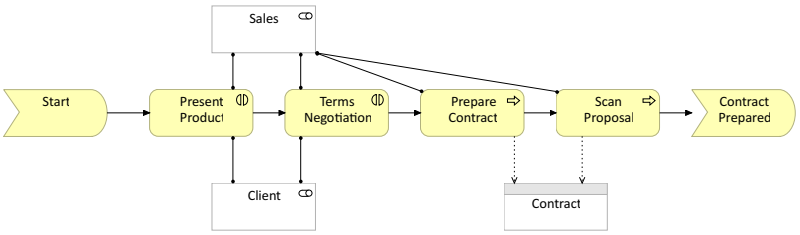


Fig. 5. Preparation stage

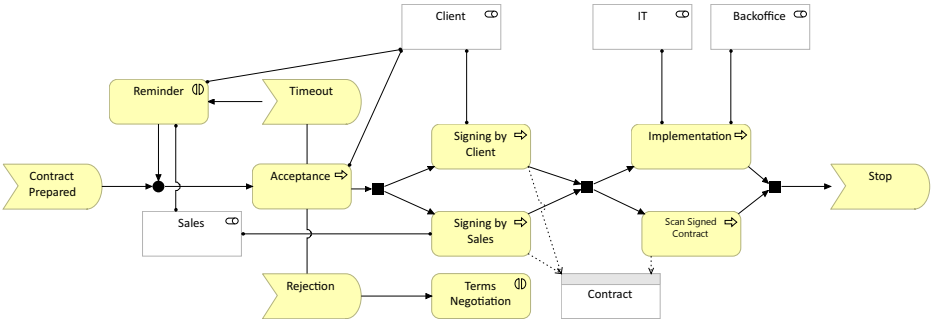


Fig. 6. Finalization stage

reachable states calculated by NuSMV based on the internal OBDD representation was equal to $7.039 \cdot 10^7$. The transition relation was total, i.e. no deadlocks were present.

Examples of automatically generated specifications based on control flow are:

1. `AG( Start.outflag ->`
   `AF( Stop.outflag | Timeout.outflag | Rejection.outflag ))`
2. `AG( Start.outflag ->`
   `AF( Stop.outflag & Timeout.outflag & Rejection.outflag ))`

The first was checked to be true, whereas the second, as expected, occurred false. The path constituting a counterexample for the second specification comprised 42 states.

An example of a user-defined CTL specification equivalent to the statement that *all contracts signed by a client are finally scanned* is:
`AG (Signing_by_Client.outflag ->(AF Scan_Signed_Contract.outflag ))`.
NuSMV reported is as true.

For the presented example verification of one CTL specification took about 43 seconds. However, after applying dynamic variable ordering this time decreased to 6.64 sec. We may conclude that although the state explosion is alleviated in NuSMV by employing internal OBDD representation, it seems that it still remains a problem. Hence, dedicated model generation techniques focusing on keeping models compact, e.g. generating partial models, should be employed.

6 Conclusions

This paper investigates the problem of automatic verification of behavioral specification embedded within ArchiMate models. We were motivated by an idea of developing a solution tightly integrated with Archi modeling tool that would allow to extract behavioral elements from an ArchiMate specification, then fully automatically translate it into a model in NuSMV language and finally verify it with the NuSMV model checker. Requirements specification in form of CTL formulas can be entered by user, but the implemented tool is capable of generating specifications based on analysis of control flows. We discuss methods of model transformation applied in the implemented software: language patterns used to model atomic processes and other elements, as well as rules for translating them into NuSMV modules. Finally, application of the method is presented on an example of a business process.

An issue that requires closer investigation is the time efficiency of the verification process. Surprisingly, papers discussed in Section 2, which claim to use the NuSMV model checker, do not provide evaluation data on complexity of verified processes and verification times. On the other hand, sample specifications distributed with NuSMV are built manually and optimized. The main factor influencing memory usage is the ordering of OBDD variables used in internal representation. Many NuSMV models are distributed with files defining ordering, which was determined by performing a separate optimization task.

ArchiMate is primarily a modeling language. It does not define semantics of behavioral elements. Application of certain modeling patterns and methods of translating them to a NuSMV language is an arbitrary decision related to assumed semantics. Probably, several options and alternatives controlled by program parameters should be considered.

References

1. Anderson, B., Hansen, J.V., Lowry, P., Summers, S.: Model checking for e-business control and assurance. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 35(3), 445–450 (2005)
2. Beauvoir, P.: Archi, archimate modelling tool (2015), http://www.archimatetool.com/, [Online; accessed March 2015]
3. Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Mbp: a model based planner. In: Proc. of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information (2001)
4. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) 24(3), 293–318 (1992)
5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification. pp. 359–364. Springer (2002)
6. Clarke, E., Heinle, W.: Modular translation of statecharts to smv. Tech. rep., Citeseer (2000)
7. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the futurebus+ cache coherence protocol. Formal Methods in System Design 6(2), 217–232 (1995)

8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification, pp. 1–30. Springer (2012)
9. Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR) 28(4), 626–643 (1996)
10. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proceedings of the 12th International Conference on Database Theory. pp. 252–267. ACM (2009)
11. Fu, X., Bultan, T., Su, J.: Formal verification of e-services and workflows. In: Web Services, E-Business, and the Semantic Web, pp. 188–202. Springer (2002)
12. Fuxman, A., Pistore, M., Mylopoulos, J., Traverso, P.: Model checking early requirements specifications in tropos. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. pp. 174–181. IEEE (2001)
13. Huuck, R.: Formal verification, engineering and business value. In: Olveczky, P.C., Artho, C. (eds.) Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, Kyoto, Japan, November 12, 2012. Electronic Proceedings in Theoretical Computer Science, vol. 105, pp. 1–4. Open Publishing Association (2012)
14. Klimek, R., Szwed, P.: Verification of ArchiMate process specifications based on deductive temporal reasoning. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, Kraków, Poland, September 8-11, 2013. pp. 1103–1110 (2013), http://fedcsis.org/2013/
15. Klimek, R., Szwed, P., Jedrusik, S.: Application of deductive reasoning to the verification of ArchiMate behavioral elements. Informatyka Ekonomiczna 29, 76–97 (2013)
16. Mongiello, M., Castelluccia, D.: Modelling and verification of BPEL business processes. In: Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006. Fourth and Third International Workshop on. pp. 5–pp. IEEE (2006)
17. Morimoto, S.: A survey of formal verification for business process modeling. In: Proceedings of the 8th international conference on Computational Science, Part II. pp. 514–522. ICCS '08, Springer-Verlag, Berlin, Heidelberg (2008)
18. OMG: Business Process Model and Notation (BPMN) version 2.0. Tech. rep., OMG (January 2011), http://www.omg.org/spec/BPMN/2.0
19. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
20. Szpyrka, M., Biernacka, A., Biernacki, J.: Methods of translation of Petri nets to NuSMV language. In: Popova-Zeugmann, L. (ed.) Proceedings of the 23th International Workshop on Concurrency, Specification and Programming, Chemnitz, Germany, September 29 - October 1, 2014. CEUR Workshop Proceedings, vol. 1269, pp. 245–256. CEUR-WS.org (2014), http://ceur-ws.org/Vol-1269/paper245.pdf
21. Szwed, P., Chmiel, W., Jedrusik, S., Kadluczka, P.: Business processes in a distributed surveillance system integrated through workflow. Automatyka/Automatics 17(1), 127–139 (2013)
22. The Open Group: Open Group Standard. Archimate 2.1 Specificattion. Van Haren Publishing, Zaltbommel (2013)
23. Watahiki, K., Ishikawa, F., Hiraishi, K.: Formal verification of business processes with temporal and resource constraints. In: Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on. pp. 1173–1180. IEEE (2011)