Data-flow based Model Analysis

Christian Saad, Bernhard Bauer
Programming Distributed Systems Lab, University of Augsburg, Germany
{saad, bauer}@informatik.uni-augsburg.de

Abstract

The concept of (meta) modeling combines an intuitive way of formalizing the structure of an application domain with a high expressiveness that makes it suitable for a wide variety of use cases and has therefore become an integral part of many areas in computer science. While the definition of modeling languages through the use of meta models, e.g. in UML, is a well-understood process, their validation and the extraction of behavioral information is still a challenge.

In this paper we present a novel approach for dynamic model analysis along with several fields of application. Examining the propagation of information along the edges and nodes of the model graph allows to extend and simplify the definition of semantic constraints in comparison to the capabilities offered by e.g. the Object Constraint Language. Performing a flow-based analysis also enables the simulation of dynamic behavior, thus providing an "abstract interpretation"-like analysis method for the modeling domain.

1 Introduction and Motivation

Meta modeling is an easy and concise way to formalize the structure of an application domain. Today, it is widely spread in computer science, most notably in the field of software engineering where the internal design of software is often described using models. Approaches like the Model-driven Architecture (MDA) improve the development process e.g. through automated code generation. Arguably the most important industrial standard in this area is the Unified Modeling Language (UML) which in turn is based on the Meta-Object Facility (MOF) framework[1].

However, the abstract syntax as defined by the meta model is often not sufficient to guarantee well-formedness. Complex restrictions that cannot be expressed through the syntax are known as static semantics. To a certain degree, they can be implemented using the Object Constraint Language (OCL), thereby extending the expressiveness of the modeling language. Due to the static nature of OCL, it is not capable of validating dynamic properties that are highly dependent on the context in which the elements appear, e.g. the correct nesting of parallel paths in activities.

An advantage of using models as means of specifying systems is their versatility. In addition to code generation or model transformation, their use can also be leveraged by extracting implicitly contained information through an evaluation of model elements and their mutual relations. This may include metrics applicable for the modeling domain, e.g. Depth of Inheritance Tree (DIT) or Number of Children (NOC), or the compliance with a set of predefined modeling guidelines. Depending on the range of application, extracting knowledge about the dynamic properties of a model may even allow to identify inactive fragments in a way similar to dead code elimination in the translation of software programs. To statically deduce information about the dynamic behavior of a model, e.g. to calculate the valid execution paths of workflows and thereby approximating their runtime behavior, can be considered an abstract interpretation of dynamic semantics.

Current methods are usually not capable of performing a static analysis of dynamic aspects in order to express context-sensitive well-formedness rules or to simulate workflows. The approach discussed in this paper is designed to overcome these limitations by extending the static character of OCL constraints with a dynamic flow analysis, thus offering a powerful and generically applicable method for model validation and simulation. It is based on the data-flow analysis (DFA) technique, a well-understood formalism used in compiler construction for deriving optimizations from a program's control flow graph. The adaption of the DFA algorithm is simplified by the conceptual similarities between the areas of compiler construction and (meta) modeling: Both rely on the definition of domain specific languages (DSL) which operate on (at least) two layers of abstraction.

The concept of using DFA for model analysis has been introduced in [4]. In this paper we extend and update this description along with an evaluation of the advantages, possible shortcomings and proposed solutions based on the experience gained up to now. Additionally, we discuss several use cases which are in the focus of our research and present the current status of our implementation.

This paper is structured as follows: The basic concept of data-flow analysis for models along with a definition of how it can be specified and a corresponding evaluation algorithm is detailed in Section 2. Several use cases are presented in Section 3 before we give a summary of the concepts described in this paper and an outlook on future developments.

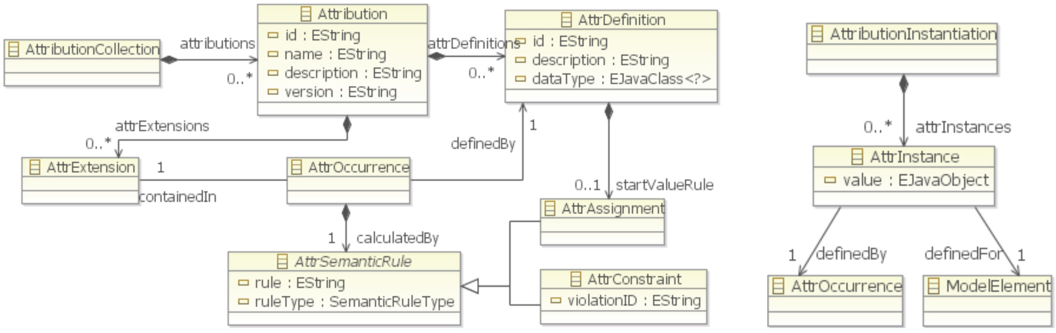
[1] Specifications are available at OMG's website: `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`

2 Data-Flow Analysis on Models

As outlined in our previous literature, the conceptual similarities between context-free grammars, which form the basis for many algorithms in compiler construction, and MOF's multi-layered architecture of (meta) models allow to implement a DFA-like analysis method for the modeling domain. We aligned the four MOF levels (M3-M0) with the abstraction layers of attribute grammars[2] and devised a model-based attribution technique for assigning data-flow equations to meta model classes and evaluating them for arbitrary models.

It was proposed that the attributes containing the semantic rules (the equivalent to the data-flow equations) should be given a type definition consisting of a name, a data type and an initial value. Occurrences of these definitions can then be assigned to M2 classes along with a rule that will calculate the results for M1 objects depending on the values available at other attributes. If a model shall be analyzed according to a given attribution, these occurrences have to be instantiated at the according model objects. An evaluation algorithm is then responsible for executing the rules in a valid order ensuring that required input arguments are available at the time of execution. This process is repeated until all attribute values are stable (the fix-point). Several iterations of (re)evaluating the attribute instances may be necessary until the results have been propagated along all (cyclic) paths. To keep in line with the notion that everything is a model, a meta model describing the structure of the attribution was given.

2.1 Refined Definition



(a) Refined attribution meta model (AttrMM) (b) Attr. instance meta model (AttrM)

Figure 1: Meta models for data flow definitions

Figure 1(a) shows the refined attribution meta model *AttrMM*: *AttributionCollections* serve as containers for a set of *Attributions* which consist in turn of *AttributeDefinitions* and *AttributeExtensions*. *AttributeDefinitions* have to be assigned a *dataType*. Their initial value is returned by an *AttrSemanticRule* of the type *AttrAssignment*[3]. Occurrences (*AttrOccurrence*) can be attached to classes in the original meta model through *AttributeExtensions*. They possess an *AttrSemanticRule* which calculates their iteration value.

This design has the advantage of introducing dependencies only from the attribution to the meta model but not the other way round. This way, attributes can be defined and stored separate from the meta model, an important aspect when integrating with existing tools.

To ensure the compatibility with existing meta modeling frameworks, the resulting attribution instances were given their own meta model *AttrM* which can be seen in Figure 1(b). Each *AttributeInstance* has a *value* slot for storing intermediate and final evaluation results. Attribute instances representing the results are connected to their defining *AttrOccurrences* in the attribution (and thus to the attributed meta model class) as well as to the concrete model object to whose meta class the corresponding *AttributeExtension* was assigned.

An important aspect of modeling is the concept of generalization. Therefore, when creating attribute instances and attaching them to model elements, the generalization hierarchy of the meta model has to be considered, i.e. an attribute connected to class A should implicitly be available at instances of subclasses of A. Also, in compliance with the MOF standard, support for the redefinition of attributes should be provided. This means if two *AttrOccurrences* O_A and O_B of the same *AttributeDefinition* O were assigned to classes A and B and B is a subclass of A, then O_B overrides O_A at all instances of B.

[2] It was shown that, while attribute grammars can be used to define data-flow equations, in their original form they are too restrictive and do not fit seamlessly into the modeling domain. Therefore, a model-based solution was chosen that drops most of these limitations at the cost of a slightly more complex - but also more versatile - evaluation algorithm.

[3] An assignment returns a value of the specified data type while a constraint just evaluates to ”true” or ”false”, ”false” representing a detected error in the model indicated to the user by the given *violationID*.

We assumed that the input dependencies, i.e. the occurrences whose instance values are required as arguments at other instances, will be explicitly specified by the user. However, this has proven to be impractical for two reasons: Aside from complicating the design of a DFA analysis, declaring relations on the meta layer has the drawback of introducing unnecessary instance dependencies if only a subset is actually needed. Instead, the evaluation algorithm is now responsible for requesting the calculation of required input for attribution rules during their evaluation.

2.2 Attribute Evaluation

DFA algorithms like the work-list approach assume that the output relationships are known beforehand in order to update the set of depending variables after each iteration. Since we are working with dynamic dependencies, another approach was chosen for evaluating an attribution for a given model: First, attributes must be instantiated in accordance to the given semantics and initialized with their start value. Then, the semantic rules have to be invoked in a valid order, backtracking their dependencies during execution, storing results and passing them as input arguments to the calling rules. This is repeated until the fix-point of the analysis has been reached.

The dependencies between attribute instances, stemming from their use as input arguments of semantic rules, impose the structure of a directed acyclic graph originating from a single root element on the execution order. This graph can be viewed as the result of a depth-first search along the input relations of the attributes, however in the case of cyclic dependencies the target x of the responsible back edge $y \rightarrow x$ is now substituted by a newly created virtual node x'. The resulting tree-like representation (which may still contain forward and cross edges) is referred to as *dependency chain*. After each bottom-up evaluation run the virtual nodes are updated with the new value available at their reference node, i.e. the result at x is transferred to x'. The evaluation is repeated until the value at each virtual node equals the result at its reference node which means that the fix-point has been reached.

As an example, consider that we need to determine the set of predecessors in a flow graph lying on a *direct path* from the start node while omitting optional routes. This can be achieved by adding the local node to an intersection of the same (recursively calculated) sets at preceding nodes. Nodes with no predecessors then correspond to leaves in the dependency chain while cyclic paths induce back edges and therefore the creation of virtual nodes.

A model may contain multiple dependency chains while at the same time a single chain may also split up into several chains if a root attribute instance was chosen that is not the absolute root, i.e. other attributes depend on it.

Because this algorithm executes the semantic rules according to their dependencies, the amount of redundant calculations is negligible. Nevertheless, there is still room for optimization, e.g. by calculating starting values only for leaves, through isolated recomputation of cycles or by parallelizing the evaluation. Also, a formal and practical evaluation of the algorithm's performance is necessary.

3 Use Cases

To demonstrate the applicability of the approach, we now give several use case examples which can be implemented using the model-based DFA technique: Section 3.1 demonstrates common compiler construction analysis while 3.2 and 3.3 deal with the domain of business processes (but are also applicable e.g. for UML activity diagrams).

Additional application fields which are currently under evaluation include the extraction of metrics in the area of model-driven testing and the definition of OCL constraints that avoid complex navigation statements and can be therefore more easily adjusted if the meta model changes.

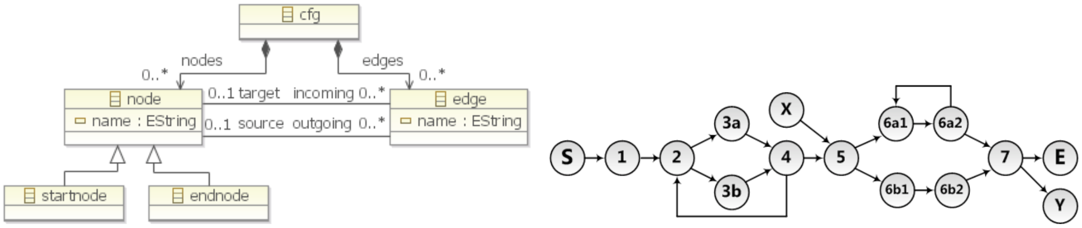
3.1 Analysis of Control-Flow Graphs

To simulate a traditional data-flow analysis we have implemented a meta model for control-flow graphs (CFG) (cf. Figure 2(a)) and instantiated the CFG model that can be seen in Figure 2(b) which will serve as an example.

An attribution model was created containing the following attributes for *node* along with their assignment rules based on an extended OCL syntax which allows to request attribute values by triggering the evaluator module:

```
is_reachable:  self.incoming.source.is_reachable()→includes(true)

is_live:  self.outgoing.target.is_live()→includes(true)

all_predecessors:  self.incoming.source.name→ union(self.incoming.source.all_predecessors())→asSet()

scc_id:  let self_pred : Set(String) = self.all_predecessors()→including(self.name) in
         if (self.incoming.source.all_predecessors()→asSet()=self_pred) then self_pred→hashCode() else 0 endif)

scc_nodes:  if (not(self.scc_id() = 0)) then self.incoming.source→collect(predNode : node |
         if (predNode.scc_id()=self.scc_id())
         then predNode.scc_nodes() else Set{} endif) →flatten()→asSet()→including(self.name) else Set{} endif
```

While *is_reachable* determines if a node can be reached from the start node by checking if at least one predecessor is reachable (the start node has a distinct rule setting *is_reachable* to true), *is_live* checks if a path to the end node



(a) A simple control-flow graph meta model

(b) An example control-flow graph model

Figure 2: Control-flow graph example

exists. As expected, the evaluation returns that X is not reachable and Y is not live. Defining the semantic rules as constraints, corresponding exceptions are thrown indicating the type and the location of the error.

The list of transitive predecessor nodes is calculated through *all_predecessors* by creating the union of the *all_predecessors* attributes at the source nodes of incoming edges and adding to it the set of names of these nodes.

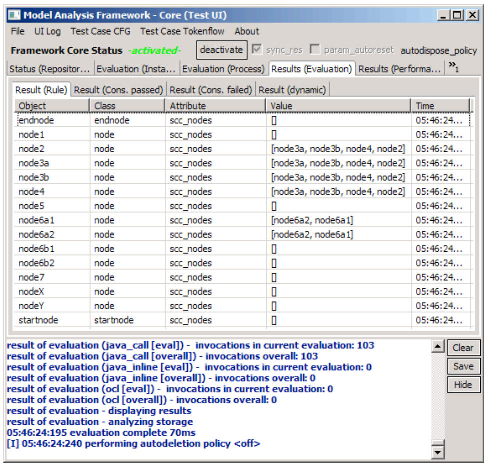
The attribute *scc_id* assigns a unique ID to each node that is part of a cyclic flow. This is accomplished by comparing the union of the *all_predecessors* attributes at preceding nodes to the local value of this attribute. If both sets contain the same elements, an ID is generated from the hash codes of the nodes that take part in the cycle.

Now, using the *scc_id*, the semantic rule for *scc_nodes* is able to determine which nodes take part in a cycle by building the set of predecessors with the same *scc_id*. Figure 3 shows the final values for this attribute which can again be used as input for the algorithms presented below.

Calculating *scc_nodes* requires 220-310 rule executions using an unoptimized evaluation algorithm. Once the OCL environment has been initialized, overall execution takes about 110ms on a standard desktop computer. Implementing the rules in Java leads to more verbose definitions but reduces the time required to about 50ms.

3.2 Business Process Decomposition

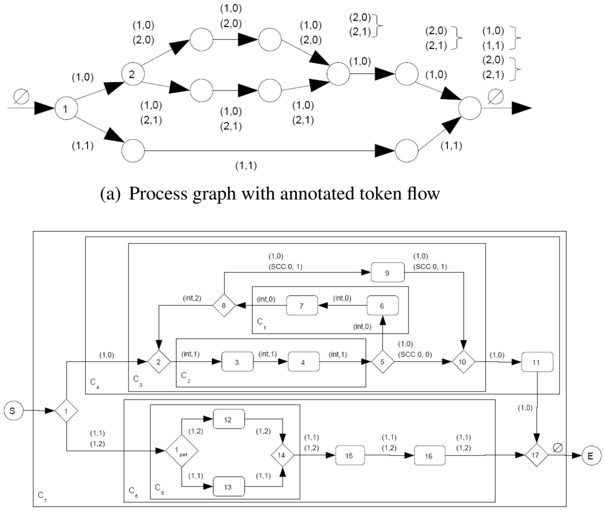
Decomposing a business process into a hierarchical layout of single-entry-single-exit (SESE) components is a common requirement, e.g. allowing to translate BPMN processes to BPEL or validate workflow graphs (cf. 3.3). In [3], the authors describe a decomposition algorithm based on token flow analysis that is able to handle cyclic graphs and to classify the detected components. Tokens are created at branches and are propagated along the control-flow as can be seen in Figure 4(a). In a second step, the tokens originating from the same vertex converge and are removed (indicated by curly brackets). Similar token labelings identify the SESE components although unambiguous classification and cyclic paths require some additional handling. Figure 4(b) shows a decomposition.



Object	Class	Attribute	Value	Time
endnode	endnode	scc_nodes	[]	05:46:24...
node1	node	scc_nodes	[]	05:46:24...
node2	node	scc_nodes	[node3a, node3b, node4, node2]	05:46:24...
node3a	node	scc_nodes	[node3a, node3b, node4, node2]	05:46:24...
node3b	node	scc_nodes	[node3a, node3b, node4, node2]	05:46:24...
node4	node	scc_nodes	[node3a, node3b, node4, node2]	05:46:24...
node5	node	scc_nodes	[]	05:46:24...
node6a1	node	scc_nodes	[node6a2, node6a1]	05:46:24...
node6a2	node	scc_nodes	[node6a2, node6a1]	05:46:24...
node6b1	node	scc_nodes	[]	05:46:24...
node6b2	node	scc_nodes	[]	05:46:24...
node7	node	scc_nodes	[]	05:46:24...
nodeX	node	scc_nodes	[]	05:46:24...
nodeY	node	scc_nodes	[]	05:46:24...
startnode	startnode	scc_nodes	[]	05:46:24...

```
result of evaluation (java_call [eval]) -  invocations in current evaluation: 103
result of evaluation (java_call [overall]) - invocations overall: 103
result of evaluation (java_inline [eval]) -  invocations in current evaluation: 0
result of evaluation (java_inline [overall]) - invocations overall: 0
result of evaluation (ocl [eval]) -  invocations in current evaluation: 0
result of evaluation (ocl [overall]) - invocations overall: 0
result of evaluation - displaying results
result of evaluation - analyzing storage
05:46:24:195 evaluation complete 70ms
[I] 05:46:24:240 performing autodeletion policy <off>
```

Figure 3: Evaluation result: Nodes that are part of cycles



(a) Process graph with annotated token flow

(b) A business process hierarchically divided into SESE components

Figure 4: Decomposition of (business) processes [3]

Since the algorithm is based on the creation and propagation of information, it is an ideal use case for a generic DFA-based approach as opposed to a proprietary implementation. Several steps of this algorithm have been realized (including the cycle-detection presented above) with the goal of a comparison to the authors' implementation.

3.3 BP Validation and Transformation

Using the method presented in [5], the soundness of (a)cyclic business processes - i.e. the absence of local deadlocks and lack of synchronization - can be tested in linear time. This is accomplished by traversing the SESE hierarchy in bottom-up direction, applying heuristics to categorize each sub-graph according to the structure of its elements. This way, if an error is found, the SESE context in which it appears allows to track down its approximate location. Implemented using DFA definitions, this algorithm can be easily integrated into the decomposition process in order to perform the validation already during the component identification phase.

Making use of the SESE decomposition also enables to transform graph-oriented BPMN diagrams (Business Process Modeling Notation) to block-oriented BPEL code (Business Process Execution Language). The author of [1] describes data-flow equations that, if computed for the SESE fragments, yield information on how the respective component can be translated to BPEL code. Since this algorithm is already defined in the form of a data-flow analysis, the implementation using the model analysis approach is straightforward.

4 Conclusions and Future Investigations

In this paper we have described a refined version of our data-flow based approach to model analysis that introduces the notion of DFA to the modeling domain and is built upon widely-accepted standards and definitions.

To the best of our knowledge there exists no comparable methodology in the context of model analysis although different DFA techniques have been considered for use in the modeling domain: The authors of [2] discuss the advantages of control-flow information for the testing phase of the software development process. A concrete use case is presented in [6], using flow-analysis to derive definition/use relationships between actions in state machines.

The presented formal method completes common techniques like OCL with the possibility of describing (cyclic) information flows in the model graph based on local propagation and (re)calculation of attribute values. This allows a more robust definition of semantic constraints since complex navigation statements which are a common drawback of OCL can be avoided. Instead, the required information can be ”transported” to where it is needed. Aside from validation scenarios, the ability to extract context-sensitive data enables to analyze dynamic aspects of models, e.g. valid execution paths in control-flows or the SESE components that make up a business process definition. This way, model-based DFA constitutes a generic and versatile ”programming-language” for implementing a wide variety of algorithms that would otherwise each require a proprietary definition.

To verify the feasibility of this approach, the Model Analysis Framework (MAF) project was created to serve as basis for performance tests under realistic settings and allow to evaluate future extensions of the presented definitions and algorithms. It was designed to act as a parametrizable and modular research platform that for example allows to choose between different inheritance semantics and evaluation algorithms as well as at the same time being suited for productive use. MAF, which is based on the Eclipse Modeling Framework (EMF) and the Eclipse OCL interpreter, will soon be available at `http://code.google.com/p/model-analysis-framework/`.

Aside from formalizing and improving the evaluation algorithm to achieve a better theoretical and practical performance the main focus in the ongoing research is on the implementation and evaluation of further use cases.

References

[1] Luciano García-Bañuelos. Pattern Identification and Classification in the Translation from BPMN to BPEL. *OTM 08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008*, pages 436–444, 2008.

[2] Vahid Garousi, Lionel Bri, and Yvan Labiche. Control Flow Analysis of UML 2.0 Sequence Diagrams. 2005.

[3] Mathias Götz, Stephan Roser, Florian Lautenbacher, and Bernhard Bauer. Token Analysis of Graph-Oriented Process Models. *New Zealand Second International Workshop on Dynamic and Declarative Business Processes (DDBP), in conjunction with the 13th IEEE International EDOC Conference (EDOC 2009)*, September 2009.

[4] Christian Saad, Florian Lautenbacher, and Bernhard Bauer. An Attribute-based Approach to the Analysis of Model Characteristics. *Proceedings of the First International Workshop on Future Trends of Model-Driven Development in the context of ICEIS'09*, 2009.

[5] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 43–55, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] T. Waheed, M.Z.Z. Iqbal, and Z.I. Malik. Data Flow Analysis of UML Action Semantics for Executable Models. *Lecture Notes in Computer Science*, 5095:79–93, 2008.