Analysis And Interpretation Modeling

1 INTRODUCTION

A use of models as an executable artifact is a desired target of the model-driven development. The main idea of this was proposed in the Object Management Group's (OMG) guide on Model Driven Architecture (MDA) in 2001 (Miller and Mukerji, 2001). It requires having a formal (computer-understandable) model as an input. This input model must have enough details for getting complete source code. Completeness of the model means that the model is likely to be complex. Unfortunately, constructing the complete complex model cannot be done in one moment; this requires incremental work. Besides, understanding of such a model is not a trivial activity (Quante, 2016b). The attempt to create a complete language is the Unified Modeling Language (UML) proposed by the OMG (OMG, 2005). At present, the UML language can be used for its model interpretation or for code generation. However, its independence from software development techniques and platforms, size, incoherence, different interpretations, lack of causality and frequent subsetting lead to ambiguous semantics, cognitive misdirection during the development process, inadequate capture of system's properties and so on (Osis and Donins, 2017). The mentioned problems affect the quality of generated code as well as require additional transformations and mechanisms for UML model interpretation as discussed in Section 3.2.

In the process of building a complete input model, it is necessary to be able to "run", "debug" and "test" it similarly to debugging and testing the source code. From one point of view, manual reviews and sometimes partial prototyping can be successfully applied, but manual work is slow and does not exclude human mistakes due to complexity of models. From the other side, automated model checking techniques exist like, for instance, those of used to verify requirements and design for real-time embedded and safety-critical systems. These automated model-checking techniques require a use of formal modeling language, e.g., finite state machines can be used for "control-oriented" systems (applied in aerospace, avionics, automotive, etc.) (Palshikar, 2004). However, the model checking also requires manual translation of requirements descriptions into these languages, as well as it is hard to follow the checking algorithm execution in case of "data-oriented" systems where business logic is more important than behavior.

Another opportunity is interpretation or "running" of models. The *goal* of this paper is to overview existing implementations of model interpretation techniques and their applications in software development.

Section 2 presents the considerations that influence the research flow. Section 3 gives an overview of existing model interpreters. Section 4 is dedicated to discussion on the main findings. Section 5 concludes the paper with main results and discussion on validity of the research done.

2 RESEARCH QUESTIONS AND METHOD

Reading literature, one can find that it is possible to “simulate” a model, to “execute or run” a model, and to “interpret” a model. Let us look what does it mean. Simulation of models usually is used for real complex systems that are hard to be analyzed analytically, e.g., day-to-day operation of a bank. “A simulation model is a parameterised model that is solved on the computer…” (Arnott, 2012). It uses statistical data about operation of the real system it describes as well as autonomously existing events. Execution of models usually refers to an executable UML model, which can represent either planned to be built software or already existing one. A systematic overview on execution of UML models (Ciccozzi, Malavolta, and Selic, 2019) indicated two ways of UML models execution, namely, interpretation and translation. The authors’ research showed that a large part of existing UML execution tools uses a translative approach to a programming language (in most cases it is Java). The reason is a desire to get generated production artifacts as the result of modeling. However, interpretative approaches are mostly used for validating and improving functional correctness at the beginning of development. Although there are few existing model-level debugging implementations, both translative and interpretative executions can be combined with the simulation mechanism based on autonomously existing events. Simulation of models extends users’ abilities to debug and correct the model itself. Thus, interpretation of models is just a way how to execute a model.

A topological functioning model (TFM) introduced by Jānis Osis in 1969 (Osis, 1969) is a formal mathematical model that can be used both for business modeling and software source code generation (Nazaruka and Osis, 2019; Osis and Donins, 2017) starting from the automated processing of textual descriptions of the system’s structure and functionality (Nazaruka, 2020). However, the constructed TFM requires experts’ check. In order to assist this check TFM interpetation can be used. Thus, we want to understand what is the state of the art in the field and what model interpretation techniques could be more beneficial.

In order to achieve the goal stated, a list of questions has been defined, answers on which should be found during the overview of the existing publications. The research questions are as follows:

- What software model interpreters exist?
- What is the purpose for which a model interpreter is used?
- What input models do they use?
- How model interpretation is implemented?
- What systems characteristics do model interpreters allow users to check?
- How interaction with a user is implemented? Is it possible at the run-time?

The information on software model interpreters published by IEEE, ACM, ScienceDirect and those of presented in Google Scholar have been searched for a period from 2000 till 2020. The starting year, 2000, was selected as the year when Model Driven Architecture was presented to the public. An additional filter used was the context, i.e., software development including embedded and real-time systems. Besides that, the research works related to pure code generation from models were omitted.

The keywords used are “model interpretation,” “model execution,” and “model interpreter.” It must be said that quite many research works were found, and selection of those found can illustrate the main principles of model interpretation.

In order to structure the information found, these key publications are grouped according to the interpretable models, i.e., business process models, UML models and domain specific models.

3 MODEL INTERPRETATION TECHNIQUES

The general definition of interpreters states (Karsai, 1999): “model interpreters are transformation programs that walk a graph (the model objects), and perform actions during this process.” A model interpreter should have the following components: model structure, visitors and traversals. The *model structure* can be a graph or its textual specification. Thus, it usually represents a set of nodes and transitions between them. One of the issues here is that the graph may consist of nodes of heterogeneous types with a certain action related to a certain type. The common solution is a use of the *visitor* design pattern, which implementing *Visitor* classes can be assigned to a specific type of a node and be invoked if needed (Karsai, 1999). In order to capture information on what node the interpreter need to go next, *traversals* are used. Traversals are objects that contain the traversal code fragments and can also contain state information. Traversals and visitors are to be directly linked to each other. However, visitor-based interpreters have two limitations: coupled interpreter-logic and generation-logic, as well as a minimal reuse of code (Hill and Gokhale, 2012).

In the model-driven engineering, a model interpreter is a software component that operates on the information captured in a system model to produce some useful artifact (Edwards, Seo, and Medvidovic, 2008). Model interpreters may extract the model structure and properties (Edwards et al., 2008) or be based on the modifiable pre-developed meta-model (Shroff, Agarwal, and Devanbu, 2009). However, universal interpreters that are independent of the application domain do not exist (Djukić, Luković, Popović, and Ivančević, 2012).

Cook et al. have investigated many techniques to define the interpretations of models (Cook, Delaware, Finsterbusch, Ibrahim, and Wiedermann, 2009). According to the authors, one common approach is to use a translator from one modeling language to another modeling language or to code. Dynamic interpreters are also common in practice, although they have received less attention in research publications. One point of confusion is that the term “interpreter” is often used to mean “translator” in the model-driven literature. Here, the term “interpreter” is also used in its more traditional meaning as a meta-program that executes a program in the given language, the same as the authors use in (Cook et al., 2009). According to the authors, translators have the advantage that they can produce efficient code and target any runtime environment. Interpreters are often easier to write then compilers, but they are typically slower and do not necessarily integrate easily with other parts of a system, which may be written in compiled languages.

Business Process Model Automation

According to (Ferme, Lenhard, Harrer, Geiger, and Pautasso, 2017), workflow automation relates to the execution of automated business processes within Workflow Management Systems (WfMSs). Ferme et al. (Ferme et al., 2017) think that the most critical part in WfMSs is the modeling language implementations that do not satisfy language standards; nevertheless, there are several proposed standards for the modeling language. The main part of the WfMSs responsible for business model running is a process engine. This is not a new thing now. There are many commercial and open-source solutions of process engines, e.g., IBM’s WebSphere, Windows Workflow Foundation, JBPM, Bonita, Apache ODE, ActiveBPEL (ActiveVOS), Oracle BPEL Process Manager (Oracle BPM), etc. In this part, several of them will be considered to understand main common principles implemented in them.

The BPEL engine uses business processes descriptions as “a series of activities” which are executed by web services (ARIS BPM Community, 2021). According to the BPEL vendors, a graphical representation of a BPEL process reduces complexity of the process flows and allows integrating BPEL diagrams into the process architecture model. These graphical models are exported into a BPEL XML (eXtensible Markup Language) based script. In essence, the BPEL combines block structures and allows describing transitions between them as directed graphs (Juric, n.d.).

The IBM’s WebSphere line’s (IBM, 2011) process engine takes as a basis high-level XML process definitions in the BPEL supplemented with code fragments in Java. The execution and monitoring of models requires IBM’s WebSphere Application Server that at present supports Kubernetes and microservices.

Microsoft also has had its embeddable workflow engine called Windows Workflow Foundation (WF) (Microsoft, 2017). The WF is a part of the .NET framework that allows developing workflow functionality using XAML-encoded workflow definitions. This possibility is implemented in the Workflow Designer (WD), which is a visual designer and debugger for the graphical construction and debugging. The basic building blocks in WF are activities. The developer can select the needed one from the Activity Designer Library templates and Visual Studio will create an activity designer definition in XAML and a code-behind implementation file. Besides that, it is possible to model arguments with values, variables for use in data-binding scenarios and conditional statements, and expressions (i.e., controls used in workflow activities to enter and evaluate expressions). Activities can form the sequential flow or a state machine workflow, can be grouped and even ordered according to a hierarchy. In WF, a workflow running is thread-based. Debugging of the workflows is possible via Workflow Designer or at the XAML level.

Process engines mentioned here depend on the language used for the workflow design and lack flexibility. Therefore, if someone wants to provide the flexibility in this context, they should create a *process virtual machine* and then some suites for each potential language.

BMPN models can also be interpreted, e.g., to ensure modification of processes without recompiling them completely, as it is demonstrated for smart contracts (Lopez-Pintado, Dumas, Garcia-Banuelos, and Weber, 2019). The main difficulty in model-based smart contracts used for blockchain-based business process execution is a lack of flexibility (because they are attached to different versions of the model) and high deployment costs. The suggested Caterpillar interpreter supports all three process modeling perspectives, namely, control-flow, data, and resources. The idea is to create a new subprocess, relate it to the existing one and generate code for this new subprocess. The one issue that remains in this implementation is a lack of consistency checking.

Another possible solution is described by Weigold, Kramp and Buhler (Weigold, Kramp, and Buhler, 2007), where they suggest the ePVM, an embeddable process virtual machine. It consists of a lightweight library with basic functionality to state and control flow managements, process persistence and transactions, monitoring, inter-process communication, and communication with the host application, as well as optional functionality for workflow systems support, human interaction and even integration with “not native” process languages. In the ePVM, a business process is developed using a programming environment supporting communicating extended finite state machines (CEFSM). The CEFSM is implemented as libraries functions called via API. Thus, the business process model can be defined by an ordinary JavaScript function. The process definitions can be also structured by using packages. Concurrency and synchronization are provided by a threading mechanism. Interprocess communication is implemented by a message-passing mechanism. Communication with the host application is organized via host API.

The Event-driven Process Execution Model (EPEM) supporting process virtual machine called OncePVM is presented in (Wu, Wei, Gao, and Dou, 2012). By the authors’ opinion, the event-driven architecture is more preferable for highly concurrent systems than the thread-driven (or similar to it – process-driven). The main idea is that a process description consists of nodes and transitions. Each node or action contains one or more ports and listens or produces the dedicated events. Besides, the action also holds a reference of a context to preserve the corresponding state. The context maintains all the requested variations of an executing instance. Preconditions and postconditions serve as indicators of the task readiness to start and success of the completion and are checked by independent actions called connectors. Thus, connectors could be modified to adapt different pre/post-conditions and ensure necessary behavior. The OncePVM has a triple-layered architecture. The bottom layer consists of a set of basic services for processing actions, threads, and objects. The middle layer provides support for the runtime process execution. The top layer is dedicated for event queue managements and even scheduling. Before execution, the process description (in any process description language) is parsed into memory objects for each presented element and then these memory objects are transformed into executable actions and events. Then it is possible to start the execution of the process by deploying the execution objects and managing them by the runtime engine. Faults are handled as related fault events.

The interesting workflow-based model interpreter called InstantApp (Shroff et al., 2009) is dedicated to web applications. It provides functionality for runtime modifications of web application forms by changing the application model stored in a model repository and cached in memory during runtime. This interpreter gives a possibility not only to change the graphical controls in forms but also to modify business logic associated with them. For handling changes in business logic, the authors use a “logic map” which is an extension of Google’s MapReduce. The logical map is a graph with *create*, *search*, *update*, *merge* and *reduce* nodes. In general, InstantApp also implements visitor-based pattern but by using other means: each activity in a logic map is assigned to an InstantApp form. The interpreter searches all the activities that the user can perform while using this form and executes them.

Summarizing, business process and workflow automation is quite developed nowadays and supports deployment of different scale applications as well as new types of application architectures. However, most of the tools are language dependent excluding process virtual machines. Nevertheless, the implementation of virtual machines is more complex, it is the future of workflows automation.

UML Model Interpretation

The various approaches for executing UML models are analyzed in Gotti and Mbarki's work (Gotti and Mbarki, 2016) in order to understand their particularities and supporting tools. Supporting the conclusion done by Cook et al. (Cook et al., 2009), the authors rightly note the UML model execution is possible either by compiling the model or by interpreting it.

Model interpretation assumes the presence of a virtual machine which is an environment where executable UML models can be read and executed without generation of executable code from them.

In 2001 the idea of a UML virtual machine was presented in (Riehle, Fraleigh, Bucka-Lassen, and Omorogbe, 2001) where authors suggested to avoid the step of generation of objects from classes but to interpret them directly. As the authors wrote, all objects should exist in the same memory space thus providing the immediate causal connection between a model and its instances. This was a kind of rapid prototyping but without direct coding. Such implementation allows runtime exploration of the model by a developer. Instructions for this machine are specified in UML, but the persistent version is in XMI (XML Interchange) language. The memory model uses facilities provided by Java, the implementation language of the virtual machine. The virtual machine has logical and physical architecture that assumes that for each object in the logical architecture a logical class and a physical (Java) class are to be defined. The proposed virtual machine allowed using of the restricted UML and OCL (Object Constraint Language), since UML's semi-formal constructs may lead to the unexpected interpretations. The most formal technique, i.e., UML state charts, were used as the primary tool for modeling object behavior. Besides that, the imperative part also was required, and the authors added "hand-programmed policy classes." The advantages of this solution are rapid user feedback, application architecture independent execution of the model. The main weakness was a limited usefulness of the UML.

Gotti and Mbarki indicated that nowadays an executable UML model usually may consist of three diagrams – the class diagram, the state chart diagram and the activity diagrams – plus behavioral specifications (Gotti and Mbarki, 2016). Executable elements of UML can be defined using foundational UML (fUML) and the supplementing Action language for foundational UML (Alf). Nevertheless, an executable UML model is first transformed into a formal control flow graph or a finite state machine and then analysis of these artifacts is performed. The analysis checks paths, dead-ends, transitions, etc. The same principles are implemented in the UML model interpreter for verification and monitoring of UML models of embedded cyber-physical systems that plays a role of synchronous observer automata (Besnard, Teodorov, Jouault, Brun, and Dhaussy, 2019). For model verification, the authors apply the OBP2 model-checker to modeled UML state machines. For runtime monitoring, the observer monitors the current execution trace of the system. The UML model interpreter has the action language which is used to access states in the state chart diagrams. The action language can specify guards and effects of transitions and provide C macros to access UML instances and their attributes.

The systematic overview of solutions suggested before and after appearing of fUML and Alf languages were overviewed in (Ciccozzi et al., 2019). The authors have found 14 interpretive solutions, e.g., Moka that is Eclipse Papyrus plug-in, fUML virtual machine, BridgePoint, etc. All of them focus on the higher-level execution for simulation and model-based analysis. The authors concluded that there is no preferable solution among the overviewed. The level of readiness of these solutions also differs and the commercial status does not affect it much (Ovchinnikova and Nazaruka, 2016, 2017).

Domain Specific Model Interpretation

Model-based approaches got visible acceptance in today's automotive software system development (Quante, 2016b). By raising the level of abstraction, they became more understandable to humans. However, the degree of complexity increases over time and limits maintenance and understanding them as well as increases the corresponding costs. Jochen Quante has presented the idea of the interpreter for such automotive software maintenance and calibration (Quante, 2016b). This interpreter can be called generic, since for any new modeling language only the transformation to the intermediate representation must be added. The interpreter itself is based on control-flow and dependency graph. It executes them step-by-step considering branches and function calls. Concrete operations and decisions can be delegated either to a visitor (the evaluator) class, or to abstraction strategies that transform concrete values into abstract values. The interpreter can detect which code is never executed and can trace back to the model level and indicate which parts of the model are irrelevant, because it investigates all possible paths. Besides, the interpreter can be used for extracting formulas from code even symbolically getting conditional formulas as a result, for concolic testing (i.e., generating test cases with full path coverage), and for recording and replay measurements. Details on the suggested architecture and functions are presented by Quante in (Quante, 2016a). The potential limitation of architecture of such model interpreters is that it is hard to modify the list of points to be visited and the list of points to be generated (Hill and Gokhale, 2012).

An approach for testing models, generated code and target interpreters is presented by Djukić et al., where they suggest using action reports – special programs (generators) that conduct synchronization between the domain-specific model, client applications and target interpreter (Djukić et al., 2012). The authors indicate that the limitation of their approach is a lack of a generated code interpreter, because then every model modification requires generation of the application code, its compilation and rerun of the application. The authors also note that in case of domain-specific models neither the source nor the target language needs to be known in advance. Thus, one of the approaches that can be used to create a stronger logical relationship between debugging environments and modeling tools is the use of patterns specific to each combination of a domain-specific language and a target platform. The action reports in the form of metadata is a specification of transition from one diagram state to another. In its turn, the target interpreter interprets synchronization commands defined in the report and sets corresponding property values in the report definition. The modeling tool takes this modified report and runs operations on the graphical interface elements. The transfer of action report is done by packets.

In order to interpret architecture of computerized numerical control (CNC) systems (Zhaogang, Di, Feng, and Suhua, 2007) the authors propose the model interpreter that generates source code automatically from models. The models are created in conformity with the predefined meta-model for CNC systems. The model interpreter acts like a translator or, in other words, a compiler of a programming language. When the model interpreter works as a model translator, it produces code in the input language of other analysis or simulation tools, e.g., UPPAAL. Otherwise, it produces code, static data-structures, configuration files or customized generic components: these artifacts can be compiled and linked.

The interesting improvement of the visitor-based interpreters by using generative programming techniques is presented by James Hill and Aniruddha Gokhale (Hill and Gokhale, 2012). The authors call their model interpretation technique *Metaprogrammable Interpreters for Model-driven Engineering (MIME)*. Similarly to (Quante, 2016a), Hill and Gokhale consider that using the Strategy design pattern and Parametrized Strategy design pattern partially addresses the problem of a reuse of core interpretation logic, but note that it is still not possible to modify the set of points to be visited. As a solution they propose the interpreter built on the parametrized strategy with a use of template metaprogramming technique. This allows avoiding points that should not be visited and, thus, the corresponding code also is not generated.

Another application of model interpreters can be for self-adaptive systems, where executable runtime “megamodels” are interpreted and modified at runtime (Vogel and Giese, 2012). A megamodel is a feedback loop specification “by means of operations, the control flow between operations, and the models that are used by operations” (Vogel and Giese, 2012). This kind of systems suggests separation of the domain logic and the adaptation logic. In between both, a feedback loop (or even a number of feedback loops) ensures that the adaptation logic dynamically governs the domain logic according to changes in the environment or requirements to the domain logic or to circumstances in the domain logic itself.

4 DISCUSSION

The search for publications on model interpretation techniques resulted in understanding that there is plenty such solutions with different purposes and for different contexts. However, the research done gave as answers on the questions set in Section 2.

What software model interpreters exist? There are many solutions that can be considered as model interpreters. Most of them are language specific. This means that architecture and implementation of the process engine is based on the language used for process specification (design). This leads to inflexible solutions that is a problem.

What is the purpose for which a model interpreter is used? The model interpreters may be used for executing and modifying the workflows (or other process flow solutions) at the runtime; for supporting the domain analysis, as well as for simulating, debugging, testing, executing, and monitoring domain-specific models of the embedded and real-time systems.

What input models do they use? The input models can be specified as workflows in BPMN or BPEL languages or other languages developed for the same field, as process specifications or domain-specific models in XML and XMI or other XML based or XML similar language (such as YAML).

How model interpretation is implemented? Implementations differ from the simplest two layered solutions to multiple executing engines dedicated to a certain task, e.g., messaging, process execution, process monitoring, etc. The one common thing is that at the physical layer most of them are based on some kind of finite automata thus providing expected execution of the designed model.

What systems characteristics do model interpreters allow users to check? Since model interpreters support parametrization, it is possible to check any control flow and data flow related characteristics. However, the focus is mostly on functional aspects, and rare solutions consider non-functional characteristics.

How interaction with a user is implemented? Is it possible at the run-time? The main principle of the model interpretation is to support direct interaction between an application and a user at the run-time without interrupting it.

Returning to the question on the TFM interpretation, we can conclude that this obviously is possible. The TFM is based on the principles of system theory and algebraic topology. Thus, it can be translated into any kind of finite automata and interpreted. The process engines are not suitable for TFM interpretation since this model combines processes in one digraph. The process engines will require additional activities on separating scenarios. The concept of virtual machines seems more appropriate. The state chart or finite automata obtained from the model would contain all possible states and transitions, thus allowing proper model debugging and simulation. Besides that, this solution could be flexible to using different TFM specification languages.

5 CONCLUSIONS

The aim of the given research was to overview the state-of-the-art software model interpretation and opportunities that the current solutions give to model debugging, testing, simulation, and execution.

The six questions were set and clear answers on them were achieved. The more advanced solutions are presented in the field of workflow automation within business process management systems. This direction proposes complex industrial interpreters of models. The main weaknesses of these solutions are the complexity and dependency on the modeling language. Plenty solutions are presented in the field of UML models execution, but only several of them are interpreters not compilers. The main weaknesses of these solutions are the degree of their readiness and uncertainty of the UML itself. Many ad-hoc solutions exist in the field of embedded and real-time systems, where domain-specific models require proper simulation before implementation. The main weakness here is that each solution is domain and model dependent.

Future research directions are related to deeper research on development of the TFM interpreter that should apply advantages presented and avoid weaknesses found.

.