Incremental Backward Change Propagation of View Models by Logic Solvers*

Oszkár Semeráth[1,2], Csaba Debreceni[1,2], Ákos Horváth[1] and Dániel Varró[1,2]
[1]Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems
{semerath,debreceni,ahorvath,varro}@mit.bme.hu

ABSTRACT

View models are key concepts of domain-specific modeling to provide task-specific focus (e.g., power or communication architecture of a system) to the designers by highlighting only the relevant aspects of the system. View models can be specified by unidirectional forward transformations (frequently captured by graph queries), and automatically maintained upon changes of the underlying source model using incremental transformation techniques. However, tracing back complex changes from one or more abstract view to the underlying source model is a challenging task, which, in general, requires the simultaneous analysis of transformation specifications and well-formedness constraints to create valid changes in the source model. In this paper we introduce a novel delta-based backward transformation technique using SAT solvers to synthetize valid and consistent change candidates in the source model, where only forward transformation rules are specified for the view models.

1. INTRODUCTION

View models are a key concept in domain-specific modeling tools to provide task-specific focus (e.g., power or communication architecture of a system) to engineers by creating a model which highlights only some relevant aspects of the system to help detect conceptual flaws. Typically multiple *view models* are defined for a given an underlying *source model*, which need to be refreshed automatically (or upon user request) upon changes in the source model.

The derivation and maintenance of views has been extensively studied for a long time in database theory over relational knowledge bases, while it has recently become a popular research topic also in model-driven engineering [6, 10, 17]. In [6], we proposed a declarative approach to define view models by graph queries [29] where the view model can correspond to a different metamodel independently from the source language. These queries are handled as unidirectional (source-to-view) incremental model transformations to automatically derive and efficiently maintain view models upon changes of the source model. This approach allows complex chaining of dependent views where a view model may act as a source model, which is beneficial in developing complex tool chains [15].

However, such view models are read-only representations derived by a unidirectional transformation, and they cannot be changed directly. When a view model needs to be changed, the engineer is forced to edit and manually check the source model until the modified model corresponds to the expected view model. Additionally, the effects of a source change need to be observed in all other view models to avoid unintentional changes and to prevent the violation of structural well-formedness (WF) constraints. The fact that changes in the view model cannot be directly propagated back to a change in the source model hinders the use of view models in an industrial case setting.

To tackle this problem, we propose a technique to automatically calculate possible source model candidates for a set of changes in different view models. First, the possibly affected partition of the source model is identified by observing traceability links to restrict the impact of a view modification. Then the modified view models, the query-based view specification and the well-formedness constraints of the source model are transformed into logic formulae. By using an iterative technique [27] over the Alloy Analyser [16], our approach enumerates multiple (but not all) valid resolutions of the source model corresponding to the changes of view models and *the constraints of the source model*. As a result, source elements unaffected by the target change may still need to be added as a side effect to make the source model consistent. We illustrate our technique on a healthcare example. The current paper extends the conceptual overview of [26] by presenting the technical contents in depth, and providing a first performance evaluation.

Our method provides advanced support for a class of bidirectional model transformations where each element in the view is defined unidirectionally by a declarative query [6, 10]. Arbitrary changes of view models are supported and incrementally back-propagated without backward transformation rules. Our approach allows the engineer to select from multiple source candidates or to restrict the scope of considered source changes. Moreover, changes from multiple view models are merged into a consistent source model where the

*This paper is partially supported by the CONCERTO (ART-2012-333053) and the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems.

MODELS '16, October 02 - 07, 2016, Saint-Malo, France

ISBN 978-1-4503-4321-3/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976767.2976788

consistency criteria also includes the well-formedness constraints of the source language.

Next, Sec. 2 overviews the concepts of view models and our past work on deriving view models by query-based unidirectional transformations. Then our backward change propagation approach (from view models to source models) is presented in details in Sec. 3. An initial experimental evaluation is provided in Sec. 4. Related work is overviewed in Sec. 5 while Sec. 6 concludes the paper.

2. VIEW MODELS

In a domain-specific modeling tool, the underlying domain model is presented to the engineers in different views. These views are frequently represented as models themselves (called view models and denoted by M_V in the sequel), which are populated from the underlying domain model (called source model, M_S). One source model may populate multiple view models. In a general setting, view models can be detached from the source model to such an extent that they correspond to a different language, thus they need to be compliant with a *view metamodel* MM_V and satisfy *view-specific well-formedness constraints* WF_V.

A view model is derived from the source model by a unidirectional *forward transformation* $M_V = \mathsf{fwd}(M_S)$. This is a restricted class of model transformations where query-based declarative techniques are especially suitable [10, 6]. Efficient live maintenance of a view model upon changes of the source model can be carried out by incremental transformation techniques [6, 11] even for multiple view models ($M_{Vi} = \mathsf{fwd}^i(M_S)$) or chains of view models ($M_V = \mathsf{fwd}^2(\mathsf{fwd}^1(M_S))$).

A forward transformation frequently creates and maintains a trace model $T = T_{obj} \cup T_{fea}$ between the source M_S and view M_V models. An *object trace* T_{obj} is a relation which connects activations of rules (queries) in the source model M_S to objects of the view model M_V. Similarly, a *feature trace* T_{fea} (i.e. reference or attribute trace) is a relation which connects rule activations in the source model M_S to references in the view model M_V.

While view models may immediately reflect live changes in the source model, view models were immutable by the engineers in our previous work [6], which restricts the use of view models in an industrial setting. In the current paper, we allow view models $M_{V1}, \dots M_{VN}$ to be changed directly to $M'_{V1}, \dots M'_{Vn}$ and present an approach for backward change propagation for view models to an updated source model M'_S using logic solvers.

2.1 Motivating scenario

Our change propagation technique will be illustrated on a case study of a remote health care system developed in the Concerto project [1], which develops an environment for pulse and blood pressure measurement controlled by a smart phone, which is illustrated in the upper left part (1.) of Fig. 1. Measurements of pulse and blood pressure is measured by the sensors of a mobile phone, which are executed periodically triggered daily by the phone timer. The completion event of measurements triggers the processing of sensor data: pressureDone and pulseDone. The result of the measurement is collected in reports pulseReport and pressureReport, and sent to the different hosts. In our case study, the blood pressure is sent to the general practitioner (gp) of the patient for logging, and signs of hearth failure is sent to hospitals (modeled by emergency).

Two view models are derived from this source model in our telecare example which are maintained as the source model changes. The *Dataflow* view ($2.A$) shows which Hosts will be notified about each InformationTypes, while *Event* layout ($2.B$) describes event sequences represented by Activation nodes with *after* references between them leading from an Init node to a Finish node.

Let us now assume that changes are made in views illustrated in $3.A$ and $3.B$: $3.A$ represents a change where dataflow from Pulse to Emergency is redirected to the General Practitioner (denoted by «del» and «new»). In $3.B$, the action dedicated to report the pulse is removed from the view, but the remaining report waits for the completion of both measurement (denoted similarly). Our technique will allow to automatically generate valid and well-formed source model candidates like (4.) that conforms to the current state of the view model.

2.2 Definition of view models

In [6] we proposed to use declarative queries as derivation rules to (i) specify new view model elements in the target M_V, and (ii) maintain a trace model between the source M_S and view M_V models based on the matching queries (patterns). A graph pattern describes structural conditions on a model with a combination of path and type expressions equivalent to first order logic predicate. A derivation rule consists of a pattern predicate p and an action part where for each activation m of predicate p, the action part is fired. An activation m is a function that maps all parameters *Params* of predicate p to a object o in the source model M_S, $m : Params \to O_S$.

To help navigation along traces, two (injective partial) lookup functions are introduced: $lookup_{VS}(v)$ maps a view object v to a predicate p with its activation m over the source model and $lookup_{SV}(p(m))$ maps an activation to a view object when $(p(m), v) \in T$. The following actions are used in derivation rules[6]:

- AddObj(class:*Class*) Activation m of precondition p creates an entry $(p(m), v)$ in the trace with a unique view object v in the view model with the corresponding type class, where $v = lookup_{SV}(p(m))$. For each activation m of precondition p, there exists a unique $v \in O_V$ view object, where

  $$M_S \models p(m) \Leftrightarrow T \models (p(m), v) \Leftrightarrow M_V \models \mathsf{class}(v)$$

- AddRef(ref: *Ref*, sp: *src.pre*, sm: *src.match*, tp: *trg.pre*, tm: *trg.match*)

  Activation m of precondition p creates an entry $(p(m), (v_s, v_t))$ in the trace with a reference ref in the view model from the source v_s to the target v_t, where $v_s = lookup_T(\mathsf{sp}, \mathsf{sm}))$ and $v_t = lookup_T(\mathsf{tp}, \mathsf{tm}))$. For each activation m of precondition p exists $v_s, v_t \in O_V$:

  $$M_S \models p(m) \Leftrightarrow T \models (p(m), (v_s, v_t)) \Leftrightarrow M_V \models \mathsf{ref}(v_s, v_t)$$

- AddAtt(attr: *Att*, sp: *src.pre*, sm: *src.match*, val: *value*) Activation m of precondition p creates an entry $(p(m), (v_s, val)$ in the trace by setting the attribute attr of v_s view object to *val*, where $v_s = lookup_T(sp, sm))$. For each activation m of precondition p exists $v_s \in O_V$:

  $$M_S \models p(m) \Leftrightarrow T \models (p(m), (v, \mathsf{val})) \Leftrightarrow M_V \models \mathsf{attr}(v, \mathsf{val})$$

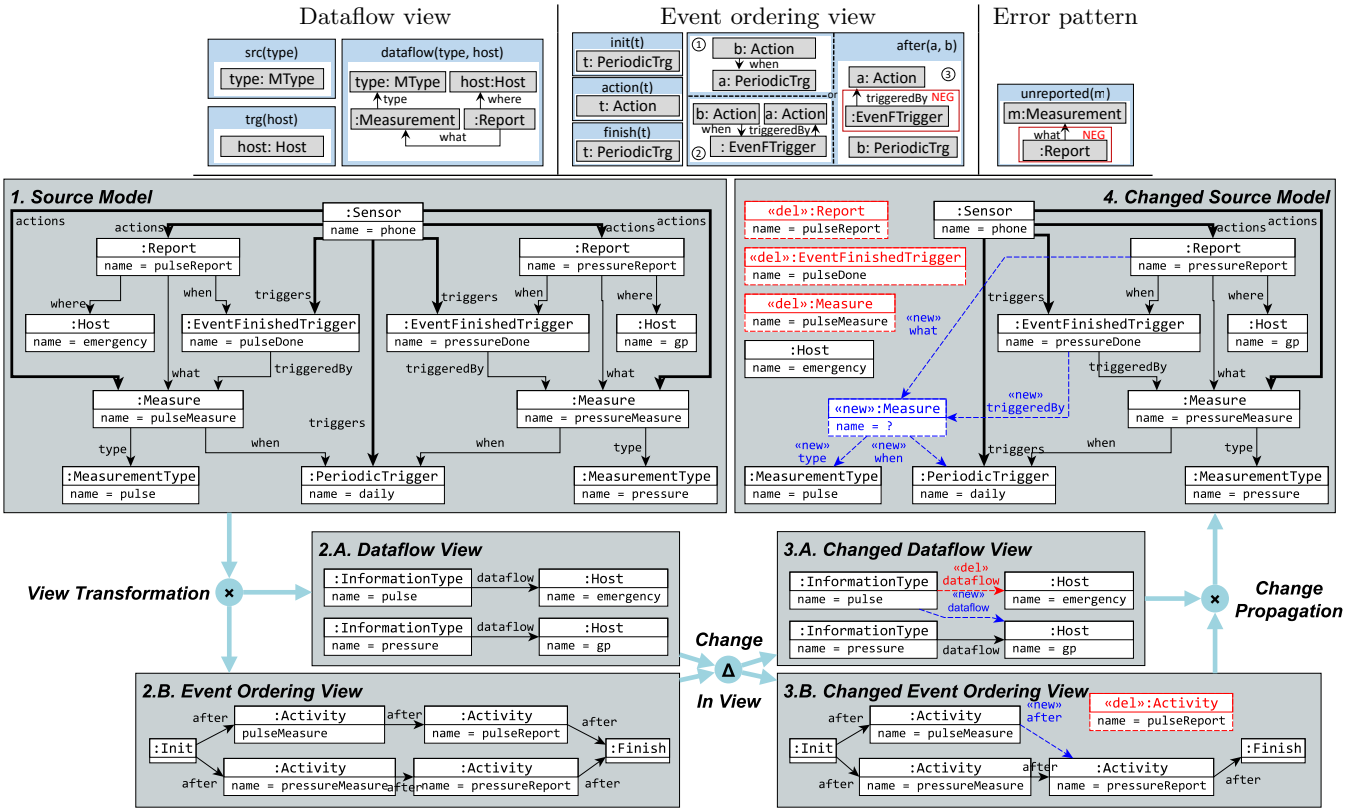

Figure 1: Motivating scenario on a healthcare example

As a result, we obtain a declarative formalism for defining view models with execution semantics compliant with incremental and live graph transformations [20]: when a new activation of a forward rule is detected, the corresponding view elements are created and when a previously existing activation of a forward rule disappears the related view elements are removed. However, this is still a restricted subclass of model transformations since (1) each rule creates exactly one new element (object or reference) in the view model, and (2) the transformation is monotonic in the sense that a view element always depends on the existence of a match of a positive pattern (i.e. we disregard from cases when a view element is created when a pattern cannot be matched).

Example Queries used for defining the views of our motivating example are depicted in the top of Fig. 1. For the *Dataflow* view, src query selects all the MeasurementType to create InformationType instances in the view, while trg is responsible for creating Host instances from the Host objects in the source model. The dataflow pattern has two parameters and selects all the type and host pairs that are connected to each other via a Measurement and a Report objects. The action part of the rule will create an edge between an InformationType and a Host associated with the two parameters upon a match appears for the pattern. Similarly, the *after* pattern is responsible for setting the after edge between view model objects. In case of (1.), the edge will go from an Init object to an Activity, (2.) describes the connection between two Activity, while (3.) activates when an Action (common ancestor of Report and Measurement) is not triggered by any EventFinishedTrigger. The init and finish queries create Init and Finish objects in the view for each PeriodicTrigger objects in the source model. Finally, the action query builds Activity instances from all Action objects.

2.3 Characterization of query-based transformation of view models

S. Hidaka et al. [14] classifies bidirectional transformation approaches based on their features. According to it, our previous work [6] is a *syntactic approach* for *forward functional* transformation of *MDE artifacts*. View models contain *no complement information*, hence they are regular models without additional annotations. These models are *total targets* as the full view model is specified by the consistency relations. Definition of a view model is *unidirectional*, however the expressiveness of definition is *Turing incomplete*. Forward propagation of the *operation-based* changes are *live*, *incremental* and executed *automatically* that also maintains *explicit traces*. However, that approach has *incomplete change support*, thus only the modification of the source model is supported.

3. BACKWARD CHANGE PROPAGATION BY LOGIC SOLVERS

3.1 Overview of approach

We present a novel approach to back-propagate view model changes into a consistent source model by using logic solvers. An overview of our approach is depicted in Fig. 2 where *target (view) models* $M_{V1}, M_{V2}, \ldots$ are derived from a

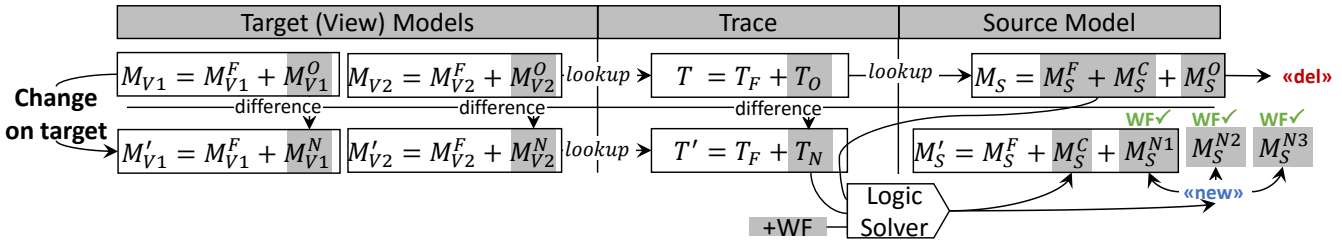

Figure 2: Overview of backward change propagation

source model (M_S) based on the matches of the view definition queries (in the source model), and a *traceability model* T is built and maintained during the forward transformation. Now the engineer makes changes to the view models, which leads to changed view models $M'_{V1}, M'_{V2}, \ldots$. The goal of our approach is to calculate (one or more) source model $M_S{}'$ which corresponds to the change, maintain T', and *satisfy additional constraints of the source model*.

A change in the view model can be separated into two partitions: the fixed model partition M^F_{Vi} denotes a partial model which remains unchanged, while M^O_{Vi} is updated to a new M^N_{Vi} (cross-references are included in M^O_{Vi} and M^N_{Vi}). The change is propagated consequently to the trace model: T_O contains the invalidated trace links, T_N symbolizes the new links to be created and T_F contains the remaining trace links which are not affected by the change. Along the traces, the change can be propagated back to the source model by identifying unchanged, newly activated and deactivated matches of queries in the source model. By analyzing the impact of changing matches in T_N, the source model can be partitioned into three partial models: a fix part ${M_S}^F$ contains the elements which cannot be changed, a changing part ${M_S}^C$ contains the object which can be modified, and a obsolete part ${M_S}^O$ which objects can be deleted. The changing trace T_N declaratively specifies structural constraints on the ${M_S}^C \cup {M_S}^O$ model which have to be satisfied in order to ensure the consistency of the forward transformation.

A possible solution M'_S for the changed parts ${M_S}^C \cup {M_S}^{Ni}$ has to (1) match the fixed part M^F_{Vi}, (2) the requirements defined by the changed matches defined in T_N, and (3) additional domain-specific *WF* (of the source model). All these constraints are transformed into a first-order logic problem to be solved by a *logic (SAT/SMT) solver* following [27]. The solver provides several (but not necessarily all) possible valid solutions for ${M_S}^C$ and it may create new object in ${M_S}^{Ni}$ from which from the model developer may choose the most appropriate one ${M_S}^{Ni}$.

As a summary, our approach integrates two novel techniques into an interactive workflow: an inverse impact analysis by *change partitioning* (Sec. 3.2) separates the affected and unaffected parts of the source model, while *partial model generation* creates candidate source models that correspond to the new target views (Sec. 3.3) and source constraints.

3.2 Change partitioning

The rationale of change partitioning is as follows: (i) if a view model element does not change, the associated traces cannot be changed; (ii) otherwise, if the change of a source object may induce a valid view model it has to be selected. However (iii) unnecessary source changes should not happen. To ensure these conditions, an impact analysis of the changes has to be conducted to identify the affected part of source model which can be modified.

3.2.1 Affected parts of view model changes

Table 1 contains the calculation of affected parts in case of modifying a view model M_V. When a view object v is removed, the affected parts are determined by its defining pattern p and its activation m that is stored in the trace model T. When a reference ref is removed from M_V, the affected part of the source v_s and the target v_t are returned. When a new view element is created, then there are obviously no activations of forward rules.

3.2.2 Affected parts of pattern activations

A pattern predicate p with its activation m marks the union of the bodies defined in Table 2. Each body consists of several conditions *const* that may introduce additional internal variables $Params_i$. Hence, an activation of a *body* is extended with m_i all the possible bindings of internal variables. The affected part of a body *body* is the union of the affected parts of each constraints *const*.

3.2.3 Affected parts of source constraints

Affected parts of source constraints are defined in Table 3. A *class* (class) condition returns the object that is bound to its parameter along activation m. Similarly, *attribute*(attr) and *reference*(ref) conditions (together feature conditions *feature*(feat) select respective parameters (x and both x, y) from m. A *path*($\mathsf{feat}_1 \ldots \mathsf{feat}_n$) condition can be split into several *feature*(feat) to calculate its affected part. For *equal*($=$) and *not equal*($\neq$), the bound objects of parameters from both side of the operators are returned.

A pattern p may call another pattern p' (*find*$[p']$ or transitively with *find*+$[p']$) by mapping symbolic parameters $Params_{p'}$ of the called pattern to concrete values of the caller. Thus given a binding : $Params_{p'} \rightarrow Params_p$, a match m' is composed from m by getting objects from the original activation m, formally:

$$m' \circ m \Rightarrow m' : m(\mathsf{binding}(var')) \rightarrow O_S, var' \in Params'_{p'}$$

However, a *negative application condition* (*neg find*$[p']$) is separated into two cases: if the sub predicate p' does not introduce any internal variable, the affected part returns the referenced objects from the activation m. Otherwise, we restrict the affected part to all the objects that has the same type as the introduced internal variables have.

3.2.4 Categorization of affected source model objects

The affected objects of the source model can be categorized into three groups:

$-\mathsf{class}(v) \rightarrow \mathsf{affected}(p(m)) : lookup_{VS}(v) = p(m)$
$-\mathsf{ref}(v_1, v_2) \rightarrow \mathsf{affected}(p(m)) : lookup_{T}(v_1, v_2) = p(m)$
$-\mathsf{attr}(v, val) \rightarrow \mathsf{affected}(p(m)) : lookup_{T}(v_1, v_2) = p(m)$

Table 1: Affected changes of view model

$p(m)$	$\rightarrow \bigcup \mathsf{affected}(body_i, m)$
$body(m)$	$\rightarrow \bigcup \mathsf{affected}(cond_i(m + m_i))$

Table 2: Affected activations

$\mathsf{class}[obj], (m) \rightarrow \{o_{obj} \mid m(obj) = o_{obj}\}$
$\mathsf{attr}[x, val], (m) \rightarrow \{o_x \mid m(x) = o_x\}$
$\mathsf{ref}[x, y], (m) \rightarrow \{o_x, o_y \mid m(x) = o_x, m(y) = o_y\}$
$\mathsf{feat}(m) \rightarrow \begin{cases} \mathsf{affected}(\mathsf{attr}(m)) \text{ if feat is attr} \\ \mathsf{affected}(\mathsf{ref}(m)) \text{ if feat is ref} \end{cases}$
$\mathsf{feat}_1 \ldots \mathsf{feat}_n[x, y](m) \rightarrow \bigcup \mathsf{affected}(\mathsf{feat}_i(m))$
$\begin{matrix} x = y, (m) \\ x \neq y, (m) \end{matrix} \rightarrow \{o_x, o_y \mid m(x) = o_x, m(y) = o_y\}$
$find[p'](m) \rightarrow \mathsf{affected}(p'(m')), m' \circ m$
$find^{+}[p'](m) \rightarrow \bigcup \mathsf{affected}(p'(m')), m' \circ m$
$neg\ find[p'](m) \rightarrow \begin{cases} \{o_x \mid m(x) = o_x, \}, \text{ if no inner var} \\ \{o_i \mid o_i.type \in \text{inner types of } p'\} \end{cases}$

Table 3: Affected changes of source constraints

- ${M_S}^F$: *neither changeable nor removable*: It includes all objects of M_S which are not in the affected part of the change from the view Δ_{view}.

$${M_S}^F = M_S - \mathsf{affected}(\Delta_{view})$$

- ${M_S}^C$: *changeable but non-removable objects*: It includes all objects in the affected part of the change from the view Δ_{view} which are responsible for the existence of other activations.

$${M_S}^C = \mathsf{affected}(\Delta_{view}) - \{o \mid o \text{ referred by } T_{obj}\}$$

- ${M_S}^O$: *changeable and removable objects*: All objects in the affected part of the change from the view Δ_{view} which are not responsible for the existence of any other activations.

$${M_S}^O = \mathsf{affected}(\Delta_{view}) - {M_S}^C$$

Example. In our example of Fig. 1, the affected part for the deletion of dataflow edge from the view model is calculated as follows. The trace model T_F stores that the existence of dataflow edge is related to the dataflow pattern where the activation binds pattern parameters emergency:Host and pulse:MeasurementType to objects in the source model M_S. The affected part of the pattern includes all objects related to the match {*emergency,pulse*}, and the affected part of the constraints includes internal variables {*pulseReport, pulseMeasure*}. However, *pulseMeasure* is also responsible for an after edge in the other view model, thus it can be changed but not allowed to be deleted from the source model. At this stage, the user may manually move objects between these categories (${M_S}^F$,${M_S}^C$,${M_S}^O$) to refine his/her intention on source candidates.

$${M_S}^C = \{pulseMeasure\}$$
$${M_S}^O = \{emergency, pulse, pulseReport\}$$
$${M_S}^F = \{\text{rest of the objects}\}$$

3.3 Model Generation by Logic Solvers

Logic solver based model generation for a domain specific language is an actively researched area. Instance models can be created to provide models that satisfies required properties, test cases or to create counterexamples for false language properties [25], and incremental model generation techniques [27, 23] are able to take advantage nearly finished partial instance models.

3.3.1 Logic Representation of View Models

In general, solver-based model generation takes the logic representation of the metamodel *Meta* to synthesize conforming instance models M with $M \models Meta$. Each class C_i is represented by a predicate over the objects of the model $C_i \subseteq O_M$, each reference R_i is mapped to a relation over pairs of objects $R_i \subseteq O_M \times O_M$, and attributes are modeled by $A_i \subseteq O_M \times Type_i$, where $Type_i$ is the domain of attribute i. Additionally, *Meta* contains basic structural constraints of the metamodel as axioms (e.g. multiplicity and containment hierarchy, see complete [25] for full details), so the interpretation of relations represents structurally correct models. The logic problem can extended to include well-formedness constraints *WF* [25] defined either as graph patterns or OCL invariants to generate valid solutions.

A model query P_i with can be represented as a predicate over objects of the target model $P_i : O_M \times \ldots \times O_M \rightarrow \{true, false\}$ which is evaluated to true only if some objects satisfy the translated query specification. A match m_i is represented as a tuple of objects: $m_i = (c_i^1, \ldots, c_i^n)$, where $c_i^j \in O_M$. Pattern matches are controlled by two formulae: (1) a pattern P_i has a match $m_i = (c_i^1, \ldots, c_i^n)$ if and only if the constants satisfies the associated logic predicate $P_i(c_i^1, \ldots, c_i^n)$; (2) each match of a pattern has to be unique: for all matches m_i and m_j there is at difference $\exists_{x \in 1..n}(c_i^x \neq c_i^x)$. Consistency with the view model is ensured by a formula set *View*, which controls the matches of query predicates. Therefore, the generation of a valid and consistent view model is specified as $M \models Meta \wedge WF \wedge View$.

Example. In the logic equivalent of our running example the type, what and where references are modeled by relations R_{type}, R_{what}, R_{where}. The specification of dataflow(t,h) pattern can be represented by the following predicate:

$$P_{DF}(t, h) \Leftrightarrow \exists i_1, i_2[R_{type}(i_1, t) \wedge R_{what}(i_2, i_1) \wedge R_{where}(i_2, h)].$$

Here t and h has to be connected by a specific path of relations. In our example the changed dataflow model has two matches: $m_1 = (c_1^1, c_1^2)$ and $m_2 = (c_2^1, c_2^2)$, where $c_1^2 = c_2^2$ (as both types are forwarded to the general practitioner). With the following axioms added to the logic problem it can be ensured that there are exactly two matches of the pattern (as defined by the dataflow view), and each match is unique:

$$\forall h, t[P_{DF}(t, h) \Leftrightarrow (t = c_1^1 \wedge h = c_1^2) \vee (t = c_2^1 \wedge h = c_2^2)]$$

$$(c_1^1 = c_2^1 \vee c_1^2 = c_2^2)$$

3.3.2 Incremental Transformation of View Models

In case of view models, the affected part ${M_S}^C \cup {M_S}^O$ typically remains proportional to the change, thus ${M_S}^F$ explicitly defines most of the generated models. Incremental model generation techniques like [27] are able to take advantage of fully specified model fragments, and encode the graph generation problem in a way that the problem is proportional to the newly created fragment. In the following, we give a brief description of the mapping technique.

- **Objects**: the object set is partitioned into three subsets: ${M_S}^F$ is mapped to the fix objects O_F, ${M_S}^C$ stands for the changing objects and finally O_N replaces the objects which are removed ${M_S}^O$. In general, predicates dealing with ${M_S}^F$ are interpreted, thus the solver already knows its truth evaluation.

- **Classes**: Each class predicate C is also separated into three subsets: a fully interpreted C_F defined over O_F, a fully interpreted C_C defined on the changing objects O_C, and the uninterpreted C_N over O_N. In summary, the solver has to interpret only relations of C_N.

- **References**: Reference predicates are also separated to multiple smaller relations: R_{OO} represents the interpreted relation between fixed objects, R_CC the reference between changing objects, and R_{NN} represents the reference between new objects. Additionally, uninterpreted cross-references have to be added for references connecting these regions: R_{OC}, R_{ON}, R_{CO}, R_{CN}, R_{NO}, R_{NC}. While only R_{OO} is interpreted from the nine new relations, it contains the most references.

- **Attributes**: Attribute predicates are also separated into three partitions for O_F, O_C and O_N.

- **Model Queries and Matches**: Model queries are separated into multiple queries, each parameter can be bound to O_F, O_C and O_N. This might add several relations to the logic problem, but the unchanged matches are already interpreted. In the construction of the constraint set $View$, the uniqueness of non-interpreted matches needs to be ensured

Compared to solving the model generation problem as a whole, our incremental approach enables the logic solver to handle much fewer variables as a large fragment of the model is already interpreted (prior to calling the solver). The downside is that constraints become more complex as they have to be separated into those groups above. However, we expect that most predicates remain interpreted, which is beneficial for the solver.

3.4 Properties of our approach

Our approach has the following properties (based on [14]):

1. *Full operation support on views:* View models can be edited as regular models, while the technique ensures consistency between source and view models.

2. *Implicit backward consistency:* View models derived from a source model candidate $M'_S = \mathsf{fwd}^i(S')$ are isomorphic to the view models $V'_1, \dots V'_n$.

3. *Delta-based and offline back-propagation:* After making changes on the target models, our approach generates source model candidates from a stable state of the views and the changes in the traces. Upon a sequence of (possibly concurrent) view changes is applied it leads the view models to a new stable state. Then the difference between the previous and current state of the views can be propagated back even if some changes are contradictory or inconsistent by themselves.

4. *Interactive execution:* There might be several source candidates for a view model change on which the solver can iterate, starting from the smallest solution. The developer or a selection strategy can select the most suitable one from the sequence of valid solutions.

5. *Well-formedness:* S' satisfies the well-formedness constraints of the source domain $S' \models WF$.

6. *Incrementality:* A view change can affect only the affected part of the source model. Additionally, if a view model element is not changed, it should not be changed by the effect of back propagation (even if the result would create isomorphic). Thus all matches m which $T \models (p(m), v)$ and $v \in M'_V$ should remain in $T' \models (p(m), v)$.

7. *Hippocraticness of unaffected partition:* If a view model element is not changed, the associated source model part has to remain unchanged.

Conditions 1-4 are related to usability and they connect the new backward propagation technique to our previous forward transformation approach. Some form of consistency is ensured in several approaches (e.g. [5]) but we also incorporate WF constraints of the source language. Hippocratic behavior defined in [28] states that a backward or forward transformation must not modify the source or the target model if they are already consistent. In Property 6. and 7. we define a stronger requirement which states that consistent partitions of the source and target models should not be modified. This constraint simultaneously keeps most of the source model untouched and makes the deduction phase more efficient by limiting the task to partial models.

4. EXPERIMENTAL EVALUATION

In order to evaluate the performance of the key step of our backward change propagation technique we have conducted initial measurements on a prototype implementation in the context of the running example as case study (taken from the CONCERTO project). The measurement scenarios and the results are available on GitHub[1]. Our measurements aim to address the following questions:

Q1 *What is the influence of the size of the source model, the size of the target change and the scope of the source model (i.e. the number of newly created source objects) on the runtime of the solver?*

Q2 *What is the difference between incremental model generation and full model generation (like in [10]) with respect to performance and the quality of results?*

[1] https://github.com/FTSRG/publication-pages/wiki/incremental_backward_change_propagation_of_view_models_by_logic_solvers

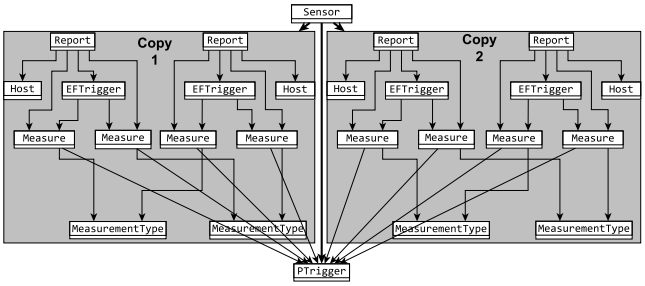

Figure 3: Structure of the generated source models

Q3 *What are the (solver-specific) limitations of our backward change propagation technique?*

Our evaluation exclusively focuses on assessing the performance of the model generation step for the source model (detailed in Sec. 3.3), and excludes the performance evaluation of change partitioning (Sec. 3.2). In our initial experiments, we experienced that both the change partitioning time (i.e. the selection of affected source elements) and the forward transformation time is negligible (less than 1 second for the largest problems we measured) compared to the time required to solve the logic problem by Alloy. Thus performance limitations are dominated by the latter.

4.1 Change propagation problem generator

In order to measure the performance of our technique we extended the running example visible in Fig. 1 to a change propagation benchmark, which can be parametrized and scaled by the *size (s)* of the source model and *the number of changes (c)* in the views. We have created valid health care models illustrated in Fig. 3 in the following way :

1. First, a Sensor is created with a PeriodicTrigger.

2. Two MeasurementTypes are added to the model, which are measured by two respective Measures activated by the periodic trigger. Then two Reports are added to the model, which are triggered by two new *EventFinishedTrigger*s waiting for the measurements of two different types. Then the result is reported to two newly created Hosts.

3. Step 2 is repeated s times, which means that the model has $2s$ MeasurementTypes, $4s$ Measures, $2s$ Reports, $2s$ EventFinishedTriggers and $2s$ Hosts.

4. The view models are derived, which creates in a dataflow model with $2s$ Hosts and $2s$ InformationTypes $4s$ dataflow references, and an event ordering model with 1 Init, 1 Finish, $6s$ Activities and $10s$ after references.

5. Then changes are applied to the view models: c MeasurementType and c Hosts with their dataflow references are randomly removed from the first view model, and a new Action is added to the event ordering view.

As a result, we obtain non-trivial source and view models, while the random changes of the view model remain semantically meaningful.

4.2 Measurement Setup

Each model generation task was executed on the generated healthcare change propagation problems using the Alloy Analyzer (with SAT4j-solver). Each change propagation problem is solved with our incremental solution which generates only the affected part in the source model. As a baseline, we compare it to a solution conceptually similar to [10] which generates full models for a view model. The full model generation is achieved as a corner case of the incremental generation where the changed view models are interpreted as if they were newly created, and no part of the source model is preserved.

We measured the runtime of Alloy Analyzer, which consists of an initial conversion to a conjunctive normal form and then solving the SAT-problem by the back-end solver. We executed each measurement 5 times, then the average of the execution times was calculated. The measurements were executed with a 120 second timeout on an average personal computer[2]. The memory usage of the solver was always below 2 GB.

4.3 Measurement Result

The execution times of our measurements (see Fig. 4–Fig. 7) are given in seconds.

- $|S|$ denotes the number of objects in the original source model (M_S);
- $|\Delta V|$ denotes the size of the target change, i.e. the sum of removed and newly created matches in the view model (M_V^O and M_V^N);
- Finally, $|N|$ denotes the source scope, i.e. the number of new objects in the changed source model candidates (the number of objects in ${M_S}^N$). It is equivalent to the scope of model generation in Alloy when incremental technique is used.

In case of full model generation where the each model object is newly created, the size of the original model is subtracted from this value, so the two techniques are comparable with respect to the number of objects.

4.4 Increasing model size

First, Fig. 4 displays the runtime results of cases where only one change is performed ($c = 1$, $\Delta V = 8..9$). Eleven different series are measured, where the change propagation is solved with source scope of $|N|$ new elements (ranging from 0 to 10) with increasing the size of source models $|S|$ up to 123 objects. With 0 new elements, the problem was unsatisfiable, otherwise valid solutions were created.

The results show a polynomial (cubic) increase of runtimes in the size of the original source model, and it always harder to solve the problem with increasing source scope size. Additionally, the solver has a similar characteristics regardless a problem is satisfiable or unsatisfiable. Fortunately, smaller solutions can be retrieved more efficiently, which is typically preferred over solutions with unnecessary elements.

We also depicted the run-times in Fig. 5 while further increasing the size of source scope. This shows that the source scope needs to be decreased when the size of source models increases in order to obtain the same run-time.

[2]CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10, Reasoner: Alloy Analyzer 4.2 with sat4j

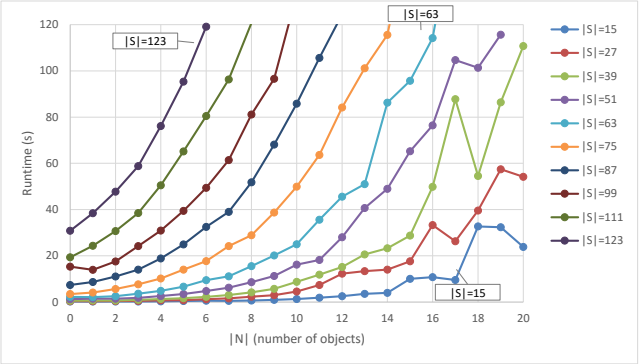

Figure 4: Runtime with increasing source model size (vertical axis: size of source model, series: size of source scope, change size: 1)

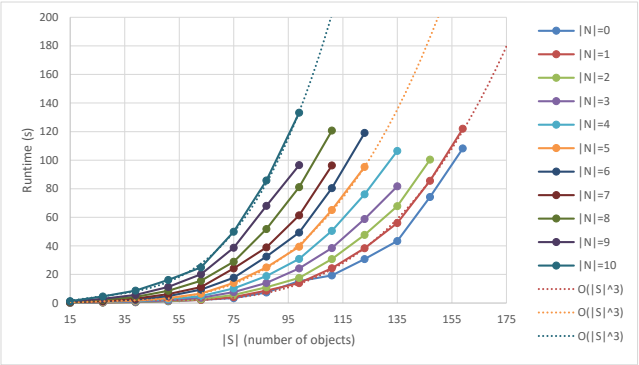

Figure 5: Runtime with increasing source scope size (vertical axis: size of source scope, series: size of source model, change size: 1)

4.5 Increasing target change size

Fig. 6 displays the results with larger target change sizes. Here, the vertical axis displays the source scope, i.e the number of new objects added to the source model by the solver, and different series shows the run-times of target change size ranging from 8 up to 43 changes, and the measurement is repeated for two model sizes. For the smallest change one new element is needed to successfully create solutions, two new object is needed for the next change size, and so on, so the largest changes needed at least 5 new objects.

The results shows that the run-time is not directly affected by the size of the change, different change sizes have the same complexity. However, a larger target change size usually implies a larger source scope, which, of course, influences the run-time of the solver. In other terms, if the consequences of a large target changes are attempted to be propagated back to the source model with a small scope size, then the logic solver will conclude that the problem is unsatisfiable. While if we also increase the scope size, then the problem might become satisfiable - but the logic solver may fail to find a solution due to its performance limits.

Therefore, based on our measurements of increasing model and change size, our first question can be addressed as follows:

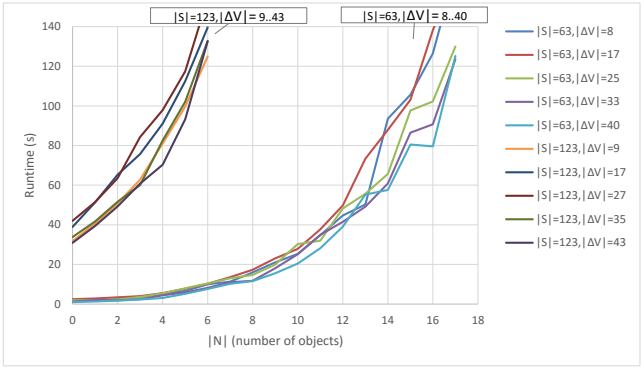

Figure 6: Runtime with different change size (vertical axis: size of source scope, series: size of model, size of change)

A1 *The runtime of the solver is a polynomial function of the original size of the model and the source scope size (i.e number of newly added elements). The size of the target change increases the size of the source scope required for a solution.*

4.6 Incremental vs. full generation

In Fig. 7 the runtime of the incremental and full model generation is depicted with respect to the size of the source model size and the source scope. Only results of two small source models (15 and 27) are presented as on all larger cases, the full model generation technique was unable to provide a solution.

In comparison, the incremental model generation technique performed much better: it was able to solve some nontrivial problems, and in general, it was orders of magnitude faster than full generation.

From the perspective of model quality, the full model generation approach redirected several relations where the target view model was unchanged (e.g. unchanged dataflows), which might be undesired in change propagation scenarios. On the other hand, the full change propagation may find solutions which requires changes in partitions categorized as unaffected by our change partitioning, which can be only retrieved by manual configuration of the affected part in case of incremental generation.

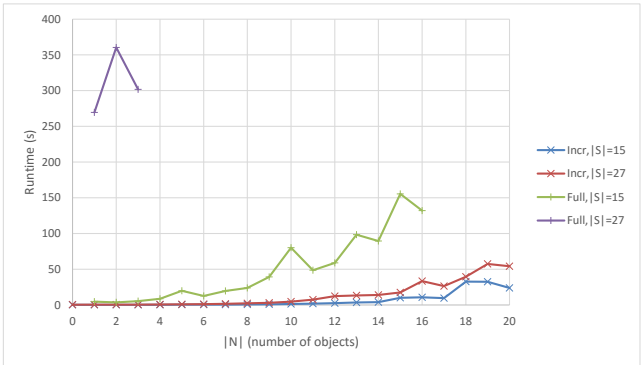

Figure 7: Incremental versus full model generation

The difference between incremental and full model gen-

eration for backward change propagation purposes can be summarized as follows:

A2 *Incremental model generation provides better performance for source models of increasing size for a given scope. Full model generation may be able to retrieve some hidden solutions. Unlike full model generation, incremental generation is able to detect if a change propagation setup is unsatisfiable for a given scope.*

4.7 Limitations

Our initial experimental results revealed several limitations of our approach most of which were caused by using Alloy as an underlying logic solver. When investigating the runtime for model generation, we found that over 80% of the time is spent for an initial conversion to Conjunctive Normal Form (CNF) prior to starting the model finding step. In fact, CNF conversion took over 99% of the time for large models with $|S| = 160$ elements. The largest source model our technique was successfully executed had 243 objects, and the run-time for propagating a single target change was over 20 minutes (again, spent dominantly on CNF conversion).

While our incremental query-based forward transformation scales to source models orders of magnitude larger, we can state as a conclusion that such scalability cannot be achieved for incremental backward change propagation when using Alloy as a solver.

However, the fact that model generation time is dominantly spent on CNF conversion in Alloy (and not on the actual SAT/SMT-based model finding), this may trigger future research to replace Alloy with a dedicated incremental model finder for EMF-based models.

Furthermore, when investigating small backward change propagation problems with our approach, one may gain domain-specific insights to identify specific target changes which can be always mapped to a source change (e.g. by some rule-based transformation techniques).

A3 *Incremental backward change propagation using model generation by Alloy scales only to small models compared to incremental forward transformations. But an incremental model generation technique scales significantly better than full model generation, which is a hope for dedicated future model generators.*

4.8 Threats to Validity

Finally, let us investigate the most critical threats to validity of our conclusions.

- While our case study is relatively complex (as it originates from the CONCERTO project), our measurements were executed only on a single case study, thus our findings may not be applicable in a more general context. However, our model generator approach has strong roots in [27] where efficiency of incremental model finding by using existing model solvers were assessed, and the experimental results point to similar limitations. Moreover, our negative results (e.g. poor performance of Alloy) are more likely generalizable for other model generation and transformation scenarios.

- We excluded the time of forward transformation and change partitioning from our measurements - as initial experiments showed that they were less than a second.

- As execution times of Alloy quickly started to increase, we only had 5 measurements for each case, thus we did not carry out a full-fledged statistical analysis of our results. Correspondingly, our findings are softer (more qualitative than quantitative).

5. RELATED WORK

Most existing view model synchronization techniques use bijective transformations where transformations can be executed in both the forward and reverse directions such in lenses [8], injective functions [18] or ATL [30]. Triple Graph Grammars (TTG) [24] are a well-know approach for model synchronization [12] where the forward and reverse transformation rules are derived automatically from a bidirectional rule definition. A special class of TGG is View TGG [2] which is specialized for efficient update propagation. As a fundamental difference, our approach uses patterns to define the well-formedness constraints and the view instead of generative graph grammars.

Most closely related approaches for view synchronization are listed in Table 4. To compare them to our approach, we use several characteristics to guide the structure of this section.

Using Logic Solvers. Using logic solvers for generating possible source and target candidates is common part of several approaches. [5] uses Answer Set Programming, [4] maps the problem to Mixed Integer Linear Programming. Those approaches use solvers to select model elements which may alter the matches related to view model changes (similar to the calculation of the affected part). As a difference, our approach takes the whole DSL specification into the account, to change source model elements which are only implicitly related to the view changes, caused by the interaction of the metamodel, the WF constraints and other view models.

[17] uses Alloy to generate change operations on the source model which leads to a modified source model which is (i) well-formed and (i) consistent with the changed target model. As a difference, our solution creates the changed partition (and not the change operations). [10] and [9] converts the transformation to Alloy similarly, but do not handle WF constraints of the source model, and changes the whole source model. By selecting the affected part, our solution likely has better scalability as it has to manage less objects: [9] scales up to 20 objects in the source and target models in total, and no other measurement is given. This technique is also suggested in the future work of [17].

Traceability Links. For the backward propagation of changes, use of traceability links is a well-accepted approach to define which part of the source model has to be updated upon a change on the target model. In [24], these links are stored as a *correspondence model* where their maintenance is derived from the TGG rules. [19] also specifies *trace* classes to facilitate and maintain traceability links. [10] stores traceability links in Alloy[16] as a bijective mapping. [3] uses a *weaving* model that stores the traces of references between different models in the view, however all objects in the view model act as proxies to an object in the source model. Our solution builds and maintains a traceability during the forward propagation of changes. Moreover, we reuse the information stored in the traces to improve the affected part calculation for changes in the view.

Well-formedness constraints. To avoid the calculation of *ill-formedness* source models after the backward propa-

Approach	Logic Solver	Traceability	WF const.	Partial Model	Interoperability
ATL[30]	-	-	-	-	+
TGG[24]	-	+	-	-	+
QVT-R[19]	-	+	+	-	+
QVT-R with Alloy[17]	SAT Solver	-	+	-	+
JTL[5]	ASP	-	-	-	+
MTE with MILP[4]	MILP	-	-	-	+
EMF Views [3]	-	+	-	+	+
F-Alloy [9]	SAT Solver	-	+	-	+
QueST[10]	SAT solver	+	+	-	+
Our solution	SAT solver	+	+	+	+

Table 4: Comparison of related approaches

gation of a view model change, well-formedness constraints should be taken into account. This property is supported in the specification of [19] and by [17] and in our approach.

Partial synchronization. [13] defines *partial synchronization* to apply the changes of target model only to the relevant part of the source model. This reduces the number of possible source candidates. Our approach identifies the fixed part of the source model, that cannot be changed, and selects the complement of this part which will be the basis of constraints. While [13] defines a formal framework for model synchronization, our solution can be interpreted as an efficient and view-model specific realization of it.

Partial models have certain similarity to *uncertain models*, which offer a rich specification language [7, 21] amenable to analysis. Uncertain models provide a more expressive language (called MAVO annotations) compared to partial snapshots (which implements only annotation V and O from MAVO) 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.

Concrete models compliant with an uncertain model can synthesized by the Alloy Analyzer [23], or refined by graph transformation rules [22]. 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.

We believe that our contribution is novel in the context of view model synchronization in the sense that the effects of uni-directional and non-injective forward rules are reversed by mapping models, WF constraints and rules into first-order logic and then using iterative calls to a SAT-solver. Furthermore, the consistency criteria for the derived source models is stricter as it includes WF constraints of the source language (and not only consistency constraints of the transformation). Moreover, a fix partial model is specified upon a change in the target model using traceability links maintained during the forward propagation. Finally, iterative and incremental calls to logic solvers scales better then a full model generation run.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an incremental backward change propagation approach from view models to source models (in full details compared to [26]), which (1) provides a change partitioning technique to separate possibly affected and unaffected partitions of the source model, (2) transforms the source model partitions, the queries and WF constraints of the source language to a logic problem and (3) generates well-formed and consistent source model candidates by the Alloy Analyzer. This way, valid source candidates can be deduced in case of multiple view models and when backward transformation rules are not explicitly specified.

An initial experimental evaluation of our approach was carried out in the context of a health care model (taken from the CONCERTO European project), which demonstrated that (A) our incremental model generation approach scales much better compared to generating the full source model in a single call to a logic solver, but (B) its scalability is severely limited by the Alloy Analyzer (especially, by an initial internal conversion to a CNF form).

As future work, we plan to (i) prioritize the synthesized solutions, (ii) improve the calculation of the source model by calling multiple solvers to minimize the size of the solution or (iii) to develop a dedicated graph solver on top of SAT/SMT solvers as replacement of the Alloy Analyzer used currently in our approach.

7. REFERENCES

[1] CONCERTO ARTEMIS project. http://concerto-project.org/.

[2] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient model synchronization with view triple graph grammars. In *Modelling Foundations and Applications*, pages 1–17. Springer, 2014.

[3] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. Emf views: A view mechanism for integrating heterogeneous models. In *Conceptual Modeling*, pages 317–325. Springer, 2015.

[4] G. Callow and R. Kalawsky. A satisficing bi-directional model transformation engine using mixed integer linear programming. *Journal of Object Technology*, 12(1):1: 1–43, 2013.

[5] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: a bidirectional and change propagating transformation language. In *Software Language Engineering*, pages 183–202. Springer, 2010.

[6] C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró. Query-driven incremental synchronization of view models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and*

Orthographic Software Modelling, page 31. ACM, 2014.

[7] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering*, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[9] L. Gammaitoni and P. Kelsen. F-alloy: An alloy based model transformation language. In *Theory and Practice of Model Transformations*, pages 166–180. Springer, 2015.

[10] H. Gholizadeh, Z. Diskin, S. Kokaly, and T. Maibaum. Analysis of source-to-target model transformations in quest. In *Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with (MODELS 2015, Ottawa, Canada*, pages 46–55, 2015.

[11] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *Model Driven Engineering Languages and Systems*, pages 321–335. Springer, 2006.

[12] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, and T. Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software & Systems Modeling*, 14(1):241–269, 2015.

[13] T. Hettel. *Model round-trip engineering*. PhD thesis, Queensland University of Technology, 2010.

[14] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, pages 1–22, 2015.

[15] Á. Horváth, Á. Hegedüs, M. Búr, D. Varró, R. R. Starr, and S. Mirachi. Hardware-software allocation specification of ima systems for early simulation. In *Digital Avionics Systems Conference (DASC)*, Colorado Spings, Colorado, US, 10/2014 2014. IEEE.

[16] D. Jackson. Alloy Analyzer. http://alloy.mit.edu/.

[17] N. Macedo and A. Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.

[18] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction*, pages 289–313. Springer, 2004.

[19] OMG. MOF 2.0 Query/View/Transformation specification (QVT), version 1.1. http://www.omg.org/spec/QVT/1.2/.

[20] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In *Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063 of *LNCS*, pages 107–121. Springer Berlin-Heidelberg, 2008.

[21] R. Salay and M. Chechik. A generalized formal framework for partial modeling. In *Fundamental Approaches to Software Engineering*, volume 9033 of *LNCS*, pages 133–148. Springer, 2015.

[22] R. Salay, M. Chechik, M. Famelis, and J. Gorzny. A methodology for verifying refinements of partial models. *Journal of Object Technology*, 14(3):3:1–31, 2015.

[23] R. Salay, M. Famelis, and M. Chechik. Language independent refinement using partial modeling. In *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 224–239. Springer, 2012.

[24] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[25] O. Semeráth, A. Barta, A. Horváth, Z. Szatmári, and D. Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and Systems Modeling*, pages 1–36, 2015.

[26] O. Semeráth, C. Debreceni, Ákos Horváth, and D. Varró. Change propagation of view models by logic synthesis using SAT solvers. In *Int. Workshop on Bidirectional Model Transformations BX'16*, 2016. In press. Preprint available for the reviewers from https://inf.mit.bme.hu/en/research/publications/change-propagation-view-models-logic-synthesis-using-sat-solvers.

[27] O. Semeráth, A. Vörös, and D. Varró. Iterative and incremental model generation by logic solvers. *Fundamental Approaches to Software Engineering, 19th International Conference, FASE 2016*, 2016.

[28] P. Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, 2008.

[29] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015.

[30] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 164–173. ACM, 2007.