Iterative and incremental model generation by logic solvers[⋆]

Oszkár Semeráth, András Vörös and Dániel Varró

Budapest University of Technology and Economics, Budapest, Hungary
Department of Measurement and Information Systems
{semerath,vori,varro}@mit.bme.hu

Abstract. The generation of sample instance models of Domain-Specific Language (DSL) specifications has become an active research line due to its increasing industrial relevance for engineering complex modeling tools by using large metamodels and complex well-formedness constraints. However, the synthesis of large, well-formed *and* realistic models is still a major challenge. In this paper, we propose an iterative process for generating valid instance models by calling existing logic solvers as black-box components using various approximations of metamodels and constraints to improve overall scalability. (1) First, we apply enhanced metamodel pruning and partial instance models to reduce the complexity of model generation subtasks and the retrieved partial solutions initiated in each step. (2) Then we propose an (over-)approximation technique for well-formedness constraints in order to interpret and evaluate them on partial (pruned) metamodels. (3) Finally, we define a workflow that incrementally generates a sequence of instance models by refining and extending partial models in multiple steps, where each step is an independent call to the underlying solver (the Alloy Analyzer in our experiments).

Keywords: domain-specific languages, logic solvers, model generation

1 Introduction

Motivation The generation of sample instance models of Domain-Specific Language (DSL) specifications has become an active research line due to its increasing industrial relevance for engineering complex modeling tools by using large metamodels (MM) and complex well-formedness (WF) constraints [25]. Such instance models derived as representative examples [2] and counterexamples [18,32] may serve as test cases or performance benchmarks for DSL modeling tools, model transformations or code generators [4]. Existing approaches dominantly use either a logic solver or a rule-based instance generator in the background.

[⋆] This paper is partially supported by the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems and by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

Problem statement *Model finding using logic solvers* [16] (like SMT or SAT-solvers) is an effective technique (1) to identify inconsistencies of a DSL specification or (2) to generate well-formed sample instances of a DSL. This approach handles *complex global WF constraints* which necessitates to access and query several model elements during evaluation. Model generation for graph structures needs to satisfy complex structural global constraints (which is typical characteristic for DSLs), which restricts the direct use of logical numerical and constraint solvers despite the existence of various encodings of graph structures into logic formulae. As the metamodel of an industrial DSL may contain hundreds of model elements, any realistic instance model should be of similar size. Unfortunately, this cannot currently be achieved by a single direct call to the underlying solver [17,32], thus existing logic based model generators *fail to scale.* Furthermore, logic solvers tend to retrieve simple *unrealistic models* consisting of unconnected islands of model fragments and many isolated nodes, which is problematic in an industrial setting.

Rule-based instance generators [4, 13, 33] are effective in generating larger model instances by independent modifications to the model by randomly applying mutation rules. Such a rule-based approach offers *better scalability* for complex DSLs. These approaches may incorporate *local WF constraints* which can be evaluated in the context of a single model element (or within its 1-context). However, they *fail to handle global WF constraints* which require to access and navigate along a complex network of model elements. Since constraint evaluation is typically the final step of the generation process, the synthesized models may violate several WF constraints of the DSL in an industrial setting.

Contribution The long term objective of our research is to synthesize large, well-formed *and* realistic models. In this paper, we propose an iterative process for incrementally generating valid instance models by calling existing logic solvers as black-box components using various abstractions and approximations to improve overall scalability. (1) First, we apply enhanced metamodel pruning [33] and partial instance models [32] to reduce the complexity of model generation subtasks and the retrieved partial solutions initiated in each step. (2) Then we propose an (over-)approximation technique for well-formedness constraints in order to interpret and evaluate them on partial (pruned) metamodels. (3) Finally, we define a workflow that incrementally generates a sequence of instance models by refining and extending partial models in multiple steps, where each step is an independent call to the underlying solver. We carried out experiments using the state-of-the-art Alloy Analyzer [16] to assess the scalability of our approach.

Added value Our approach increases the size of generated models by carefully controlling the information fed into and retrieved back from logic solvers in each step via abstractions. Each generated model (1) increases in size by only a handful number of elements, (2) satisfies all WF constraints (on a certain level of abstraction), and (3) it is realistic in the sense that each model is a single component (and not disconnected islands). The incremental derivation

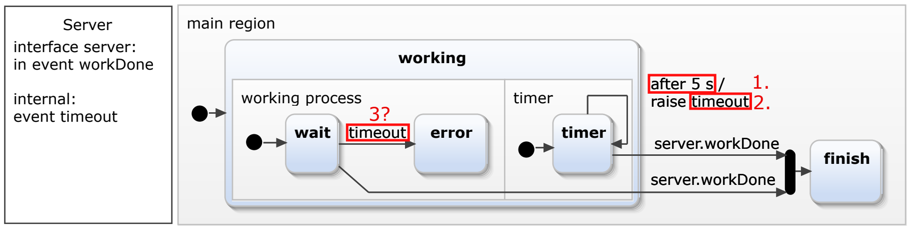

Fig. 1. Example Yakindu statechart with synchronisations.

of the result set provides graceful degradation, i.e. if the back-end solver fails to synthesize models of size N (due to timeout), all previous model instances are still available. From a practical viewpoint, the DSL engineer can influence or assist the instance generation process by selecting the important fragment of the analyzed metamodel (so called *effective metamodel* [4]). This is also common practice for testing model transformations or code generators.

Structure of the Report Next, Section 2 introduces some preliminaries for formalizing metamodels, constraints and partial snaptshots. Our approach is presented in Section 3 followed by an initial experimental evaluation in Section 4. Related work is assessed in Section 5 while Section 6 concludes our paper.

2 Preliminaries

In this section we present an overview of model generation with logic solvers with a running case study of Yakindu statecharts. Yakindu Statecharts Tools [37] is an industrial integrated modeling environment developed by Itemis AG for the specification and development of reactive, event-driven systems based on the concept of statecharts captured in combined graphical and textual syntax. Yakindu simultaneously supports static validation of well-formedness constraints as well as simulation of (and code generation from) statechart models. A sample statechart is illustrated in Figure 1. Yakindu provides two types of synchronization mechanisms: explicit synchronization nodes (marked as black rectangles) and event-based synchronization (i.e. raising and consuming events).

Validation is crucial for domain-specific modelling tools to detect conceptual design flaws early and ensure that malformed models does not processed by tooling. Therefore missing validation rules are considered as bugs of the editor. While Yakindu is a stable modeling tool, it is still surprisingly easy to develop model instances as corner cases which satisfy all (implemented) well-formedness constraints of the language but crashes the simulator or code generator due to synchronization issues. One of such problems is depicted in Figure 1 where (1) after 5 seconds a (2) timeout event raised in region timer, but (3) it cannot be accepted in state wait in the simulator and in the generated code.

Our goal is to systematically synthesize such model instances by using logic solvers in the background by mapping DSL specifications to a logic problem [17,32]. Such model generation approach usually takes three inputs: (1) a *metamodel of the domain* (Section 2.1), (2) a set of *well-formedness constraints* of the language (Section 2.2), and optionally (3) a *partial snapshot* (Section 2.3) serving as an initial seed which generated models need to contain.

2.1 Domain Metamodel

Metamodels define the main concepts, relations and attributes of the target domain to specify the basic structure of the models. In this paper, the Eclipse Modeling Framework (EMF) is used for domain modeling, which is dominantly used in many industrial DSL tools and modeling environments. The main concepts are illustrated using Yakindu state graph metamodel [37] in Figure 2.

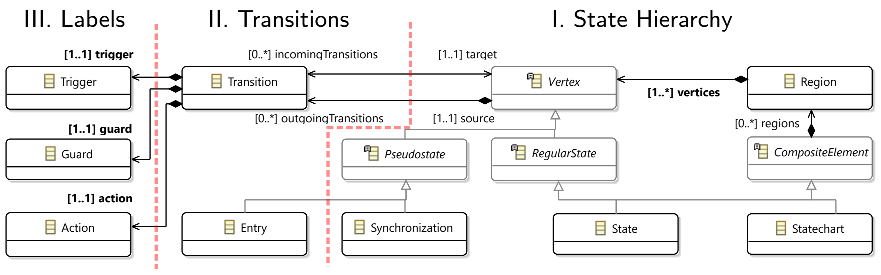

Fig. 2. Metamodel extract from Yakindu state machines

A state machine consists of Regions, which in turn contain states (called Vertexes) and Transitions. An abstract state Vertex is further refined into RegularStates (like State) and PseudoStates like Entry and Synchronization states. Note that we intentionally kept the generalization hierarchy unchanged and simplified the original metamodel only by removing some elements. Metamodel elements are mapped to a set of logic relations as defined in [17,32]:

- **Classes (CLS)**: In EMF, *EClass*es can be instantiated to *EObjects*, where the set of objects of a model is denoted by *objects*. Whether an o is an instance of a class C it is denoted as $\mathsf{C}(o)$.
- **References (REF)**: *EReference*s between classes S and T capture a binary relation $R(S,T)$ of the metamodel. When two objects o and t are in a relation R, an *EReference* is instantiated leading from o to t denoted as $\mathsf{R}(o,t)$.
- **Attributes (ATT)**: *EAttributes* enrich a class C with values of predefined primitive types like integers, strings, etc by binary relations $A(C,V)$. If an object o stores a value v as attribute A it is denoted as $\mathsf{A}(o,v)$.

Further structural restrictions implied by a metamodel (and formalized in [32]) include (1) **Generalization (GEN)** which expresses that a more specific (child) class has every structural feature of the more general (parent) class, (2) **Type compliance (TC)** that requires that for any relation $\mathsf{R}(o, t)$, its source and target objects o and t need to have compliant types, (3) **Abstract (ABS)**: If a class is defined as *abstract*, it is not allowed to have direct instances, (4) **Multiplicity (MUL)** of structural features can be limited with upper and lower bound in the form of "lower..upper" and (5) **Inverse (INV)**, which states that two parallel references of opposite direction always occur in pairs. Finally EMF instance models are expected to be arranged into a *containment hierarchy*, which is a directed tree along relations marked in the metamodel as containment (e.g. regions or vertices).

An instance model M is an instance of a metamodel $Meta$ (denoted with $M \models Meta$) if all the corresponding constraints above are satisfied, i.e. $Meta = CLS \wedge REF \wedge \cdots \wedge MUL \wedge INV$ [32]. Therefore a model generation task for a given size s and a metamodel $Meta$ can be solved as logic problem, where the solver creates an interpretation for all class predicates, all reference and attribute relations over the set of $objects = \{o_1, \ldots, o_s\}$ and sets of enum literals, which satisfies all structural constraints.

2.2 Well-formedness Constraints

Structural well-formedness (WF) constraints (aka design rules or consistency rules) complement metamodels with additional restrictions that have to be satisfied by a valid instance model (in our case, statechart model). Such constraints are frequently defined by graph patterns [36] or OCL invariants [27]. To abstract from the actual constraint language, we assume in the paper that WF constraints are defined in first order logic. Given a set WF of well-formedness constraints, a model M is called valid if $M \models Meta \wedge WF$.

Example The Yakindu documentation states several constraints for statecharts including the following ones regulating the use of synchronization states. (Abbreviated names of classes and references are used as predicates).

Φ_1 Source states of a synchronization have to be contained in different regions!
$\forall syn, s_1, s_2, t_1, t_2, r_1, r_2 :$
$(\mathsf{Synchron}(syn) \wedge \mathsf{outgoing}(s_1, t_1) \wedge \mathsf{outgoing}(s_2, t_2) \wedge \mathsf{target}(t_1, syn) \wedge$
$\mathsf{target}(t_2, syn) \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r_2, s_2) \wedge s_1 \neq s_2) \Rightarrow r_1 \neq r_2$

Φ_2 Source states of a synchronization are contained in the same parent state!
$\forall syn, s_1, s_2, t_1, t_2, r_1, r_2 \exists p :$
$(\mathsf{Synchron}(syn) \wedge \mathsf{outgoing}(s_1, t_1) \wedge \mathsf{outgoing}(s_2, t_2) \wedge \mathsf{target}(t_1, syn) \wedge$
$\mathsf{target}(t_2, syn) \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r_2, s_2) \wedge s_1 \neq s_2)$
$\Rightarrow (\mathsf{regions}(p, r_1) \wedge \mathsf{regions}(p, r_2))$

Φ_3 Target states of a synchronization have to be contained in different regions!
$\forall syn, s_1, s_2, t_1, t_2, r_1, r_2 :$
$(\mathsf{Synchron}(syn) \wedge \mathsf{incoming}(s_1, t_1) \wedge \mathsf{incoming}(s_2, t_2) \wedge \mathsf{source}(t_1, syn) \wedge$
$\mathsf{source}(t_2, syn) \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r_2, s_2) \wedge s_1 \neq s_2) \Rightarrow r_1 \neq r_2$

Φ_4 Target states of a synchronization are contained in the same parent state!
$\forall syn, s_1, s_2, t_1, t_2, r_1, r_2 \exists p :$
$(\mathsf{Synchron}(syn) \wedge \mathsf{incoming}(s_1, t_1) \wedge \mathsf{incoming}(s_2, t_2) \wedge \mathsf{source}(t_1, syn) \wedge$
$\mathsf{source}(t_2, syn) \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r_2, s_2) \wedge s_1 \neq s_2)$
$\Rightarrow (\mathsf{regions}(p, r_1) \wedge \mathsf{regions}(p, r_2))$

Φ_5 A synchronization shall have at least two incoming or outgoing transitions!
$\forall syn : \mathsf{Synchron}(syn) \Rightarrow \exists t_1, t_2 : t_1 \neq t_2 \wedge ($
$(\mathsf{incoming}(t_1, syn) \wedge \mathsf{incoming}(t_2, syn)) \vee (\mathsf{outgoing}(t_1, syn) \wedge \mathsf{outgoing}(t_2, syn)))$

2.3 Partial Snapshots

Partial Snapshots (PS) specify required instance model fragments of a metamodel [32]. A partial snapshot is a model constructed from the same classes and relations as a valid instance model. Formally, a PS satisfies the constraints *CLS*, *GEN*, *REF* and *TC*, but it possibly violates *ABS*, *ATT*, *MUL* and *INV*, which means that even abstract classes can be instantiated, and multiplicity constraints, the inverse relation of references and containment hierarchy rules might be violated. If a PS is a partial snapshot of a metamodel it is denoted by $PS \models_P Meta$.A model M contains a partial snapshot PS (denoted with $M \models PS$) if there is a morphism $m : PS \rightarrow M$ (composed of a pair of morphisms $objects_{PS} \rightarrow objects_M$ and $references_{PS} \rightarrow references_M$ for mapping objects and references) which satisfies the following constraints for each $o_1, o_2 \in objects_{PS}$:

1. m is injective: $o_1 \neq o_2 \Rightarrow m(o_1) \neq m(o_2)$
2. For each class C the mapping preserves the type: $\mathsf{C}(o_1) \Rightarrow \mathsf{C}(m(o_1))$
3. For each reference R the mapping preserves the source and the target of the reference: $\mathsf{R}(o_1, o_2) \Rightarrow \mathsf{R}(m(o_1), m(o_2))$
4. For each attribute A the mapping preserves the attribute value v and the location: $\mathsf{A}(o_1, v) \Rightarrow \mathsf{A}(m(o_1), v)$

A partial snapshot can be generalized from a regular (fully specified) instance model by relaxing specific properties identified by the DSL developer [32] to guide testing in practical cases. In the current paper, we create partial snapshots by iteratively reusing the instance models generated in a previous run to achieve incremental model generation (see Section 3.3).

3 Incremental Model Generation by Approximations

Despite the precise definition of logic formulae for our statechart language using existing mappings [32], a major practical drawback is that a direct (single step) model generation using Z3 or Alloy as back-end solver only terminates for very small model sizes. If we aim to improve scalability by omitting certain constraints, the synthesized models are no longer well-formed thus they cannot be fed into Yakindu as sample models.

To increase the size of synthesized models while still keeping them well-formed, we propose an incremental model generation approach (Section 3.3) by iterative calls to backend solvers exploiting two enabling techniques of metamodel pruning (Section 3.1) and constraint approximation (Section 3.2).

$$\begin{tikzcd}
Meta \arrow[r, phantom, "\Rightarrow"] \arrow[d, "model"', "+snapshot"', ""{name=A}] & \arrow[l, phantom, "prune"] Meta_p \arrow[d, "model"] \\
M \arrow[r, phantom, "\Rightarrow"] & M_p \arrow[to=A, bend right] \arrow[l, phantom, "snapshot"']
\end{tikzcd}$$

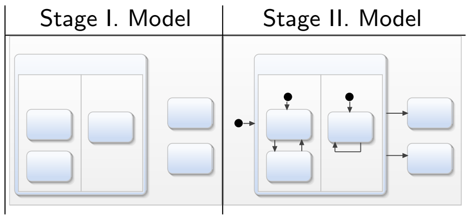

Fig. 3. Metamodel pruning with overapproximation

3.1 Metamodel Pruning

Metamodel pruning [13, 33] takes a metamodel $Meta$ as input and derives a simplified (pruned) metamodel $Meta_P$ as output by removing some *EClasses*, *EReferences* and *EAttributes*. When removing a class from a metamodel, we need to remove all subclasses, all attributes and incoming or outgoing references to obtain a consistent pruned metamodel. Formally, we may iteratively remove certain predicates from $Meta$ by pruning as follows:

- **EReference**: if $R(S,T) \in Meta$ then $R(S,T) \notin Meta_P$;
- **EAttributes**: if $A(C,V) \in Meta$ then $A(C,V) \notin Meta_P$;
- **EClasses**: if $C \in Meta$ and $sub(C,Sub) \notin Meta_P$ and $A(C,V) \notin Meta_P$ and $R(C,T) \notin Meta_P$ and $R(S,C) \notin Meta_P$ then $C \notin Meta_P$;

Example We prune our statechart metamodel in two phases (see the slices in Figure 2): classes Trigger, Guard and Action are omitted together with incoming references (Stage II), and then classes Transition, Pseudostate, Entry and Synchronization are removed (Stage I).

By using metamodel pruning, we first aim to generate valid instance models for the pruned metamodel and then extend them to valid instance models of the original larger metamodel. For that purpose, we exploit a property we call the *overapproximation property of metamodel pruning* (see Figure 3), which ensures that if there exist a valid instance model M for a metamodel *Meta* (formally, $M \models Meta$) then there exists a valid instance model M_P for the pruned metamodel $Meta_P$ (formally, $M_P \models Meta_P$) such that M_P is a partial snapshot of M ($M_P \subseteq M$). Consequently, if a model generation problem is unsatisfiable for the pruned metamodel, then it remains unsatisfiable for the larger metamodel. However, we may derive a pruned instance model M_P which cannot be completed in the full metamodel *Meta*, which is called a *false positive*.

Example The statechart model in the middle of Figure 3 corresponds to the pruned metamodel after Stage II. In our example, it can be extended by adding transitions and entry states to the model illustrated in the right side of Figure 3, which now corresponds to the pruned metamodel of Stage I.

$$R(x)^O = \begin{cases} R(x) & \text{if } R \in \mathit{Meta}_P \\ \mathit{true} & \text{else} \end{cases} \qquad \begin{aligned} (\neg\Phi)^O &= \neg(\Phi^U) \\ (\neg\Phi)^U &= \neg(\Phi^O) \\ (\Phi_1 \wedge \Phi_2)^O &= {\Phi_1}^O \wedge {\Phi_2}^O \\ (\Phi_1 \wedge \Phi_2)^U &= {\Phi_1}^U \wedge {\Phi_2}^U \\ (\Phi_1 \vee \Phi_2)^O &= {\Phi_1}^O \vee {\Phi_2}^O \\ (\Phi_1 \vee \Phi_2)^U &= {\Phi_1}^U \vee {\Phi_2}^U \end{aligned} \qquad \begin{aligned} (\exists x : \Phi(x))^O &= \exists x : \Phi(x)^O \\ (\exists x : \Phi(x))^U &= \exists x : \Phi(x)^U \\ (\forall x : \Phi(x))^O &= \forall x : \Phi(x)^O \\ (\forall x : \Phi(x))^U &= \forall x : \Phi(x)^U \end{aligned}$$

$$R(x)^U = \begin{cases} R(x) & \text{if } R \in \mathit{Meta}_P \\ \mathit{false} & \text{else} \end{cases}$$

Fig. 4. Constraint pruning and approximation

3.2 Constraint Pruning and Approximation

When removing certain metamodel elements by pruning, related structural constraints (such as multiplicity, inverse, etc.) can be automatically removed, which trivially fulfills the overapproximation property. However, the treatment of additional well- formedness constraints needs special care since simple automated removal would significantly increase the rate of false positives in a later phase of model generation to such an extent that no intermediate models can be extended to a valid final model.

Based on some first-order logic representation of the constraints (derived e.g. in accordance with [32]), we propose to maintain approximated versions of constraint sets during metamodel pruning. In order to investigate the interrelations of constraints, we assume that logical consequences of a constraint set can be derived manually by experts or automatically by theorem provers [21]. The actual derivation approach falls outside the scope of the current paper. Given a DSL specification with a metamodel *Meta* and a set of WF constraints $\mathit{WF} = \{\Phi_1, \ldots, \Phi_n\}$, let Φ be a formula derived as a theorem $\mathit{WF} \vdash \Phi$.

Now an *overapproximation* of formula Φ over metamodel *Meta* for a pruned metamodel Meta_P is a formula Φ_P such that (1) $\Phi \Rightarrow \Phi_P$, (2) Φ_P contains symbols only from Meta_P. The details of approximation are illustrated in Figure 4 where R denotes a relation symbol derived for class or reference predicates in accordance with the metamodel. While more precise approximations can possibly be defined in the future, the current approximation is logically correct as if a model generation problem is unsatisfiable for an approximated set of constraints (over the pruned metamodel) then it remains unsatisfiable for the original set of constraints.

Example Based on the set of WF constraints $\{\Phi_1, \Phi_2, \Phi_3, \Phi_4, \Phi_5\}$ defined in Section 2.2, a prover can derive the following formula as a theorem over the metamodel of Stage II: $\Phi_{syncout} \vee \Phi_{syncin}$, where $\Phi_1, \Phi_5 \models \Phi_{syncout} \vee \Phi_{syncin}$. The generated theorem $\Phi_{syncout}$ (and Φ_{syncin}) restricts the number of outgoing (ingoing) transitions from (to) a synchronization as follows:

$\Phi_{syncout} = \forall syn \exists \underline{t_1, t_2}, s_1, r_1, r_2, p : \mathsf{Synchron}(syn) \Rightarrow$
$\underline{(\mathsf{outgoing}(syn, t_1)} \wedge \underline{\mathsf{target}(t_1, s_1)} \wedge \underline{\mathsf{outgoing}(syn, t_2)} \wedge \underline{\mathsf{target}(t_2, s_2)} \wedge s_1 \neq s_2 \wedge$
$\mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r2, s2) \wedge r_1 \neq r_2 \wedge \mathsf{regions}(p, r1) \wedge \mathsf{regions}(p, r2))$

The variables and relations approximated in this phase are underlined: in Stage I the generation is restricted to the model by omitting transitions. The result of overapproximation states that if a model contains a synchronization, then needs to contain at least two regions:

$\Phi^O_{syncout} \vee \Phi^O_{syncin} = \forall syn \exists s_1, r_1, r_2, p : \mathsf{Synchron}(syn) \Rightarrow$
$(s_1 \neq s_2 \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r2, s2) \wedge r_1 \neq r_2 \wedge \mathsf{regions}(p, r1) \wedge \mathsf{regions}(p, r2))$

Applying the approximation rules of Figure 4 directly on $\{\Phi_1, \Phi_5\}$ would lead to $\Phi^O_1 : true$ and $\Phi^O_5 : true$. These constraints are too coarse overapproximations providing no useful information to the model generator at this phase.

3.3 Incremental Model Generation by Iterative Solver Calls

By using metamodel pruning, we first aim to generate valid instance models for the pruned metamodel, which is a simplified problem for the underlying logic solver. Instance models of increasing size will be gradually generated by using valid models of the pruned metamodel as partial snapshots (i.e. initial seeds) for generating instances for a larger metamodel. Therefore, an incremental model generation task is also given with a target size s and a target metamodel *Meta*, but with an additional partial snapshot M_P. M_P is a valid instance of pruned metamodel $Meta_P$. M_P has s_P number of objects ($s_P \leq s$).

From a logic perspective, the partial snapshot defines a partial interpretation of relations for model generation, which may simplify the task of the solver compared to using fully uninterpreted relations. In order to exploit this additional information, the relations in the logic problem are partitioned into two sets of interpreted and uninterpreted symbols. $objects_P = \{o_1, \ldots, o_{s_P}\}$ are the objects in the partial snapshot. The extra objects to be generated in this step are denoted by $objects_N = \{o_{s_P+1}, \ldots, o_s\}$. The relations are partitioned according to the following rules:

- **Classes (CLS)**: Each class predicate $\mathsf{C}(o)$ in *Meta* is separated into two: a fully interpreted $C_O(o)$ predicate for the objects in the partial snapshot $objects_P$, and an uninterpreted $C_N(o)$ for the newly generated objects $objects_N$. Therefore an object o is instance of a class C in the generated model if $C_O(o) \vee C_N(o)$ is satisfied. If the class is not in the pruned metamodel ($C \notin Meta_P$) then $C_O(o)$ is to be omitted, and if no new elements are created from a class then $C_N(o)$ can be omitted.
- **References (REF)**: Each reference predicate $\mathsf{R}(o, t)$ is separated into four categories: a fully interpreted $R_{OO}(o, t)$ between the objects of the partial snapshot ($objects_P$), an uninterpreted $R_{NN}(o, t)$ between the objects of the newly created objects ($objects_N$), and two additional uninterpreted relations $R_{ON}(o, t)$ and $R_{NO}(o, t)$ connecting the elements of the partial snapshot with the newly created elements (relations over $objects_O \times objects_N$ and $objects_N \times objects_O$ respectively). Therefore a reference $R(o, t)$ exists in the generated model if $R_{OO}(o, t) \vee R_{NN}(o, t) \vee R_{NO}(o, t) \vee R_{ON}(o, t)$. If the relation is not in the pruned metamodel ($R \notin Meta_P$) then $R_{OO}(o, t)$ can be omitted, and if no new elements are created from a class then $R_{NN}(o, t)$, $R_{NO}(o, t)$ and $R_{ON}(o, t)$ can also be omitted.

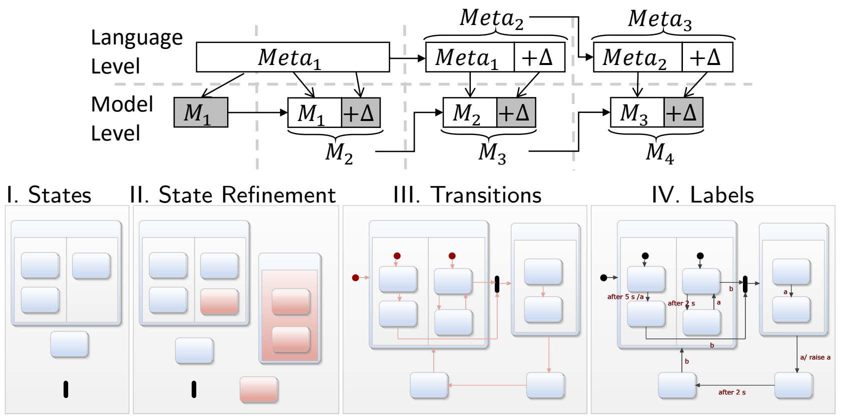

Fig. 5. Model generation iterations

– **Attributes (ATT)**: Attribute predicates are separated into a fully interpreted $A_O(o, v)$ for the objects in the partial snapshots $objects_P$, and an uninterpreted relation $A_N(o, v)$ for the newly created elements $objects_N$. An object o has an attribute value v $(A(o, v))$ if $A_O(o, v) \vee A_N(o, v)$. Attribute predicates are treated as reference predicates for omission.

The level of incrementality is still unfortunately limited from an important aspect. The background solvers typically provide no direct control over the simultaneous creation of new elements, i.e. we cannot provide domain- specific hints to the solver when the creation of an object always depends on the creation or existence of another object. This can still cause issues when a multitude of WF constraints are defined.

Example In our running example, the instance models are generated in four steps, which is illustrated in Figure 5. First, initial seeds are generated for the state hierarchy (M_1 over $Meta_1$), which are extended in the second step to model M_2 with the same metamodel elements. Then the metamodel is extended to $Meta_2$, and the transitions and the initial states are added to model M_3. Finally, triggers, guards and actions can be added to the model to obtain M_4.

4 Measurements

In order to assess the effectiveness of incremental model generation using constraint approximation for synthesizing well-formed instance models for domain-specific languages, we conducted some initial experiments using the Alloy Analyzer as background solver. We were interested in the following questions:

– Is incremental model generation with metamodel pruning and constraint approximation effective in increasing the size of models, the success rate or decreasing the runtime of the solver?

– Is incremental model generation still effective if metamodel pruning or constraint approximation is excluded?

Configurations We conducted measurements on two versions of the Yakindu statechart metamodel: Phase 1 and Phase 2 (see Figure 2). The pruned metamodel of Phase 1 (*MM1*) contains 8 classes and 2 references, and no well-formedness constraints by default. The metamodel of Phase 2 (*MM2*) contains 10 classes, 4 references and 8 constraints (including the 5 WF constraints listed in the paper and 3 more for restricting entry states).

– As a **base** configuration, the Alloy Analyzer is executed separately for the two problems with 1 minute timeout. We record two cases: the largest model derived and a slightly larger model size where timeout was observed.
– Next, we run the solver incrementally with an initial model of size N and an increment of size K denoted as $N + K$ in Figure 6 **without constraint approximation** but with metamodel pruning. Moreover, instance models derived for Phase 1 are used as partial snapshots for Phase 2.
– Then we run the solver incrementally with constraint approximation but **without metamodel pruning**. For that purpose, the constraint set for Phase 1 constains two approximated constraints: (1) Each region has a state where the entry state will point, and (2) There are orthogonal states in the model. Again, instance models derived for Phase 1 are used as partial snapshots for Phase 2, but the full metamodel is considered in Phase 2.
– Finally we configure the solver for **full** incrementally with constraint approximation and metamodel pruning by reusing instances of Phase 1 as partial snapshots in Phase 2.

Measurement setup Each model generation task was executed on the DSL presented in this paper 5 times using the Alloy Analyzer (with SAT4j- solver), then the median of the execution times was calculated. The measures are executed with one minute timeout on an average personal computer[1]. We measure the *runtime* of model generation, the *model size* denoting the maximal number of elements the derived model may contain, and the *success rate* denoting the percentage of cases when a well-formed model was derived, which satisfies all WF constraints within the given search scope.

Measurement results Results of our measurements are summarized in Figure 6. We summarize our observations below.

– **Base:** For *MM1*, Alloy was able to generate models with up to 60 objects. As there are no constraints at this level, many synchronizations are created (about half of the objects were synchronization and with only 5-10 states).

[1] CPU: Intel Core-i5-m310M, MEM: 16GB but the back-end solver can use 4GB only, OS: Windows 10 Pro, Reasoner: Alloy Analyzer 4.2 with sat4j

				MM1			MM2		
				#CLS:X	#REF:Y	#WF:Z	#CLS:X	#REF:Y	#WF:Z
				8	2	0 + 2	10	4	8
	Incre-mental	MM Pruning	Constraint Approx	Runtime (ms)	Model size (#)	Success rate (%)	Runtime (ms)	Model size (#)	Success rate (%)
Base	No	No	No	18349	60	100%	39040	12	0%
				Timeout	70	N/A	Timeout	16	N/A
W/o Prune	Yes	No	Yes	7327 + 11176	50+50	100%	Timeout	16	N/A
W/o Approx	Yes	Yes	No	12600+34804	50+50	100%	230 + 183465	20+30	0%
Full	Yes	Yes	Yes	7327 + 11176	50+50	100%	1644 + 44362	20+30	100%

Fig. 6. Measurement results

Over 60 objects, the runtime grows rapidly as the SAT solver runs out of the maximal 4 GB memory. For *MM2*, Alloy was unable to create any models that satisfies all of the constraints as the search scope turned out to be too small to create valid models with synchronizations.

- **W/o approx** Alloy was able to generate models with 100 elements in two steps where each iterative step had comparable runtime. However, since no constraints are considered for *MM1*, Alloyed failed to extend partial snapshots of *MM1* to well-formed models for *MM2* (success rate: 0%, although for this specific case, we executed over 100 runs of the solver due to the unexpectedly low success rate). Furthermore, we had to reduce the scope of search to 20 and 30 new elements with types taken from *MM2* $\setminus$ *MM1* due to timeouts.
- **W/o prune** When metamodel pruning was excluded but approximated constraints were included for *MM1*, model generation succeeded for 100 elements, but extending them to models of *MM2* failed (as in this case, new elements could take any elements from *MM2*)
- **Full** With incremental model generation by combining metamodel pruning and constraint approximation, we were able to generate well-formed models for both *MM1* and *MM2*, which was the only successful case for the latter.

Analysis of results While we used a reasonably sized statechart metamodel extracted from a real modeling tool (including everything to model state machines, but excluding imports and namespacing), we avoid drawing generic conclusions for the exact scalability of our results. Instead, we summarize some negative results which are hardly specific to the chosen example:

- Mapping a model generation problem to Alloy and running the Alloy Analyzer in itself will likely fail to derive useful results for practical metamodels, especially, in the presence of complex well-formedness constraints. Our observation is that many objects need to be created at the same time in consistent way, which cannot be efficiently handled by the underlying solver (either the scope is too small or out-of-memory). Altogether, the Alloy Analyzer was more effective in finding consistent model instance than proving that a problem is inconsistent, thus there are no solutions.

		Logic Solvers	Uncertain Models	Rule-Based Generators	Iterative Solver Call
Inputs	Partial Snapshot	+	++	-	+
	Effective Metamodel	-	-	+	+
	Local Constraints	+	-	+	+
	Global Constraints	+	-	-	+
Outputs	Metamodel-compliant	+	+	+	+
	Well-formed	+	-	-	+
	Diverse	-	-	+	?
	Scalable	-	-	++	+/-
	Decidability	-	+	+	- (graceful degradation)

Table 1. Comparison of related approaches

– An incremental approach with metamodel pruning but without constraint approximation will increase the overall size of the derived models, but the false positive rate would quickly increase.
– An incremental approach without metamodel pruning but with constraint approximation will likely have the same pitfalls as the original Alloy case: either the scope of search will become insufficient, or we run out of memory.
– Combining incremental model generation with metamodel pruning and constraint approximation is promising as a concept as it significantly improved wrt. the baseline case. But the underlying solver was still not sufficiently powerful to guarantee scalability for complex industrial cases.

5 Related Work

We compared our solution with existing model generation techniques with respect to the characteristics of *inputs* and *output results* in Table 1. As for *inputs*, the model generation can be (1) initiated from a *partial snapshot*, (2) focused on an *effective metamodel*. Additionally, an approach may support (3) *local* and (4) *global constraints* well-formedness constraints: a local constraint accesses only the attributes and the outgoing references of an object, while a global constraint specifies a complex structural pattern. Local constraints are frequently attached to objects (e.g. in UML class diagrams), while global constraints are widely used in domain-specific modeling languages. As *outputs*, the generated models may (i) be metamodel-compliant (ii) satisfy all *well-formedness* constraints of the language. When generated models are intended to be used as test cases, some approaches may guarantee a certain level of coverage or (iii) *diversity*. We consider a technique (iv) *scalable* if there is no hard limit on the model size (as demonstrated in the respective papers). Finally, a model generation approach may be (v) *decidable* which always terminates with a result. Our comparison excludes approaches like which do not guarantee metamodel- compliance of generated instance models.

Logic Solver Approaches Several approaches map a model generation problem (captured by a metamodel, partial snapshots, and a set of WF constraints) into a logic problem, which are solved by underlying SAT/SMT-solvers. Complete frameworks with standalone specification languages include Formula [17] (which uses Z3 SMT- solver [26]), Alloy [16] (which relies on SAT solvers like Sat4j [23]) and Clafer [2] (using backend reasoners like Alloy).

There are several approaches aiming to validate standardized engineering models enriched with OCL constraints [14] by relying upon different back-end logic-based approaches such as constraint logic programming [6,8,9], SAT-based model finders (like Alloy) [1,7,22,34,35], first-order logic [3], constructive query containment [28], higher-order logic [5,15], or rewriting logics [10].

Partial snapshots and WF constraints can be uniformly represented as constraints [32], but metamodel pruning is not typical. Growing models are supported in [19] for a limited set of constraints. Scalability of all these approaches are limited to small models / counter-examples. Furthermore, these approaches are either a priori bounded (where the search space needs to be restricted explicitly) or they have decidability issues.

The main difference of our current approach is its *iterative derivation of models* and the *approximative handling of metamodels and constraints*. However, our approach is independent from the actual mapping of constraints to logic formulae, thus it could potentially be integrated with most of the above techniques.

Uncertain Models Partial models are also similarity to uncertain models, which offer a rich specification language [12,29] amenable to analysis. Uncertain models provide a more expressive language compared to partial snapshots but without handling additional WF constraints. Such models document semantic variation points generically by annotations on a regular instance model, which are gradually resolved during the generation of concrete models. An uncertain model is more complex (or informative) than a concrete one, thus an a priori upper bound exists for the derivation, which is not an assumption in our case.

Potential concrete models compliant with an uncertain model can synthesized by the Alloy Analyzer [31], or refined by graph transformation rules [30]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from the respective papers, but refinement of uncertain models is always decidable.

Rule-based Instance Generators A different class of model generators relies on rule-based synthesis driven by randomized, statistical or metamodel coverage information for testing purposes [4,13]. Some approaches support the calculation of effective metamodels [33], but partial snapshots are excluded from input specifications. Moreover, WF constraints are restricted to local constraints evaluated on individual objects while global constraints of a DSL are not supported. On the positive side, these approaches guarantee the diversity of models and scale well in practice.

Iterative approaches. An iterative approach is proposed *specifically for allocation problems* in [20] based on Formula. Models are generated in two steps to increase diversity of results. First, non-isomorphic submodels are created only from an effective metamodel fragment. Diversity between submodels is achieved by a problem-specific symmetry-breaking predicate [11] which ensures that no isomorphic model is generated twice. In the second step the algorithm completes the different submodels according to the full model, but constraints are only checked at the very final stage. This is a key difference in our approach where an approximation of constraints is checked at each step, which reduces the number of inconsistent intermediate models. An iterative, counter-example guided synthesis is proposed for higher-order logic formulae in [24], but the size of derived models is fixed.

6 Conclusion and Future Work

The validation of DSL tools frequently necessitates the synthesis of well-formed *and* realistic instance models, which satisfy the language specification. In the paper, we proposed an incremental model generation approach which (1) iteratively calls black- box logic solvers to guarantee well-formedness by (2) feeding instance models obtained in a previous step as partial snapshots (compulsory model fragments) to a subsequent phase to limit the number of new elements, and using (3) various approximations of metamodels and constraints. Our initial experiments show that significantly larger model instances can be generated with the same solvers using such an incremental approach especially in the presence of complex well-formedness constraints.

However, part of our experimental results are negative in the sense that the proposed iterative approach is still not scalable to derive large model instances of complex industrial languages due to restrictions of the underlying Alloy Analyzer and the SAT solver libraries. We believe that dedicated decision procedures and heuristics for graph models would be beneficial in the long run to improve the performance of model generation.

As future work, we aim to generate a structurally diverse set of test cases by enumerating different possible extensions of a partial snapshot in each iteration step. Additionally, we plan to check other underlying solvers and further approximations and strategies for deriving relevant formulae as logical consequences of constraints. And finally, we will investigate if the metamodel partitions and the iteration steps can be automatically created, thus creating a (semi-)automated process with improved DSL-specific heuristics.

References

1. 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)
2. Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: unifying class and feature modeling. Software & Systems Modeling pp. 1–35 (2013)
3. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into First-order Predicate Logic. In: Proc. of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark (2002)
4. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In: 17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06. pp. 85–94 (Nov 2006)
5. Brucker, A.D., Wolff, B.: The HOL-OCL tool (2007), `http://www.brucker.ch/`
6. Büttner, F., Cabot, J.: Lightweight string reasoning for OCL. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Störrle, H., Kolovos, D.S. (eds.) Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings. LNCS, vol. 7349, pp. 244–258. Springer (2012)
7. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: 14th International Conf. on Formal Engineering Methods,ICFEM'12. pp. 198–213. LNCS 7635, Springer (2012)
8. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conf. on. pp. 73–80 (April 2008)
9. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). pp. 547–548. ACM, New York, NY, USA (2007)
10. Clavel, M., Egea, M.: The ITP/OCL tool (2008), `http://maude.sip.ucm.es/itp/ocl/`
11. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. KR 96, 148–159 (1996)
12. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: Proceedings of the 34th International Conference on Software Engineering. pp. 573–583. IEEE Press, Piscataway, NJ, USA (2012)
13. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: International Workshop on Model, Design and Validation. pp. 29–40 (Nov 2004)
14. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and Systems Modeling 4, 386–398 (2005)
15. Grönniger, H., Ringert, J.O., Rumpe, B.: System model-based definition of modeling language semantics. In: Formal Techniques for Distributed Systems. LNCS, vol. 5522, pp. 152–166. Springer (2009)
16. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)
17. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: Model Driven Engineering Languages and Systems, pp. 653–667. Springer (2011)
18. Jackson, E.K., Sztipanovits, J.: Towards a formal foundation for domain specific modeling languages. In: Proceedings of the 6th ACM / IEEE Int. Conf. on Embedded Software. pp. 53–62. EMSOFT '06, ACM, New York, NY, USA (2006)

19. Jackson, E.K., Sztipanovits, J.: Constructive techniques for meta-and model-level reasoning. In: Model Driven Engineering Languages and Systems, pp. 405–419. Springer (2007)
20. Kang, E., Jackson, E., Schulte, W.: An approach for effective design space exploration. In: Calinescu, R., Jackson, E. (eds.) Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems, LNCS, vol. 6662, pp. 33–54. Springer Berlin Heidelberg (2011)
21. Kovács, L., Voronkov, A.: Interpolation and symbol elimination. In: Schmidt, R.A. (ed.) Automated Deduction CADE-22. LNCS, vol. 5663, pp. 199–213. Springer Berlin Heidelberg (2009)
22. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into use. In: TOOLS'11 - Objects, Models, Components and Patterns. LNCS, vol. 6705, pp. 290–306 (2011)
23. Le Berre, D., Parrain, A.: The sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
24. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: 37th IEEE/ACM Int. Conf. on Software Engineering, ICSE. pp. 609–619 (2015)
25. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. pp. 130–145. ECMDA-FA '09, Springer-Verlag, Berlin, Heidelberg (2009)
26. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008). LNCS, vol. 4963, pp. 337–340. Springer (2008)
27. The Object Management Group: Object Constraint Language, v2.0 (May 2006)
28. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. 73, 1–22 (2012)
29. Salay, R., Chechik, M.: A generalized formal framework for partial modeling. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, LNCS, vol. 9033, pp. 133–148. Springer Berlin Heidelberg (2015)
30. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. Journal of Object Technology 14(3), 3:1–31 (2015)
31. Salay, R., Famelis, M., Chechik, M.: Language independent refinement using partial modeling. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering, LNCS, vol. 7212, pp. 224–239. Springer Berlin Heidelberg (2012)
32. Semeráth, O., Barta, A., Horváth, A., Szatmári, Z., Varró, D.: Formal validation of domain-specific languages with derived features and well-formedness constraints. Software and Systems Modeling pp. 1–36 (2015)
33. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model Pruning. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). Denver, Colorado, USA (Oct 2009)
34. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation. pp. 1–10. ACM (2009)
35. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: Design, Automation and Test in Europe, (DATE'10). pp. 1341–1344. IEEE (2010)
36. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (October 2007)
37. Yakindu Statechart Tools: Yakindu, `http://statecharts.org/`