Data-Flow Based Model Analysis and Its Applications

Christian Saad and Bernhard Bauer

University of Augsburg, Germany
{saad,bauer}@informatik.uni-augsburg.de

Abstract. In this paper we present a data-flow based approach to static model analysis to address the problem of current methods being either limited in their expressiveness or employing formalisms which complicate seamless integration with standards and tools in the modeling domain.

By applying data-flow analysis - a technique widely used for static program analysis - to models, we realize what can be considered a generic "programming language" for context-sensitive model analysis through declarative specifications. This is achieved by enriching meta models with data-flow attributes which are afterward instantiated for models. The resulting equation system is subjected to a fixed-point computation that yields a static approximation of the model's dynamic behavior as specified by the analysis. The applicability of the approach is evaluated in the context of a running example, the examination of viable application domains and a statistical review of the algorithm's performance.

1 Introduction and Motivation

Modeling languages have become a prominent instrument in the field of computer science as they enable the formalization of an application domain's concepts, their properties and the relationships between them. An abstract syntax given in the form of a meta model allows to validate and enforce structural constraints and fosters automated processing of the formalized information, e.g. through code generation or model transformations. In addition, the rise of modeling techniques has lead to new approaches to software engineering such as the Model-driven Architecture [1] and Model-based Testing [2].

Since their introduction, the OMG's [3] Meta-Object Facility (MOF) and derived languages like the Unified Modeling Language (UML) have become the de-facto standard in industry and research alike. Building upon a common meta meta model, the MOF's *M3* layer, a family of *M2* languages has evolved with applications ranging from software engineering to business process management.

An important factor for the popularity of modeling techniques is that they are often perceived to provide an intuitive way for practitioners to formalize application domains. However, the less rigorous theoretical framework can also be a serious drawback when attempting to assert a model's correctness: Although the basic form of the language expressions (models) is given by the abstract syntax (meta model), it is often necessary to enforce additional constraints on

A. Moreira et al. (Eds.): MODELS 2013, LNCS 8107, pp. 707–723, 2013.

the language's structural layout. The subset of these constraints that can be statically verified is known as the static semantics or the well-formedness rules of a language. To formalize these rules, a technique is required that allows to enrich meta model elements with a specification of their static semantics.

Over time, existing formal approaches have been proposed for the purpose of model analysis. However, this usually involves a translation of (meta) models into logic-based representations [4, 5] resulting in a gap between the two domains that can be difficult to manage on a technical level but may also lead to problems on a conceptual level as model-specific semantics have to be mapped to the logic-based systems on which the analyses are defined and executed.

This issue is addressed by the OMG's Object Constraint Language (OCL) which allows to annotate constraints at meta model elements and to evaluate them for models. However, limitations of its expressiveness due to its static navigational expressions are the subject of ongoing discussion [6, 7]. The `closure()` operator[1] introduced in version *2.3.1* (January 2012) of the specification only applies to `Set` types and is limited to calculating the transitive closure of a relationship. Finally, it has been argued that OCL itself lacks a proper formalization [8] and multiple proposals have been made to address this problem [9–11].

The approach detailed in this paper represents a generic, declarative method for computing properties that can be derived from the structural layout of a model. It is based on attribute grammars (AG) and data-flow analysis (DFA), two well-understood and well-defined methods from the field of compiler construction used to validate static semantics and to derive optimizations from a program's control-flow respectively. Data-flow analysis is a powerful method that implicitly provides support for transitive declarations. For example, the following (recursive) definition computes the transitive closure of the parent relationship: `allParents = directParent` ∪ `directParent.allParents`. Since DFA applies fixed-point semantics to resolve cyclic dependencies, analyses can derive static approximations of dynamic behavior, e.g. by computing which nodes will be visited on all paths leading to an action in an activity diagram.

In this paper we detail the approach initially outlined in [12]. Its intended target audience are language engineers responsible for developing (model-based) domain-specific languages (DSL) and tooling as opposed to users of the implemented languages (who may also be developers in their respective domain).

The presented methodology allows to attach data-flow attributes to elements of MOF-based meta models in a fashion similar to OCL's derived attributes. These attributes can then be automatically instantiated and evaluated for derived models. Result computation consists of the execution of data-flow rules, applying fixed-point evaluation semantics when necessary. Structural differences between modeling and formal languages required an adaption of the worklist algorithm commonly employed to solve DFA equation systems.

The proposed analysis specification language is a textual DSL which itself is based on a meta model that is tied to the MOF. On a technical level, the

[1] An example use case would be the enforcement of non-cyclic generalization hierarchies for Classifiers: `self->closure(superClass)->excludes(self)`.

presented approach therefore integrates with standards, languages and tools in the modeling domain, avoiding the inherent difficulties in the application of formal methods. Its applicability is evaluated in the context of several use cases.

This paper is structured as follows: In Section 2, we outline basic principles of data-flow analysis and attribute grammars. Their suitability for model analysis is examined in Section 3.1 through a comparison of the domains of modeling and formal languages. Section 3.2 describes the structure and semantics of the specification language while Section 3.3 demonstrates how resulting equation systems can be computed taking into account the adjustments made to traditional DFA. The approach is evaluated in Section 4 and its versatility is exemplified through several use cases in Section 5. We conclude with a survey of related work and a summary of the approach along with an outlook on future developments.

2 Background

Data-flow analysis (DFA, [13]) is a method commonly used in compiler construction in order to derive context-sensitive information from a program's control-flow, usually for optimization purposes. Canonical examples for this approach include the calculation of reaching definitions or variable liveness analysis.

Data-flow equations are annotated at control-flow nodes $n \in N$ and operate on sets containing values from a specific value domain: Applying a join operator $\varDelta \in \{\bigcap, \bigcup\}$ to the output values calculated at neighboring nodes in the flow graph yields the input value for each node: $in(n) = \varDelta_{m \in \varTheta(n)} out(m)$ where $\varTheta$ is either the direct predecessor or successor relationship. By using values at preceding nodes as input, information is propagated in a forward direction[2]. Inserting the intersection operator for $\varDelta$ retains only values which are contained in any incoming set, i.e. information which reaches a node on all of its incoming paths, while the use of the union operator aggregates results "arriving" on any incoming path. The result $out(n)$ is determined by removing ($kill$) information which is locally destroyed and adding (gen) information which is locally generated: $out(n) = gen(n) \cup (in(n) - kill(n))$. The equation system formed by the entirety of all equation instances induces a global information flow throughout the graph as local results are distributed along outgoing paths.

In the presence of back edges in the control-flow, the equation system contains cyclic dependencies. This case is handled by applying fixed-point evaluation semantics: First, all nodes are initialized with either the empty set in the case of $\varDelta = \bigcup$, or the complete value domain for $\varDelta = \bigcap$. Then, the equations are evaluated repeatedly until all values are stable. This indicates that the most accurate approximation, a minimal or maximal fixed-point, has been detected. The existence of a fixed-point itself is guaranteed if operations are monotonic and performed on values which have a partial order with finite height.

[2] Some analyses, for example the detection of live variables, require information flow in a backwards direction in which case the process is reversed, i.e. results calculated at successor nodes are used as the equations' arguments.

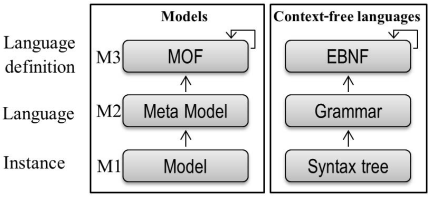
A canonical optimization is the worklist algorithm: Starting with the execution of the flow-equation at the entry node, each time the (re)calculation of an equation yields a value that differs from its previous result, the equations at the depending nodes are added to the worklist since they are the ones affected by the new input. This process is repeated until the worklist is empty.

A second technique for static analysis used in compiler construction are attribute grammars. Introduced by [14], they are used to analyze context-sensitive information - e.g. the set of defined variables - depending on the layout of the language expression's syntax tree. Traditional AGs extend a context-free grammar G with a set of attributes A, each of which is assigned to a (non) terminal symbol $X \in N \cup T$ and is either of the type Inh (inherited) or Syn (synthesized). The attributes can be thought of as property fields of the nodes in the syntax trees, their values being calculated by semantic rules R assigned to the productions that describe how an attribute value can be calculated from the values of other attributes in the same production. Semantic rules are given in the form $X_i.a = f(...)$, where a is an attribute assigned to X_i and f is an arbitrary function that calculates a result for a based on its arguments. This leads to information being transported from one place in the AST to another, either bottom-up (synthesized) or top-down (inherited). Therefore, attribute grammars can be considered to be form of data-flow analysis [15] and support the definition of regular DFA if supplemented with fixed-point semantics [16, 17].

3 Data-Flow Based Model Analysis

3.1 Applying Data-Flow Analysis to Models

Transferring DFA to the modeling area requires a careful consideration of conceptual similarities and differences between the domains of formal languages and modeling. As discussed in [18, 19], relationships between these technical spaces can be identified by aligning their respective layers of abstraction.



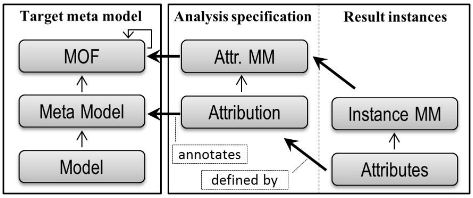


Fig. 1. Alignment of abstraction layers **Fig. 2.** Analysis specification/instances

Figure 1 illustrates how MOF employs a common meta meta model on the *M3* abstraction layer to implement capabilities for defining *M2* meta models which represent the abstract syntax of a modeling language. Prominent examples include the Unified Modeling Language (UML) and the Business Process Modeling

Notation (BPMN). In MOF terminology models, e.g. UML diagrams or BPMN processes, are located on *M1*. A model is syntactically valid if it complies with the syntactic restrictions defined in its meta model. A similar hierarchy is used by formal languages, more specifically context-free grammars, which are used for programming language specification. In addition to enforcing syntactic correctness, the integrity of static semantic constraints can be validated by extending the grammar with semantic attributes as described in Section 2.

From a conceptual view point, the analysis of instances (models / syntax trees) therefore requires analysis specification on the language level which has to be supported by appropriate constructs on the *M3*/language definition layer. In the DFA context, a method is required which enables to assign flow equations to meta model elements $Model_{M1} \lhd MetaModel_{M2}$ alongside semantics for instantiating and solving the analysis for arbitrary models ($\lhd$ signifies $instanceof$).

To accomplish this, an approach was chosen that mirrors the concept of attribute grammars to assign semantic attributes to meta model elements. While this could be achieved by either extending the *M3* layer or the meta model with constructs for analysis specification, this would lead to incompatibilities with standards and tools that depend on compliance to MOF. Instead, attributes and their instantiations are defined separately (Figure 2), allowing all artifacts to remain unaware of the analyses. The language for analysis specifications - termed *Attributions* is given by an attribution meta model (*Attr.MM*) while their instantiations are defined by a separate meta model (*Instance MM*).

Computing flow-based analyses for models requires adaptations of the traditional algorithms for evaluating attribute grammars and DFA. The reason for that is that edges in model graphs - which are instances of associations or references defined in its meta model - denote relationships between objects which may possess arbitrary semantics depending on the domain for which the meta model was defined. In fact, associations between elements are often not directed, and if they are, two elements may be connected via multiple paths with undefined semantics in the context of flow-analysis. As such, they cannot be aligned with edges in flow graphs which carry the implicit semantics of a control flow, making it safe to automatically route information along incoming/outgoing paths.

In attribute grammars, attributes in syntax trees depend on results from the same grammatical production as input. This means that different rules may apply in different contexts depending on the production instance's respective neighbors in the syntax tree. In that, productions compare to classifiers in the meta model while the occurrences of productions in the syntax tree correspond to objects in the model. However, compared to syntax trees, the graph structure of models does not offer an easily identifiable direction for inheritance/synthesis.

In summary, information flow in models is highly specific to an application domain and an analysis since they don't possess an inherent flow direction as exists for control-flow graphs and syntax trees. This problem can be circumvented by ensuring that information is routed only along relevant, analysis-dependent paths: To provide maximal flexibility, rather than flow-equations being automatically supplied with input values depending on the context in which they appear, they must be able to request required input as needed. Input/output dependencies between attribute instances are therefore encoded inside the flow-equations, thereby superimposing the model with a (dynamically constructed) data-flow graph. The work-list algorithm must be adapted to record dependencies as they become visible through the execution of the rules and to schedule the re-computation of unstable attribute values using this information.

3.2 Analysis Specification and Instantiation

In this section we describe the language for analysis specification and the instantiation semantics in the context of a running example. They are based on and comply to the Essential MOF (EMOF) subset of MOF and have been implemented using the Eclipse Modeling Framework (EMF, [20]).

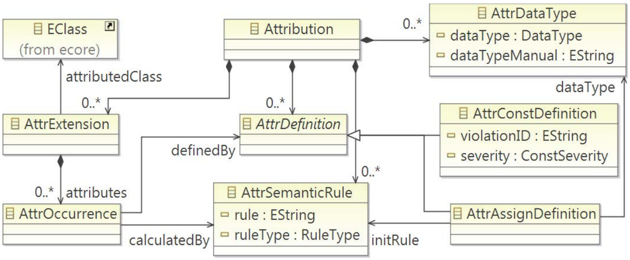


Fig. 3. Analysis meta model (*Attr.MM*)

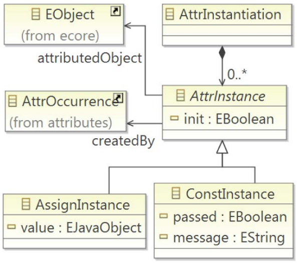
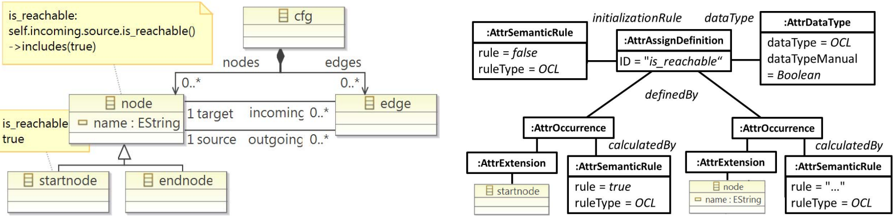


Fig. 4. Instantiation MM

Figure 3 shows the elementary concepts of the analysis specification meta model: In the notion of attribute grammars, *attribute occurrences* indicate the presence of *attribute definitions* (of the type *assignment* or *constraint*) at classes (*EClass*) in the target meta model. *Attribute extension* containers connect these *occurrences* to meta model classes through the *attributedClass* relationship. Attached to the *definitions* and *occurrences* are *semantics rules* (corresponding to data-flow equations) that calculate the fixed-point initialization and iteration values respectively. They may be defined in an arbitrary language for which the language interpreter implements an interface to the DFA solver (cf. Section 3.3).

The instantiation meta model (cf. Figure 2) is shown in Figure 4. Each *attribute instance* links to the *occurrence* from which it was instantiated and to the model object for which it was created. Depending on the *attribute definition* type, it is either an *assignment instance*, returning a result value complying to the definition's *data type*, or a *constraint instance* of type `boolean` indicating whether a constraint/well-formedness rule was violated.

This is exemplified in Figure 5(a) which shows a reachability analysis annotated at a control-flow graph meta model. It is assumed that an *attribute definition* with the id `is_reachable`, the type `boolean` and the initialization value `false` was specified. Two *occurrences* of this *definition* have been assigned to the classes `node` and `startnode`, the latter overwriting the first to always



(a) Attributed meta model (b) Abstract syntax of the attribution

Fig. 5. Reachability analysis defined for control-flow graph meta model

return `true` at instances of `startnode`. The abstract syntax of the attributed meta model can be seen in Figure 5(b). The OCL rule uses the injected operation `is_reachable()`[3] to request the value of this attribute at the respective predecessor nodes from the DFA solver, resulting in a recursive definition in which a node is considered to be reachable if at least one of its predecessors is reachable.

The instantiation semantics for attributes follows the EMOF semantics for the instantiation of meta model classes: An attribution AT(MM, AT_{DEF}, AT_{RULE}, AT_{OCC}, AT_{DT}, AT_{TYPE}, AT_{ANN})[4] extends a meta model MM(MM_{CL}, MM_{GEN}) given by the set of classes MM_{CL} and their generalization relationships MM_{GEN} indicating inheritance of structural and behavioral features in accordance to EMOF semantics. The attribution consists of attribute definitions AT_{DEF}, each possessing a data type (AT_{DT}) and an initialization rule (AT_{RULE}) assigned by the relation AT_{TYPE}. Furthermore, the annotation relation AT_{ANN} ties each occurrence in AT_{OCC} to a class $c \in \text{MM}_{CL}$ and an iteration rule in AT_{RULE}.

An instantiation INST(AT, M, INST_{AT}, INST_{LINK}) contains attribute instances INST_{AT} for an attribution AT and a model M $\lhd$ MM with objects M_{OBJ} and a relation M_{TYPEOF} denoting their class type. For each *obj* $\in \text{M}_{OBJ}$, an attribute instance $i \in$ INST exists *iff* there are ≥ 1 occurrences *occ* $\in \text{AT}_{OCC}$ for the class type of *obj* or its super-types. To realize overwriting at subtypes the most specialized type is used. This can be implemented by starting at a model object's concrete type and traversing the generalization hierarchy upwards. For the first occurrence of each distinct attribute definition which is encountered an instance is created. Multiple inheritance is only supported if generalization relations are diamond-shaped and a unique occurrence candidate can be identified.

The control-flow model in Figure 6 depicts the instances of the attribute `is_reachable` which are attached to the corresponding model elements. The dashed lines indicate the implicit dependencies encoded in the flow equations. The corresponding abstract syntax representation is shown in Figure 7.

The meta model is complemented by a concrete syntax using the Eclipse Xtext parser/editor generator which maps grammatical symbols to meta model

[3] Attribute access operations can be automatically injected into an OCL environment: For all *attribute definitions* connected to a class through *occurrences*, an operation is added to the class with the id of the *definition* and the *data type* as return type.

[4] Multiple attributions can be merged if they extend the same meta model.

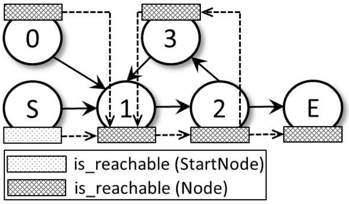


Fig. 6. Attributed model

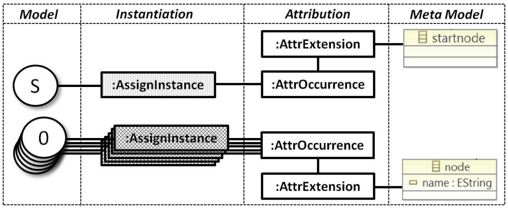


Fig. 7. Abstract syntax

elements. The syntax comprises all relevant artifacts: *Attribute definitions*, *attribute extensions*, *semantic rules* and *datatypes*. Except *attribute extensions* (and the therein contained *attribute occurrences*), all objects can be cross-referenced by other parts of the attribution. This excerpt from the language's grammar defines the declaration syntax for *assignments* and *occurrences* and their connection to the targeted meta model classes:

```
Attribution returns attribution::Attribution:
 'attribution' id=ID '{' ( (attrDefinitions+=AttributeDefinition)∗ &
                           (attrSemanticRules+=SemanticRule)∗ &
                           (attrDataTypes+=AttrDataType)∗ &
                           (attrExtensions+=AttrExtension)∗ ) '}' ;

AttributeDefinition returns attributes::AttrDefinition:
  'attribute' (AttrAssignDefinition | AttrConstDefinition) ;

AttrAssignDefinition returns attributes::AttrAssignDefinition:
  'assignment' id=ID (name=STRING)? (``["description=STRING``]")? ':'
     dataType=[datatypes::AttrDataType]
     'initWith' initializationRule=[semanticrules::AttrSemanticRule] ';' ;

AttrExtension returns attributes::AttrExtension:
  'extend' attributedClass=[ecore::EClass] 'with' '{' (attributes += AttrOccurrence)∗ '}' ;

AttrOccurrence returns attributes::AttrOccurrence:
  'occurrenceOf' definedBy=[attributes::AttrDefinition]
  'calculateWith' calculatedBy=[semanticrules::AttrSemanticRule] ';' ;
```

The following example[5] specifies the attributes `is_reachable`, `all_predecessors` and `scc_id` which perform reachability analysis and calculate a node's transitive predecessors as well as strongly connected component (SCC) membership.

```
attribution flowanalysis {
  – attribute definitions (consisting of id, data type and initialization rule)
  attribute assignment is_reachable : OCLBoolean initWith boolean_false;
  attribute assignment all_predecessors : OCLSet initWith set_empty;
  attribute assignment scc_id : OCLBoolean initWith int_zero;

  – semantic rules (ocl rules using helper operations injected into OCL environment)
  rule ocl isreachable_node : standard
   " self .incoming.source. is_reachable ()−>includes(true)";
  rule ocl allpredecessors_node : imperative
   " self .incoming.source ∪ self .incoming.source. all_predecessors ()";
```

[5] Common types (e.g. `OCLBoolean`) and rules for trivial calculations such as `boolean_true` are contained in a "standard library" omitted here for lack of space. For the same reason, imperative OCL statements were converted to formula.

```
  rule ocl sccid_node : imperative
   " self ∪ self . all_predecessors () == self.incoming.source. all_predecessors ()";

  – attribute occurrences (define occurrences and bind them to classes)
  extend node with {
   occurrenceOf is_reachable calculateWith isreachable_node;
   occurrenceOf all_predecessors calculateWith allpredecessors_node;
   occurrenceOf scc_id calculateWith sccid_node;
  }
  extend startnode with {
   occurrenceOf is_reachable calculateWith boolean_true;
  }
}
```

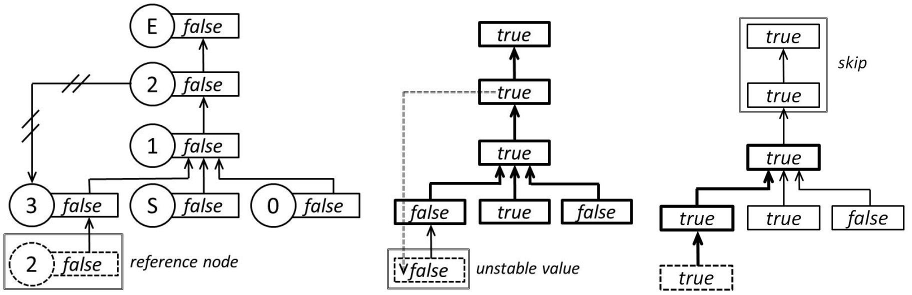
3.3 Dynamic, Demand-Driven Fixed-Point Analysis

Compared to an exhaustive algorithm, a demand-driven DFA solver limits computation to a subset of requested results [21]. In this context, this subset corresponds to a set of requested instances $\mathrm{INST}_{AT(REQ)} \subseteq \mathrm{INST}_{AT}$, e.g. all instances of a specific attribute, all attributes located at a given class etc. However, unknown to the solver, transitive dependencies to instances $\mathrm{INST}_{AT} \setminus \mathrm{INST}_{AT(REQ)}$ may exist. For example, `scc_id` relies on `all_predecessors`. $\mathrm{INST}_{AT(REQ)}$ must therefore be expanded dynamically on discovery of these dependencies.

Because dependencies between attribute instances are "hidden" inside flow-equations, traditional methods for call-graph construction [22, 23] are not applicable. The dependency graph that superimposes the attributed model therefore has to be constructed on-the-fly during the fixed-point computation using dynamic dependency discovery. As a side-effect, support for the inclusion of transitive dependencies as described above is implicitly provided by such an algorithm.

The adapted worklist algorithm carries out the following steps: The requested instances $\mathrm{INST}_{AT(REQ)}$ are initialized before their associated iteration rules are executed. If a rule requests another instance's value as input, this access is relayed to the solver which is thereby able to record the dependency between the calling and the called instance and at the same time can discover calls to attributes not in $\mathrm{INST}_{AT(REQ)}$. A new iteration starts at the leaves of the constructed dependency graph, i.e. at instances without input dependencies, and at cyclic dependencies whose values are updated after each iteration.

As an optimization for this method in the context of flow-based model analysis, we propose a demand-driven, iterative algorithm that constructs and operates on a directed acyclic dependency graph with multiple root and leaf nodes. Each root node represents an attribute instance not required as input by other instances. Leaves are either instances which themselves do not depend on input or so-called reference nodes that indicate the presence of cyclic dependencies and are used to trigger the fixed-point computation. This method compensates for the absence of a CFG structure by maintaining a set of starting points for the fixed-point iterations (the leaf nodes) while the identification of independent branches enables parallelized computation. It also provides a comprehensive representation of the computation process useful for debugging purposes.



(a) Recorded dependencies (b) After iteration it_0 (c) Final result after it_1

Fig. 8. Dependency discovery and result computation

In the first phase of the evaluation process, the DFA equations corresponding to the instances from $\mathrm{INST}_{AT(REQ)}$ are executed. By monitoring the input requests during the rules' execution, the solver is able to construct an initial dependency graph from the recorded data-flow dependencies. The graph is then converted into an acyclic representation by identifying cyclic dependencies through a depth-first traversal strategy and replacing back edges with *reference nodes*. Finally, all instances in are reset to their respective initialization value. This is demonstrated in Figure 8(a) for the example presented in Figure 6: The back edge between `is_reachable` instances at nodes *3* and *2* has been replaced by a reference node and all values have been reset to `false`.

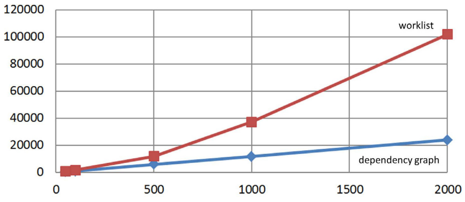
In the second phase, the graph is traversed repeatedly in a bottom-up fashion, starting at unstable leaf nodes. Each instance node's iteration rule can be executed once its input dependencies have been satisfied, i.e. all of its children have been either executed or do not have an unstable node in their transitive children set. Parallelization is possible if rules are executed through a working queue to which the parents of traversed nodes are added once the aforementioned condition applies. Since rules are free of side effects, it is safe to stop traversal at nodes if their execution yields the same result for an instance as in the last iteration. This avoids unnecessary recalculations of stable results. After the traversal, unstable instances at cyclic dependencies can be detected: A reference node is classified as unstable if its result from the previous iteration $it_{(n-1)}$ is different from the current iteration (it_n) value at the referenced node. As long as instances with values that differ between iteration $it_{(n-1)}$ and it_n are identified, a new fixed-point iteration $it_{(n+1)}$ is triggered starting with the parents of the unstable reference nodes. For the first iteration it_0, all leaves are classified as unstable with the DFA initialization values representing $it_{(n-1)}$.

Figure 8(b) shows the result after the initial iteration with the highlighted nodes representing the executed rules. Since `is_reachable` at the model object *2* now differs from its previous value, the new result is transferred to the reference node. Its predecessor, the instance at model node *3*, is scheduled as starting point for bottom-up traversal in it_1. The stable fixed point is reached after iteration it_1, shown in Figure 8(c). Since the value for model object *1* has not changed, the traversal can be aborted without recalculation of *2* and *E*.

The discovery of new dependencies during the evaluation process can result in the introduction of additional nodes, the reconnection of existing nodes or the merging of previously separate graphs. To handle this case, the required modifications are postponed until after the current iteration it_n finishes. Then, an intermediate step $it_{n'}$ is carried out in which the existing graphs are extended by repeating the chain-building steps of phase 1 for the discovered attribute instances. For iteration $it_{(n+1)}$, re-evaluation is scheduled to start at the smallest set of leaf nodes that includes all newly created instances and nodes which introduced new dependencies to existing instances as parents.

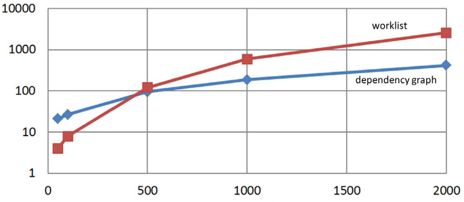
4 Evaluation

In this section we present our findings in the evaluation of the scalability of the fixed-point computation for models. Both the number of rule executions in relation to the amount of instances and the time for the analysis are indicators for its performance aspects. The goal is a qualitative assessment of the applicability of the approach for the analysis of large models. The evaluation employs the attributes defined in Section 3.2 - `is_reachable`, `all_predecessors` and `scc_id` - as well as `all_predecessors_min` which calculates the dominating sets, using equivalent bitvector-based implementations of the semantic rules. To evaluate the scalability with respect to the amount of instances, five models have been generated randomly to contain 50, 100, 500, 1000 and 2000 nodes. Except the start and the final node, each node has exactly two outgoing connections to arbitrary targets. Because each attribute is calculated for each node, the number of results therefore amounts to four times the number of nodes. The computation has been carried out with the algorithm described in Section 3.3 and a modified worklist algorithm that does not construct a dependency graph to demonstrate the unoptimized application of traditional DFA to the modeling context. The values represent the median of 90 of 100 analysis runs (to eliminate caching issues, the first 10% have been discarded) on an Intel i7 2,20GHz computer.



	50	100	500	1000	2000
dependency graph	614	1138	6026	11896	23951
worklist	826	1687	12029	37412	102020

Fig. 9. Number of rule executions



	50	100	500	1.000	2.000
dependency graph	21	27	95	189	419
worklist	4	8	121	596	2601

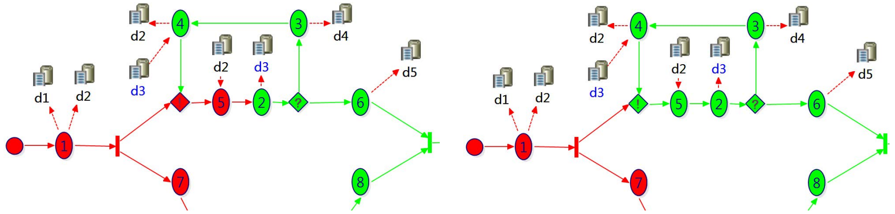
Fig. 10. Analysis time in ms (log. scale)

Figure 9 shows the total amount of rules executed in the fixed-point iterations. The time in milliseconds is pictured in Figure 10 using a logarithmic scale. From the results it can be deduced that while the worklist method is faster at a lower number of instances, it is soon outperformed by the dependency graph approach. This can be explained by the overhead induced by the complex data structures maintained by the graph-based algorithm. The dependency graph algorithm breaks even between 100 and 500 nodes (400-2000 instances) as the time and the amount of rule executions scales with the total number of results.

In the master thesis [24] our approach has been applied to detect illegal backward data dependencies in AUTOSAR[6] models. The author concludes that with an execution time of 2.4 seconds (including pre-analysis steps) for the TIMMO-2-USE breaking system use case, the "case study shows that the analysis tool is able to cope with medium sized systems".

5 Applications

The presented approach has been applied to different domains to verify its viability and versatility as a technique that supports a wide range of use cases. The open source Model Analysis Framework[7] (MAF, [25]), was developed as a proof-of-concept platform and a reference implementation. The tooling suite is built on top of Eclipse technology such as the Eclipse Modeling Framework, Xtext, MDT OCL and M2M QVT. It contains a DFA solver module which can be integrated into third party applications and an IDE that supports analysis specification, configuration and debugging.



(a) Minimal availability

(b) Maximal availability

Fig. 11. Minimal and maximal availability of *d3*

Multiple analyses (available from the MAF repository) have been implemented for Eclipse's Java Workflow Tooling[8] (JWT) project - a tooling suite for modeling executable business processes. In the process shown in Figures 11(a) and 11(b), resource objects have been assigned to business actions, outgoing arrows denoting the production and incoming arrows the use of a resource.

[6] http://www.autosar.org, http://www.timmo-2-use.org/

[7] http://code.google.com/a/eclipselabs.org/p/model-analysis-framework/

[8] http://www.eclipse.org/jwt/

Flow-analysis can now be used to detect whether resources will be available at steps where they are required as input. Fixed-point computation yields two results: We can track the propagation of resources assuming that all paths are taken ($\Delta = \bigcup$, maximal availability) and the case where only information is regarded arriving on all paths at once ($\Delta = \bigcap$, minimal availability). The latter case differs from the former if resources are created inside cycles or in diamond-shaped (i.e. alternative) paths because not every execution of such a process will traverse these paths. In the figures the availability of the resource *d3* is highlighted in green. The notable difference lies in node *5* where *d3* will only be available after the cycle has been traversed at least once. This is reflected in the minimal availability result depicted in Figure 11(a). It indicates that it cannot be guaranteed that the resource will be available at *5* on all executions of this process. On the other hand, from Figure 11(b) we can deduce that there is at least one path on which *d3* will have been created once we arrive at this point.

By combining this information with the local input/output of each node, the user can be given an indication about the validity of the process with respect to resource availability. This use case can be extended in multiple ways, e.g. to approximate how many instances of a resource must be provisioned if it can be accessed multiple times at once in parallel execution paths.

Additional use cases which are currently being evaluated include the detection of structural clones, the formalization of modeling guidelines and the computation of model metrics for different application domains (cf. [26]).

Currently, the Model Analysis Framework is also used in several research projects, including the ITEA2 project VERDE[9] and WEMUCS[10] (IuK Bayern).

VERDE employs state-machines to derive test cases in the notion of model-based testing (cf. Deliverables 5.3.1, 5.4.2). Subjecting them to static analysis therefore enables early feedback to the developer on whether a model conforms to its intended behavior. Specifically, DFA is used to compute edge coverage information to drive test path generation and to perform a variable analysis in the notion of compiler construction on the code embedded in the state machine's states and transitions. Results of static analysis are used to detect relevant test cases, e.g. paths where variables are accessed that might not have been initialized or adopt border case values. Applying static analysis to state machine models presents a unified approach that enables early violation detection and indication of potential problems as well as a focused test case generation.

The goal of the ongoing WEMUCS project is to provide methods and tools for the development, optimization and testing of software for embedded multi-core systems. The analysis of AUTOSAR models (cf. Section 4) is used to identify dependencies between functions (RunnableEntities) incurred by their data accesses. The dependencies detected using DFA are used to derive a valid execution order for the entities (or to ask for manual problem resolution if this is not

[9] Validation-driven design for component-based architectures, `http://www.itea-verde.org/`

[10] Methods and tools for iterative development and optimization of software for embedded multicore systems, `http://www.multicore-tools.de`

possible). Afterward, a DFA implementation of the token flow algorithm [27] is applied to the constructed control-flow graph to cluster the entities into single-entry-single-exit (SESE) components. These components represent parallelizable blocks and can subsequently be used as input for a scheduling algorithm.

6 Related Work

The canonical method for formalizing the static semantics of modeling languages is the Object Constraint Language which was recently extended with the ability to handle transitive closures[11]. However, as a constraint language it is not well suited for the derivation and approximation of context-sensitive information - a limitation removed by the fixed-point semantics of the data-flow method.

Several attempts were made to convert UML models with annotated OCL constraints to other technical spaces by translating constraints into satisfiability problems [28–30]. With the existence of powerful OCL interpreters, these methods are not strictly required for constraint evaluation, however in some cases they provide additional features, e.g. snapshot generation [31], to validate whether the semantics of the modeling language are preserved.

The relevance of flow-based analysis is evident from the amount of research work that employs DFA: The authors of [32] convert UML sequence diagrams to control-flow graphs for validation purposes while [33, 34] attempt to improve test case generation from statecharts. Def-use relationships for UML Action Semantics are derived in [35] and [36] applies DFA to identify patterns for translating graph-oriented BPMN models into block-oriented BPEL code. While originally given as an imperative algorithm, the SESE decomposition proposed in [27] was easily converted to a declarative flow analysis (cf. Section 5). It can be assumed that these methods could have profited from the presented approach as a unified method for defining flow-based analyses in their respective domains.

Although there are many usage scenarios for DFA in the modeling area, there exists - to our knowledge - only one approach that is directly comparable in that it provides a generalized technique for analysis specification and evaluation: JastEMF [37] translates meta models to circular reference attribute grammars (CRAG) [38], an extension of traditional attribute grammars, by mapping the containment hierarchy of the meta model to grammatical productions. Both the cross-references between meta model elements and semantic specifications (comparable to flow equations) are then defined as semantic attributes. CRAGs support fixed-point evaluation semantics through designated remote and circular attributes. Compared to the flow-analysis, this method strongly relies on the formalism of formal languages and attribute grammars, substituting the syntax tree with the model's containment tree to which the notion of attribute inheritance/synthesis is applied while the graph structure of the model has to be specified as part of the analysis in form of reference attributes.

[11] `http://www.omg.org/issues/issue13944.txt`

7 Conclusions and Outlook

In this paper we presented an approach for static model analysis in the notion of data-flow analysis, a well-understood technique from the field of compiler construction. The stated goal was to provide language engineers with a unified method for complementing (existing) model-based DSLs with static analysis capabilities. By validating well-formedness constraints and deriving static approximations of behavioral properties based on contextual, flow-sensitive information, many aspects of modeled systems can be evaluated on a conceptual level.

To motivate the applicability of flow analysis, we studied the relationships between the area of formal languages, in which this method is traditionally applied, and the field of modeling. Based on an alignment of the respective abstraction layers, we proposed an analysis specification DSL that transfers the underlying principles to the modeling domain. Because this language itself is model-based, it closely integrates with the target domain, eliminating the need for transformations between different technological and conceptual backgrounds and thus reducing the effort for implementation and usage. Since analyses are defined non-intrusively and arbitrary languages can be used to specify DFA equations, full compatibility with existing modeling languages and tools is retained and flexibility is provided with regard to adaption to diverse technological ecosystems.

As opposed to traditional DFA where dependencies between flow-equation instances are derived from the control flow itself, the ambiguous edge semantics in model graphs make the automatic propagation of results along these paths impractical. To overcome this problem, we use a demand-driven, iterative algorithm supporting the dynamic discovery of dependencies during solving. It allows for partial parallelization and its performance has been evaluated experimentally.

In conclusion, this approach provides the capabilities and the versatility required to implement sophisticated analyses - as demonstrated in the context of several use cases - along with a close integration with modeling concepts, namely the widely-used OMG standards. It provides a generic "programming language" for specifying declarative analyses that rely on an examination of flow-sensitive properties. The application range also extends to structural models, e.g. computing metrics for UML class diagrams such as the Attribute Inheritance Factor (AIF) relating the inherited attributes at a class to all available attributes [39].

Next steps include the examination of additional application areas and an evaluation of practical experiences with relation to the specification process. The solving algorithm will be complemented with a formalized description and in-depth evaluation. A "standard library" containing common flow analyses will be defined to serve as starting point for custom implementations.

References

1. Object Management Group. Model-Driven Architecture (June 2003), http://www.omg.org/mda/
2. Apfelbaum, L., Doyle, J.: Model based testing. In: Software Quality Week Conference, pp. 296–300 (1997)

3. Object Management Group (OMG) specifications, http://www.omg.org/spec
4. Malgouyres, H., Motet, G.: A UML model consistency verification approach based on meta-modeling formalization. In: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 1804–1809. ACM (2006)
5. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to alloy and back again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
6. Mandel, L., Cengarle, M.V.: On the expressive power of OCL. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, p. 854. Springer, Heidelberg (1999)
7. Baar, T.: The definition of transitive closure with OCL – limitations and applications. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 358–365. Springer, Heidelberg (2004)
8. Brucker, A.D., Doser, J., Wolff, B.: Semantic issues of OCL: Past, present, and future. Electronic Communications of the EASST 5 (2007)
9. Cengarle, M.V., Knapp, A.: A formal semantics for OCL 1.4. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 118–133. Springer, Heidelberg (2001)
10. Marković, S., Baar, T.: An OCL semantics specified with QVT. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 661–675. Springer, Heidelberg (2006)
11. Brucker, A.D., Wolff, B.: A proposal for a formal OCL semantics in isabelle/HOL. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 99–114. Springer, Heidelberg (2002)
12. Saad, C., Bauer, B.: Data-flow based model analysis. In: Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp. 227–231. NASA (April 2010)
13. Kildall, G.A.: A unified approach to global program optimization. pp. 194–206 (1973)
14. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems 2(2), 127–145 (1968)
15. Babich, W.A., Jazayeri, M.: The Method of Attributes for Data Flow Analysis. Acta Inf. 10, 245–264 (1978)
16. Rodney, F.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1986, pp. 85–98. ACM, New York (1986)
17. Jones, L.G.: Efficient evaluation of circular attribute grammars. ACM Trans. Program. Lang. Syst. 12(3), 429–462 (1990)
18. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
19. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2009)
21. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1995, pp. 104–115. ACM, NY (1995)
22. Ryder, B.G.: Constructing the call graph of a program. IEEE Transactions on Software Engineering (3), 216–226 (1979)
23. Jahromi, S.A.H.M., Honar, E.: A framework for call graph construction (2010)

24. Minnerup, P.: Models in the development process for parallelizing embedded systems. Master's thesis, Augsburg University, 86159 Augsburg, Germany (2012)
25. Saad, C., Bauer, B.: The Model Analysis Framework An IDE for Static Model Analysis. In: Industry Track of Software Language Engineering (ITSLE), 4th International Conference on Software Language Engineering (SLE 2011) (May 2011)
26. Baroni, A.L., Abreu, O.B.E.: An OCL-based formalization of the MOOSE metric suite. In: Proceedings of ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering (2003)
27. Götz, M., Roser, S., Lautenbacher, F., Bauer, B.: Token Analysis of Graph-Oriented Process Models. In: New Zealand Second International Workshop on Dynamic and Declarative Business Processes (DDBP), 13th IEEE International EDOC Conference (EDOC 2009) (September 2009)
28. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW 2008, pp. 73–80. IEEE (2008)
29. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2010)
30. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1341–1344. European Design and Automation Association (2010)
31. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 265–279. Springer, Heidelberg (2003)
32. Garousi, V., Bri, L., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams (2005)
33. Briand, L., Labiche, Y., Lin, Q.: Improving the coverage criteria of uml state machines using data flow analysis. Software Testing, Verification and Reliability 20(3), 177–207 (2010)
34. Kim, Y.G., Hong, H.S., Bae, D.-H., Cha, S.-D.: Test cases generation from uml state diagrams. IEEE Proceedings Software 146, 187–192 (1999)
35. Waheed, T., Iqbal, M.Z.Z., Malik, Z.I.: Data Flow Analysis of UML Action Semantics for Executable Models. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 79–93. Springer, Heidelberg (2008)
36. García-Bañuelos, L.: Pattern Identification and Classification in the Translation from BPMN to BPEL. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 436–444. Springer, Heidelberg (2008)
37. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference Attribute Grammars for Metamodel Semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 22–41. Springer, Heidelberg (2011)
38. Magnusson, E., Hedin, G.: Circular Reference Attributed Grammars - Their Evaluation and Applications. ENTCS 82(3) (2003)
39. Abreu, F.B., Carapuça, R.: Object-oriented software engineering: Measuring and controlling the development process. In: Proceedings of the 4th International Conference on Software Quality (1994)