Quick fix generation for DSMLs

Ábel Hegedüs*, Ákos Horváth*, István Ráth*, Moisés Castelo Branco† and Dániel Varró*
*Budapest University of Technology and Economics
Budapest, Hungary
Email: {hegedusa,ahorvath,rath,varro}@mit.bme.hu
†University of Waterloo
Waterloo, Ontario, Canada
Email: mcbranco@gsd.uwaterloo.ca

***Abstract*—Domain-specific modeling languages (DSML) proved to be an important asset in creating powerful design tools for domain experts. Although these tools are capable of preserving the syntax-correctness of models even during free-hand editing, they often lack the ability of maintaining model consistency for complex language-specific constraints. Hence, there is a need for a tool-level automatism to assist DSML users in resolving consistency violation problems. In this paper, we describe an approach for the automatic generation of quick fixes for DSMLs, taking a set of domain-specific constraints and model manipulation policies as input. The computation relies on state-space exploration techniques to find sequences of operations that decrease the number of inconsistencies. Our approach is illustrated using a BPMN case study, and it is evaluated by several experiments to show its feasibility and performance.**

I. Introduction

In model-driven software engineering, domain-specific modeling (visual) languages (*DSMLs*) provide a popular rapid prototyping and generative technique to increase design reusability, speed up the development process and improve overall quality by raising the level of abstraction from software specific details to the problem domain itself. Today, DSMLs are deployed in many development toolchains to aid both software designers and domain experts, in order to integrate deep domain knowledge at an early design phase.

While many standard DSMLs are readily available (such as AUTOSAR for automotive software design, or process modeling languages such as BPMN [1] for service-oriented systems), custom modeling environments are developed in-house by many organizations to tailor development tools and code generators to their specific needs. This *language engineering* process relies on frameworks such as the Eclipse Modeling Framework (EMF [2]) or MetaEdit+ [3] that provide powerful tools for defining DSMLs and to automatically generate textual or graphical editors for creating domain models.

As domain-specific models are abstract and thus highly compact representations of the system-under-design, a key issue (that arises in both standardized and custom-made modeling environments) is *inconsistency management*. Inconsistencies are violations of the well-formedness and correctness rules of the language that may correspond to design errors, or violations of the company's policies and best practices. While domain-specific editors are usually capable of ensuring that elementary editing operations preserve syntactic correctness (by e.g. *syntax-driven editing*), most DSMLs include additional language-specific consistency rules that must also be checked.

In the current state-of-the-art of modeling tools, inconsistency management of complex rules focuses primarily on the *detection* of inconsistency rule violations, facilitated by dedicated constraint evaluators that take e.g. an OCL [4] expression and generate code that continuously scans the models and reports problematic model elements to the user. However, the *resolution* of these violations is mostly a manual task: the user may manually alter the model to eliminate the violations, but this can be a very challenging problem due to the complexity of the language or the concrete model. As a result, manually fixing a violation of one inconsistency rule may introduce new violations of other rules.

In *programming* languages, the concept of *quick fixes* (also called error correction, code completion) is a very popular feature of integrated development environments such as Eclipse or Microsoft Visual Studio, which aids programmers in quickly repairing problematic source code segments. This feature is deeply integrated into the programming environment and it can be invoked any time for a detected violation, giving the developer a list of fixing actions that can be applied instantly.

In the current paper, we propose to adapt this concept to domain-specific modeling languages. Our aim is to provide a domain-independent framework (that is applicable for a wide range of DSMLs), which can efficiently compute complex fixing action sequences even when multiple, overlapping inconsistency rule violations are present in the model. To capture inconsistency rules of a DSML, we use graph patterns that define declarative structural constraints. Our technique uses graph transformation rules to specify elementary fix operations (policies). These operations are automatically combined by a structural constraint solving algorithm that relies on heuristics-driven state-space exploration to find quick fix sequences efficiently. As a result, our approach provides high-level extensibility that allows both the language engineer and the end user to extend the library of supported inconsistency rules and fixing policies.

The rest of the paper is structured as follows. Section II introduces the problem of model editing for DSMLs and describes our BPMN case study used for illustration throughout the paper. In Section III, we give a more precise definition on quick fixes for DSMLs, describe the process for generating them and specify the architecture used for implementation. In Section IV, we show the application of the approach on the BPMN case study, while Section V contains our evaluation of the application. Finally, related work is discussed in Section VI and Section VII concludes the paper.

II. Case study: business process modeling

In this paper, we use the Business Process Model And Notation (BPMN [1]) as an illustrative case study. BPMN is a well-known and widely used standard, flowchart-like notation system for specifying business processes. BPMN supports the scoped modeling of both control and data flow; for control flow, activities, events, gateways (conditional decision, fork-join) may be used while data flow may be facilitated between activities and artefacts (e.g. data objects). All elements of a process may be organized into pools and swimlanes to create a structural breakdown according to e.g. organizational or functional rules (scopes).

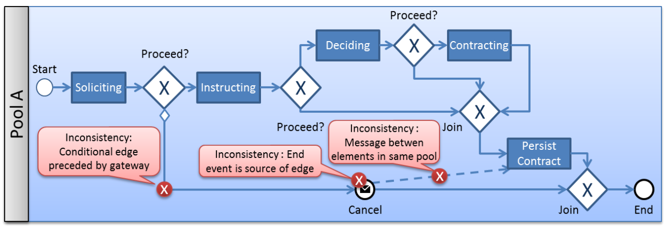

Pool A
Start
Soliciting
Proceed?
Instructing
Deciding
Proceed?
Contracting
Proceed?
Join
Persist Contract
Inconsistency: Conditional edge preceded by gateway
Inconsistency : End event is source of edge
Inconsistency : Message betwen elements in same pool
Cancel
Join
End

Fig. 1. Example BPMN process

Currently, there is a variety of BPMN-capable tools available (e.g. The MEGA Suite, Tibco Business Studio, or SAP NetWeaver BPM) that all use the standardized graphical notation (depicted in Figure 1), where activities (tasks) are represented by rectangles, events by circles, gateways by diamonds (rhombi) and sequence flow by arrows – inconsistencies are indicated by red circles). However, in our examples we use a graph-like *abstract syntax representation* that indicates the types and names of each BPMN node more explicitly in order to ease understanding.

Unfortunately, these tools do not address automated inconsistency resolution. For example, while the popular BPMN editor included in the Eclipse SOA Tools Platform suite [5] is able to detect a number of simple inconsistency rule violations, it does not guide the user in any way how to fix them.

Example 1 (Quick fix in a BPMN context) We give an intuitive example to clarify the concept of *quick fixes* applied in a domain-specific modeling context. *Inconsistency rules* in BPMN may describe process patterns that should not appear according to the standard (or custom design policies), e.g. because they would lead to incorrect behavior (when the process is executed). For instance, as message sending provides communication between different processes (which are modeled as pools), *messages between elements belonging to the same pool* are discouraged.

When an instance of such an inconsistency is detected in a BPMN process model (e.g. a given message is to be sent between two activities that belong to the same pool, as illustrated by the dashed arrow in Figure 1), the Eclipse editor places an error feedback marker in the model to indicate the location of the error. The user can only attempt to correct the error by a manual process, starting from guidance given by the (ad-hoc) textual report. One may attempt to remove the erroneous model configuration by performing a sequence of elementary operations (such as deleting an element, or altering its location), and then re-running the validation on the model to check whether the attempt was successful.

We aim to automate this lengthy and error-prone process by a *quick fix generator*. Given the error marker and a set of elementary fix operations (policies), the generator performs the try-and-test search (without permanently altering the model) and presents the possible solutions (if any) to the user. In the small example above, the generator may find that the violation can be removed by either deleting the message or moving the receiver to a separate pool, and present a list of these solutions to the user, where the (sequence of) correcting operations can be quickly executed in one compound operation, restoring the correctness of the model.

Challenges of quick fix generation

While simple fixing actions (such as deleting the ill-configured message) can be provided (in a hard-wired manner) by the programmer of the violation detection component (as available in e.g. the EMF Validation Framework [6]), our fix generator is a *more generic* solution as it can effectively deal with situations where (i) multiple errors are present in the model, possibly affecting an overlapping set of elements, and (ii) the fixing of individual errors may interfere with each other generating intermediate violations.

From the end-user perspective, our approach addresses the following *challenges/requirements*:

- *quick feedback to the user*: all fixing proposals should be calculated quickly to keep the interactive nature of the modeling process intact (if no proposal can be calculated, the user may wish to continue looking for a solution which takes more time).
- *offer the best fixes to the user*: the inconsistency resolution might have many solutions. The tool should pick the *best* options and present only those to the user, by discarding too complicated or "dangerous" ones (i.e. high number of modifications on a large part of the model).
- *keep model changes at a minimum*: all offered fix proposals should use conservative manipulation sequences that keep most of the model intact, in order to maintain the user's feeling of being in control of model manipulation.
- *support for local and global scope for fixes*: inconsistency rule violations are usually local in the sense that their scope is defined in terms of a few interconnected model elements. While a model may have many erroneous configurations scattered, a fixing proposal generator should allow the user to select the context to which solutions will be generated.

- *extensibility*: the supported library of inconsistency rules and (elementary) fixing policies should be easily extensible by the end users as well. This is a key feature for enforcing customized design standards that organizations or individuals can develop and tailor to their needs.

III. Problem formalization

A. Definition of Quick Fixes for DSMLs

First, we define the components of a DSML, which are foundational concepts in our approach (depicted in Figure 2).

Definition 1 The *metamodel* MM for a DSML includes the set of model element *types* of the domain, their *attributes* and *relationships* between model elements. A model conforming to the metamodel is called an *instance model* M.

Definition 2 A *query* q over a model M represents constraints that have to be fulfilled by a part of the instance model $M_0 \subseteq M$. Given an instance model M and a query q, a submodel that fulfills the query is called an injective *match* $m : q \mapsto M_0$ (or shortly, $q \overset{m}{\mapsto} M_0$), where M_0 is the context of the match m (denoted as $ctx(m)$).

Definition 3 *Inconsistency rules* r describe situations which should not appear in instance models and they are defined as queries over an instance model. The set of inconsistency rules defined for a DSML is denoted by R.

Definition 4 An *operation* o is a pair of $o = (p, \vec{s})$ with a set of symbolic parameters p, a sequence of elemental model manipulation steps $\vec{s}$, which specify how the model is modified by adding, removing or changing parts of it similar to syntax-directed editing. The set of all operations for a DSML is denoted by O.

Definition 5 The *execution* of the operation o on M_{old} $(M_{old}, b) \overset{o}{\to} M_{new}$ modifies the model M_{old} according to the elemental steps, based on the bindings b from symbolic parameters to model elements, resulting in model M_{new}.

Definition 6 The *DSML* is a triplet $DSML = (MM, R, O)$ containing the metamodel, the set of inconsistency rules and operations defined for the language.

Next, the above definitions are used for specifying the concepts used in our quick fix generation approach.

Definition 7 An inconsistency rule r is *violated* by an instance model M if there exists at least one match (*violation*, v) $r \overset{v}{\mapsto} M_0$, where $M_0 \subseteq M$. The set of violations that contain a given model element e is denoted by $V_e(M) = \{v | e \in ctx(v)\}$, and finally, the set of all violations in M by $V(M)$.

Definition 8 An instance model M of metamodel MM, is *inconsistent* if one or more inconsistency rules are violated by (a part of) M, formally $\exists r \in R, \exists v \in V(M) : r \overset{v}{\mapsto} M$. The *total number of violations* in M for all inconsistency rules is denoted by $|V(M)|$, while the *number of local violations* for a given model element $e \in M$ is denoted by $|V_e(M)|$.

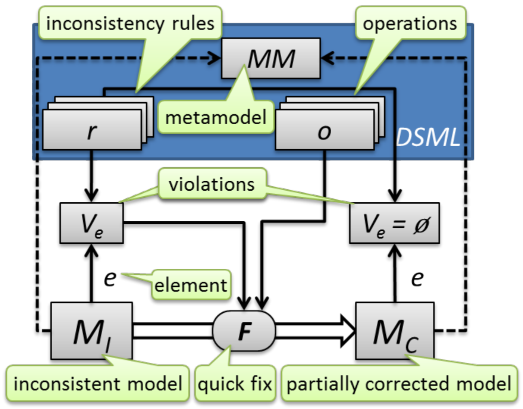


Fig. 2. Quick fixing an inconsistent model

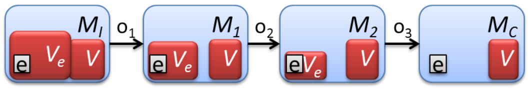


Fig. 3. Application of a Quick fix

Quick fixes were defined informally as a sequence of model manipulation operations, which change an inconsistent model in a way to eliminate constraint violation instances.

Definition 9 A *quick fix* $M_I \overset{F}{\Rightarrow} M_C = (M_i, b_1) \xrightarrow{o_1} M_1, (M_1, b_2) \xrightarrow{o_2} M_2, \ldots, (M_{n-1}, b_n) \xrightarrow{o_n} M_C$ for a model element e is an ordered sequence of operations executed on an inconsistent model M_I, resulting in a *partially corrected model* M_C, with the following conditions:

- $\exists v_I \in V(M_I) : e \in ctx(v_I)$; There exists a violation v_I in the incositent model M_I, where the model element e is in the context of v_I.
- $\nexists v_C \in V(M_C) : e \in ctx(v_C)$; There is no violation v_C in the partially corrected model M_C, where the model element e is in the context of v_C.
- $|V(M_I)| > |V(M_C)|$; F decreases the total number of violations in the model

Figure 3 illustrates how the application of the operations in a quick fix affects the instance model by eliminating the violations step-by-step. Initially, there are several model elements included in $V_e(M_I)$ in the inconsistent model M_I. After applying the first operation o_1 from the quick fix, the resulting M_1 where some of the violations may be fixed already. By executing the rest of the operations o_2, o_3 in the quick fix, the final model M_C contains no violations for the selected element e. It is important to note that not all violations in the model are eliminated by the quick fix, only those that contained the selected model element e. However, the total number of violations $|V(M_I)|$ decreases.

In order to map our approach to a given specific modeling environment, the generic definitions (such as query and operation) are mapped to the well-known formal technique of graph

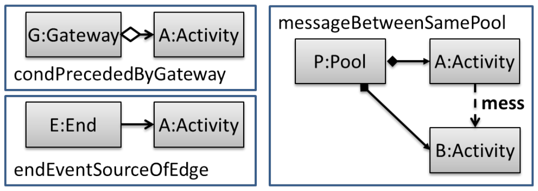


Fig. 4. Inconsistency rules for BPMN

transformation [7] with available, extensive tool support.

B. Graph transformation as modeling formalism

In this paper, both the metamodel and the instance models are specified by (typed and attributed) graphs, with special *instance of* relations between elements and types.

Queries (e.g. inconsistency rules) are defined using *graph patterns* (or graph constraints) [8] including structural, attribute, nested and negated constraints (the last describing cases where the match of a pattern is not valid) and logic composition operators. Querying is performed by graph *pattern matching* [9] that returns matches in the form of graphs.

Graph transformation (GT) [7] provides a declarative language for defining the manipulation of graph models by means of GT rules. At GT rule consists of (i) a left-hand side (LHS), (ii) a right-hand side (RHS) graph, and (iii) arbitrary number of negative application conditions (NAC) attached to the LHS. Model manipulation is carried out by replacing a matching of the LHS in the model by an image of the RHS. This is performed in two phases. In the pattern matching phase, matchings of the LHS are sought in the model and the absence of the graph structures of NACs is checked and ensured. In the updating phase, the selected matching parts are modified based on the difference of LHS and RHS.

The foundation of our approach is similar with [10]–[12] in using graph transformation based techniques for specifying inconsistency rules and model generation.

C. Quick fixes for BPMN

The metamodel of the BPMN language is defined in EMF and is incorporated in the Eclipse BPMN Modeler tool [5]. We used this metamodel for specifying inconsistency rules and model manipulation operations for the language.[1]

1) Inconsistency rules: Figure 4 shows three of such rules as graph pattern using a simple graphical notation. Model elements are depicted with rectangles and relationships with arrows, while the name of the element and the its type are separated by colons.

Conditional Edge Preceded By Gateway: This inconsistency rule (*condPrecededByGateway*) specifies the situation where a *Gateway G* is the source of a conditional sequence edge (depicted as a continuous arrow with a empty diamond source end) with the target an arbitrary activity *A*. If there are two elements in the model, which can be matched to *G* and *A* (respecting the type restriction), then it is a violation. In the case study, such a violation exists on the edge between the first *Proceed?* gateway and the *Cancel* event (see Figure 1).

[1]The full list of inconsistency rules and operations can be found at http://viatra.inf.mit.bme.hu/publications/quickfix.

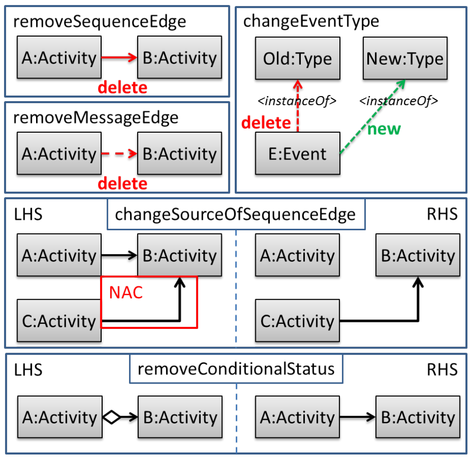


Fig. 5. Operation definitions for BPMN

End Event Source of Edge: This inconsistency rule (*endEventSourceOfEdge*) specifies the situation where an *End* event (*E*) is the source of a sequence edge (with the target an arbitrary activity *A*). If there are two elements in the model, which can be matched to *E* and *A* (respecting the type restriction), then it is a violation (e.g. the sequence edge starting from *Cancel* in the case study as illustrated in Figure 1).

Message Between Elements in Same Pool: The inconsistency rule *messageBetweenSamePool* describes the situation outlined in Example 1. Activities *A* and *B* are both elements in *P* (as defined by the arrow with the diamond shaped end). A violation exists, if three elements matching *A*, *B* and *P* can be found in the model (e.g. the message edge leading from *Cancel* to *Persist Contract* in the case study, see Figure 1).

2) Operation definitions: Operations for manipulating BPMN models can be defined specifically for each metamodel type (e.g. create Parallel Gateway) or generically to decrease the total number of operations (e.g. create element with type). Figure 5 shows five operations for various modifications using graph transformation rules. Negative application conditions are depicted with red rectangles. When possible (the upper three in Figure 5), the LHS and RHS are merged and model parts removed by the operation are annotated with *delete*, while parts that are created are annotated with *new*.

The rules *removeSequenceEdge* and *removeMessageEdge* remove an existing sequence or message edge from between two activities, respectively, while *changeEventType* changes the type of an event element to the given type, depicted as replacing the *instanceOf* relation (e.g. the type of an event is changed from End to Start).

The *changeSourceOfSequenceEdge* moves the source end of a sequence edge between activity *A* and *B*, so that the new source will be activity *C*. It also restricts the application by stating that *C* must not have an existing sequence edge from the same element (*NAC*). Finally, the *removeConditionalStatus* rule removes the conditional status from a sequence edge

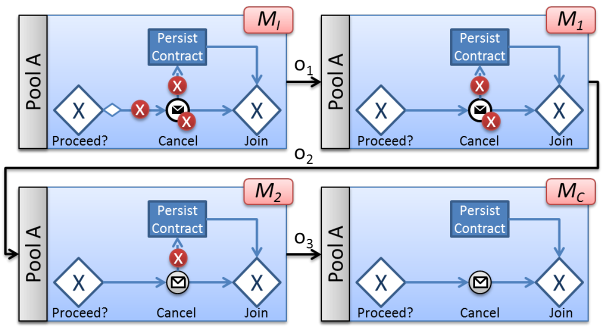


Fig. 6. Application of a fix

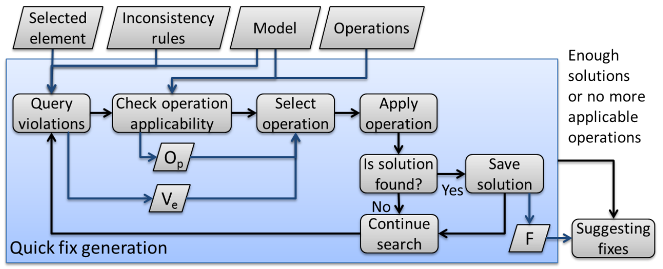


Fig. 7. Overview of the Quick fix generation

between activity *A* and *B*.

3) Application of a quick fix: The quick fixes (generated by the techniques detailed in Section IV) should contain enough information to allow deterministic application over the model (to a degree that violations are eliminated in all cases). A selected quick fix is applied to the model by taking each operation in order and execute it with the stored input bindings (illustrated in Figure 6).

The application starts from the inconsistent model M_I (upper left) and first removes the conditional attribute from the sequence edge between *Proceed?* and *Cancel* resulting in M_1 (upper right). Next, the type of event *Cancel* is changed to intermediate (since it has incoming edges), thus leading to M_2 (lower left). Finally, the message edge between *Cancel* and *Persist Contract* is removed (since they are in the same pool). In the resulting model M_C (lower right) no violations remain on *Cancel*.

IV. Generation of Quick Fixes

A. Constraint satisfaction problem over models

Quick fixes are generated directly on the DSML by using a state-space exploration approach capable of solving structural constraint satisfaction problems over models, also called CSP(M) [13]. In CSP(M), problems are described by: (i) an *initial model* representing the starting point of the problem, (ii) *goals* that must be met by a solution model defined as graph pattern and finally, – as a distinctive feature from traditional CSP approaches – (3) *labeling rules*, which explicitly define permitted operations as graph transformation rules.

In CSP(M) a state is represented by the underlying model – the starting state is the initial model – and a transition between states is an application of a labeling (GT) rule. To explore this state space the CSP(M) solver uses guided traversal algorithms [14] to find a valid solution model that satisfies all goals and minimize the visited states (*effective selection* challenge). Based on the traversal algorithm (which supports both backtracking and cycle detection) the output of the solver can be a single or multiple solution models and the sequence of labeling rules applied to achieve these models.

In order to generate quick fixes using the CSP(M) approach, we encoded the negated inconsistency rules as goals, the allowed operations as labeling rules and used the model from the editor directly as the initial model. This way the CSP(M) solver tries to find a model that satisfies each inconsistency rule using the allowed operations. The main advantage of using the CSP(M) approach is that it allows to define the quick fix problem directly over the DSML model and it does not need any mapping or abstraction to other mathematical domain as used in similar model generation approaches [15], [16].

However, it is important to mention that as the CSP(M) framework is extendable with custom solver algorithms, we modified its solver algorithm by restricting (1) the application of operations and (2) the solution checking to the violations of the selected model element. This is important in order to *support local fixing scopes*. Furthermore, the solver supports both (i) breadth-first and (ii) depth-first search, and (iii) parameterizable limits on solution length and number of alternative solutions. We defined priorities for the operations, which are taken into consideration during the iteration, thus higher priority operations are executed first. By giving higher priority to more conservative operations, the *conservativity* challenge is addressed, since solutions containing these operations are explored before others.

B. Quick Fix Generation Process

The process of quick fix generation depicted in Figure 7 consists of the following steps:

1) *Query violations* The quick fix generation starts with selecting an element e in the model to work as a scope for inconsistency rules. Next, all the violations that include e are queried from the model for each inconsistency rule, initializing the set of violations $V_e(M)$.
2) *Check operation applicability* First, all operations are checked for executability (i.e. whether they can be executed at all) and executable operations are collected in a list Op.
3) *Select operation* The state space exploration then iterates through Op and checks the possible input parameter bindings against elements in the matches for violations in $V_e(M)$ (e.g. a sequence edge violating *endEventSourceOfEdge* will be part of the possible inputs of *removeSequenceEdge*, see Figure 4 and Figure 5).
4) *Apply operation* If it finds a matching input, then the operation is applied to the model with the given input

resulting in a new model state.

5) *Is solution found?* The new model state may be a correct model, this is checked by re-executing the query against the model again to get $V_e(M)$. If $V_e(M)$ is empty, then the total number of violations and violations on elements in the original $V_e(M)$ are checked as well.
   a) *Save solution* When a valid quick fix is found, the trace (with the executed operations and input bindings) is saved to a solution list. Quick fix generation terminates once a predefined number of solutions are found.
   b) *Continue search* If the new model state is not a correct model or further solutions are required, the next applicable operation is selected. The state-space exploration terminates if there is no applicable operation within the limited search space.
6) *Suggesting fixes* The solutions are then suggested for inspection to the user who may choose one quick fix to be applied on the model. If no solutions were found, this information is displayed instead.

It is important that the set of inconsistency rules and operations are easily extensible by the end users (*extensibility* challenge). In our approach, these definitions are not hard-coded into the solver and can be modified using the graph transformation formalism. The CSP(M) framework also supports dynamic handling of inconsistency rules and operations, e.g. to generate solutions for different subsets of operations.

C. Implementation architecture

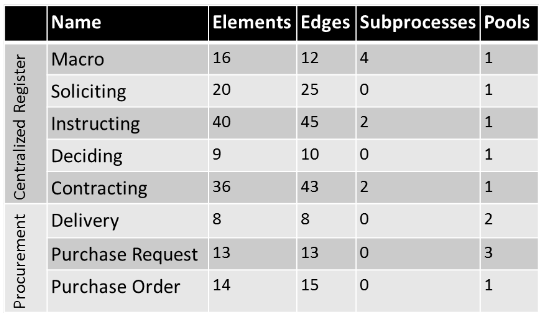
We implemented our quick fix generation approach using the VIATRA2 model transformation framework [17], which provides metamodeling capabilities and supports model transformations based on the concepts of graph transformations and abstract state machines. Its incremental pattern matcher is used as a powerful query engine [18].

The state space exploration part of the approach is executed by the constraint satisfaction engine presented in [13], where operations and inconsistency rules are used in solving a constraint satisfaction problem over the input model. BPMN processes can be developed in the Eclipse BPMN modeler tool [5]. Since both the modeler tool and VIATRA2 are Eclipse technologies, we could seamlessly integrate the quick fix generator to be usable directly from the modeling tool, just like quick fixes work in integrated development environments.

V. EVALUATION

A. BPMN models used for evaluation

We evaluated the approach for scalability on two real BPMN projects, obtained from an industrial partner from the banking sector. One project is a corporate customer registering workflow, composed of five processes and approximately 250 model activities in total. The other project is a corporate procurement workflow, composed of three processes and around 70 model activities. The projects were selected among others available from the partner by the following criteria: 1 – they can be converted to the Eclipse BPMN Modeler editor with minor changes, since they were originally modeled in another tool; 2 – they allow to explore all the errors described for the case study by containing the necessary modeling scenarios, such as multiple pools and message flows; 3 – all eight BPMN models are classified as typical real-life BPMN processes [19]. The name and size of the different processes are shown in Figure 8.



	Name	Elements	Edges	Subprocesses	Pools
Centralized Register	Macro	16	12	4	1
	Soliciting	20	25	0	1
	Instructing	40	45	2	1
	Deciding	9	10	0	1
	Contracting	36	43	2	1
Procurement	Delivery	8	8	0	2
	Purchase Request	13	13	0	3
	Purchase Order	14	15	0	1

Fig. 8. Processes in the case study

B. Evaluation environment and method

The evaluation was carried out by adding inconsistencies to each process and running the quick fix generation approach independently. We performed measurements[2] multiple times for each test case including different total and local number of inconsistencies in the model.

The measurement of a given test case was done as follows: the inconsistent BPMN model is loaded into VIATRA2, the inconsistency rules and operations are added to the framework, the quick fix engine is initialized and time measurement is started. Next, the quick fix engine looks for three different solutions and gathers them in a list, once it is done the time measurement is stopped. Finally, the results are saved and the framework is disposed to return the environment to the initial state.

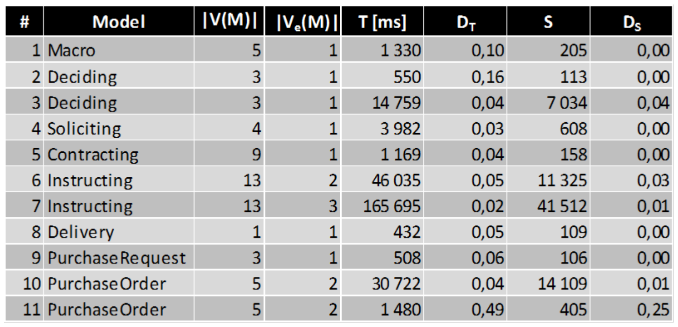
C. Evaluation of results

The table in Figure 9 shows the results of our measurements using the case study models. For each model we measured the performance for the given number of total and local inconsistencies. For each case, we measured the number of visited states and the time of quick fix generation. Finally, measurement results are given with the mean values along with deviations.

We made the following observations based on the results from the different models:

One local violation $(\#1-5,8,9)$*:* In these cases fixing is possible in a reasonable time even during editing, since the quick fix generation takes less than 4 seconds in all cases except $\#3$, where finding three different solutions takes 15 seconds. Although the deviation of runtime is significant in some cases (50% for $\#11$), it causes only a few seconds longer runtime.

[2]All measurements were carried out on a computer with Intel Centrino Duo 1.66 GHz processor, 3 GB DDR2 memory, Windows 7 Professional 32 bit, Eclipse 3.6.1, EMF 2.6.1, BPMN 1.2, VIATRA2 3.2 (SVN version)



#	Model	|V(M)|	|V$_e$(M)|	T [ms]	D$_T$	S	D$_S$
1	Macro	5	1	1 330	0,10	205	0,00
2	Deciding	3	1	550	0,16	113	0,00
3	Deciding	3	1	14 759	0,04	7 034	0,04
4	Soliciting	4	1	3 982	0,03	608	0,00
5	Contracting	9	1	1 169	0,04	158	0,00
6	Instructing	13	2	46 035	0,05	11 325	0,03
7	Instructing	13	3	165 695	0,02	41 512	0,01
8	Delivery	1	1	432	0,05	109	0,00
9	PurchaseRequest	3	1	508	0,06	106	0,00
10	PurchaseOrder	5	2	30 722	0,04	14 109	0,01
11	PurchaseOrder	5	2	1 480	0,49	405	0,25

Fig. 9. Evaluation results ($|V(M_I)|$: total number of violations, $|V_e(M)|$: max. no. of violations per element, T: time [ms], D_T: standard deviation of time, S: no. of visited states, D_S: standard deviation of visited states)

Locality $(\#3, 5)$*:* The higher number of local violations for the selected element leads to slower fix generation, while the total number of violations in the model does not affect performance. Generating quick fixes for one violation in case $\#3$, where there are only 3 violations, is 15 times slower than for case $\#5$, which has 9 violations. This is a direct consequence of our approach, which applies operations on elements specified by violations of the selected element.

Multiple local violations $(\#6, 7, 10, 11)$*:* Finding quick fixes in these cases is possible but takes a considerable amount of time, especially if more than one solution is generated. For example, finding three solutions for case $\#7$ takes almost 3 minutes and the exploration of more than 40000 states. However, we found that even with complex DSMLs such as BPMN visiting one state only takes between $2ms$ and $4ms$, independently of the number of states explored before (at least in the scope of the measurements this held).

First solution $(\#6, 7)$*:* We found that often the quick fix generation finds a solution early on even for large models and multiple local violations, but then the majority of runtime is spent looking for alternative solutions.

To summarize, it is feasible to generate quick fixes for DSMLs, in most cases without interrupting the editing process. Our approach finds alternative solutions for local violations without considerable deviation between executions and the memory usage remains in the acceptable range (between *30MB* and *200MB* in all cases). Although in some cases the fix generation for multiple violations is costly, the generation can be interrupted without waiting for multiple solutions, thus resulting in an almost anytime-like algorithm.

VI. Related work

The quick fix generation approach presented in our paper deals with correcting local inconsistencies in models using predefined model manipulation operations. Similar approaches are found in the areas of *model construction and syntax-directed editing* and *inconsistency handling* of models. In this section we place our approach with regards to existing works.

Model construction and syntax-directed editing: Model construction deals with creating consistent models through a series of operations. In [16] models are constructed by providing hints to the designer in the form of valid operations, which are calculated using logic constraints and reasoning algorithms, to maintain global correctness. Mazanek [20] introduced an auto-completion method for diagram editors based on hyper-edge grammars, where model construction proposals are generated for incomplete models (although not inconsistent). In [10] they extended the approach for generating correctness-preserving operations for diagram editing, by identifying irrelevant or incorrect operations. This approach uses local scopes for operations (i.e. the user selects the elements where auto-completion is desired), similarly to our approach. In [15] models are transformed to Alloy specifications and automatic completion suggestions are calculated on the derived model.

These approaches present construction methods, where models are built in a monotonously increasing way, while our approach can also remove parts of the model when inconsistency handling requires it. Furthermore, these approaches translate models into some analysis formalism for generating operations, while our method works directly on the original models. Finally, the extensibility (adding and removing constraints and operations) of these approaches is limited by using derived analysis models, while our approach supports dynamic handling of inconsistency rules and operations.

Inconsistency handling: Fixing common inconsistencies in design models (such as UML) are regarded as an important challenge and several techniques were proposed for addressing it. Egyed et al. [21] presents an inconsistency resolution method for UML models, where possible consistent models are generated based on predefined inconsistency rules. The approach is restricted to operations, which change only one element at a time, while our quick fix generation approach allows the definition of complex operations.

[22] proposes an approach for generating *repair plans* is presented for EMF-based UML models. It uses *generator functions* as operations and extends inconsistency rules with information about the causes of inconsistencies, to make their search algorithms more efficient. It supports the restriction of the maximum size of the explored state-space and fitting repairs to most recent inconsistencies similarly to our approach.

The inconsistency resolution problem is solved using automated planning in [23] without manually defining operations. Instead, it generates valid models by translating design models to logic literals and executing analysis on this derived model. While planning is similar to heuristic-driven exploration, our approach does not use derived models.

Nentwich et al. [24] define a distributed approach, where operation definitions (resolving one inconsistency at a time) are generated automatically from inconsistency rules described in a logical language and incremental consistency checking is performed directly on the design models. However, the approach handles inputs as XML documents, while our technique works directly over models.

Graph transformation is also frequently used for handling inconsistencies. In [25] inconsistency checking and operations are performed on the model in response to user interface events with actions defined with triple-graph grammar. In [11] models are checked and corrected based on modeling guidelines defined using graph transformation rules to help users in resolving a large number of violations automatically. Mens [12] proposes critical pair analysis of inconsistency rules and operations defined as graph transformation rules for improving inconsistency management by detecting incompatible resolutions and ordering solutions.

All of these approaches are similar to our quick fix generation method in using graph transformation as a formalism for capturing inconsistency rules and operations. However, our approach uses heuristic-driven state-space exploration and is able to generate quick fixes with local scopes independently of the total number of violations in other parts of the model.

VII. Conclusion

In the paper, we elaborated a novel approach for generating quick fixes (as instantly applicable, complex corrections to consistency violations) for DSMLs. Our technique is based on a heuristics-driven state exploration algorithm for solving structural constraints directly over domain-specific models. We have assessed and justified the scalability of our implementation using real-life models provided by an industrial partner.

Our implementation (accessible as an open source add-on to VIATRA2) fulfills the challenges established in Section II as follows: (i) as quick fixes are generated in an anytime fashion, the first solution is usually quickly found and presented to the user; (ii) the heuristics-guided algorithm automatically selects the best, conservative solutions, with additional fine-tuning possible as future work; (iii) as demonstrated in the evaluation, the engine is capable of generating solutions for a given local scope as well as the entire model; (iv) our high abstraction level, declarative formalism allows both the language engineer and the end user to build extensible inconsistency rules and operations at runtime.

Regarding future research, we plan to investigate various heuristics (such as dependencies between rules) as well as different solver algorithms, to extend the scalability even further. A very promising idea is to add the ability to *learn* user-selected quick fixes to a runtime knowledge base, and reuse such knowledge for consequent solution generation. This could be investigated in a model versioning and conflict management case study. Technologically, we aim to develop our current tool further to support the generation of quick fixes over generic EMF models, to enable the integration of this technology to all existing Eclipse-based modeling tools.

Acknowledgment

This work was partially supported by the SecureChange (ICT-FET-231101) and CertiMoT (ERC_HU_09-1-2010-0003) projects and the Janos Bolyai Scholarship. We would like to thank the Bank of the Northeast of Brazil (Banco do Nordeste – BNB) for providing the case study.

References

[1] Object Management Group, "Business Process Model and Notation (BPMN) Version 1.2," http://www.omg.org/spec/BPMN/1.2/.

[2] The Eclipse Project, "Eclipse Modeling Framework Project," http://www.eclipse.org/emf.

[3] J.-P. Tolvanen and M. Rossi, "MetaEdit+: defining and using domain-specific modeling languages and code generators," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 92–93.

[4] Object Management Group, "Object Constraint Language (OCL)," http://www.omg.org/spec/OCL/.

[5] SOA Tools Platform, "Eclipse BPMN Modeler," http://www.eclipse.org/bpmn/.

[6] The Eclipse Project, "EMF Validation Framework," http://www.eclipse.org/modeling/emf/?project=validation.

[7] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook on Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999, vol. 2: Applications, Languages and Tools.

[8] F. Orejas, H. Ehrig, and U. Prange, "A logic of graph constraints," in *Fundamental Approaches to Software Engineering (FASE)*, 2008, pp. 179–198, LNCS 4961, Springer.

[9] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, "Incremental pattern matching in the viatra model transformation system," in *Proceedings of the Third International Workshop on Graph and model transformations*. ACM, 2008, pp. 25–32.

[10] S. Mazanek and M. Minas, "Generating correctness-preserving editing operations for diagram editors," in *Proc. of the 8th Int. Workshop on Graph Transformation and Visual Modeling Techniques. ECEASST*, vol. 18, 2009.

[11] C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer, "Checking and enforcement of modeling guidelines with graph transformations," in *Applications of Graph Transformations with Industrial Relevance*, 2008, pp. 313–328, LNCS 5088, Springer.

[12] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," in *Proc. of Model Driven Engineering Languages and Systems*, 2006, pp. 200–214, LNCS 4199, Springer.

[13] Á. Horváth and D. Varró, "Dynamic constraint satisfaction problems over models," *Software and Systems Modeling*, 11/2010 2010.

[14] Á. Hegedüs and D. Varró, "Guided state space exploration using back-annotation of occurrence vectors," in *Proceedings of the Fourth International Workshop on Petri Nets and Graph Transformation*, 2010.

[15] S. Sen, B. Baudry, and H. Vangheluwe, "Towards domain-specific model editors with automatic model completion," *Simulation*, pp. 109–126, 2010, 86(2).

[16] M. Janota, V. Kuzina, and A. Wasowski, "Model construction with external constraints: An interactive journey from semantics to syntax," in *Proceedings of the 11th Int. Conf. on Model Driven Engineering Languages and Systems*, 2008, pp. 431–445, LNCS 5301, Springer.

[17] A. Balogh and D. Varró, "Advanced model transformation language constructs in the VIATRA2 framework," in *ACM Symp. on Applied Computing (SAC 2006)*. Dijon, France: ACM Press, 2006, p. 1280–1287.

[18] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró, "Experimental assessment of combining pattern matching strategies with VIATRA2," *Journal of Software Tools in Technology Transfer*, 2009.

[19] P. Gilbert, "The next decade of BPM," 2010, keynote at the 8th International Conference on Business Process Management.

[20] S. Mazanek, S. Maier, and M. Minas, "Auto-completion for diagram editors based on graph grammars," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008*. IEEE, 2008.

[21] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in uml design models," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 99 –108.

[22] M. Almeida da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *Advanced Information Systems Engineering*, B. Pernici, Ed., 2010, pp. 348–362, LNCS 6051, Springer.

[23] J. Pinna Puissant, T. Mens, and R. Van Der Straeten, "Resolving Model Inconsistencies with Automated Planning," in *Proceedings of the 3rd Workshop on Living with Inconsistencies in Software Development*. CEUR Workshop Proceedings, 2010, pp. 8–14.

[24] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003, pp. 455 – 464.

[25] E. Guerra and J. de Lara, "Event-driven grammars: Towards the integration of meta-modelling and graph transformation," in *Graph Transformations*, 2004, pp. 215–218, LNCS 3256, Springer.