User-defined Signatures for Source Incremental Model-to-text Transformation

Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos

Department of Computer Science, University of York
Deramore Lane, Heslington, York, YO10 5GH, UK.
[bjo500, louis.rose, dimitris.kolovos]@york.ac.uk

Abstract. Model-to-text (M2T) transformation is an important part of model driven engineering, as it is used to generate a variety of textual artefacts from models, such as build scripts, configuration files, documentation and code. Despite the importance of M2T transformation, building M2T transformations that scale with the size of the input model(s) remains challenging because most contemporary M2T transformation languages do not provide adequate support for incremental transformations. We have previously proposed the use of automatic signatures, as a technique for source incremental transformations. In this paper, we introduce user-defined signatures, which outperform automatic signatures. We perform a comparative analysis of user-defined signatures with automatic signatures, and non-incremental transformation by application to an existing M2T transformation.

1 Introduction

Model-to-Text (M2T) transformation is a model management operation that involves generating text (e.g., source code, documentation, configuration files, reports, etc.) from models. As M2T transformations become increasingly popular for generating textual artefacts in software projects, so also is the concern for building scalable M2T transformations [1]. According to Bennett et. al. [2], software evolution is inevitable and it involves activities that are necessary to fulfil the requirements of end-users. However, software evolution incurs costs associated with finding a subset of changed parts of the system model, analysing the impact of the change, implementation of the change, re-validation of the system [3]. For example, re-generation of text files upon making changes to a source model should not take as much time as it took to generate the text files in the first instance, and the process of re-generation should also be devoid of redundant re-computations, i.e., files that are not affected by changes need not be re-generated.

In our previous work [4], we proposed *signatures* for constructing efficient, scalable M2T transformations. Signatures can be used to detect changes in source model(s) and limit the execution of a transformation to the parts of the transformation that are affected by the change(s). Signatures must be derived from the transformation that is to be executed incrementally, and we have previously

proposed automation for deriving signatures (which we now term *automatic signatures* and discuss further in Section 3.3). In this paper, we reiterate and then address the shortcomings of automatic signatures via *user-defined signatures*. User-defined signatures have the further advantage of being more efficient than automatic signatures (Section 5).

2 Related Work

Model transformation has been described as the heart and soul of MDE [5]. Although little work has been done on incremental M2T transformation, there have been a few published techniques on incremental model-to-model (M2M) transformation. For example, Hearnden et. al. in [6] proposes an incremental method which represents the trace of a transformation execution as a tree. The Hearnden approach maintains an entire transformation context throughout all transformation executions, which allows propagation of changes between source and target models by re-executing the transformation on computed model deltas. Other incremental M2M approaches like PMT [7] synchronises models via trace links, which contain information relating to the provenance of target model elements with respect to source model elements. PMT is a rule-based M2M transformation language and the transformation engine uses identifiers to match source model elements to target model elements.

To the best of our knowledge, Xpand[1] is the only contemporary M2T language that supports source incremental transformation. Incremental generation in Xpand is a threefold process: generating trace links; performing model differencing; and analysing the difference model with respect to generated files. The generated trace links specify how source model elements are mapped to generated files. The difference model enables the transformation engine to identify the elements of the model that have changed. Model differencing is achieved in one of two ways: either by listening to changes made by the user in a model editor, or by comparison of the current and previous versions of the input model. Once the difference model is constructed, impact analysis is performed to determine which changed model elements are used in which templates. A template is re-executed if it uses a model element that has changed. The approach to incrementality employed by Xpand cannot utilise additional information about change, which might be known only to the developers. As we shall see in Section 3.3, user-defined signatures can utilise domain-specific information to increase the efficiency of incremental transformation.

3 Background: Incremental M2T

Incrementality in software engineering refers to the process of reacting to changes in an artefact in a manner that minimises the need for redundant re-computations.

[1] `http://eclipse.org/modeling/m2t/?project=xpand`

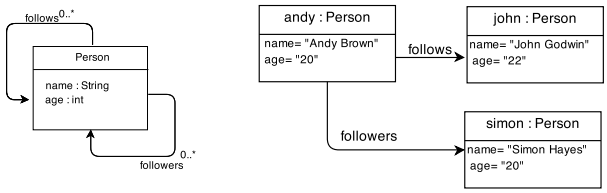
Generally, incremental model transformations reduce the amount of time and computation expended on propagating changes from inputs to outputs.

In the context of M2T transformation, incrementality provides mechanisms for the transformation engine to only re-invoke templates that are affected by changes to the input model. Incrementality in M2T transformation is categorised into 3 types [8]: user-edit preserving, target, and source incrementality. User-edit preserving incrementality prevents loss of user-crafted content by mixing generated text with manually written text. Target incrementality updates already generated targets with output from the current transformation execution after having invoked all templates. Source incrementality, unlike target incrementality, isolates modified model elements and invokes only those templates that are affected by the changes.

Many contemporary M2T transformation languages support user-edit preserving and target incrementality, but not source incrementality. Developing a sound theory of source incrementality is an open research challenge, and is crucial for significantly improving the efficiency of transformations that are complex (i.e., operate on large or densely connected models).

3.1 Running Example

In the following section, we demonstrate the use of signatures in an M2T template language. Figure 1 is a simplified model of a person's contact list on a social media platform (e.g. Twitter), which consists of: persons, the people that a person follow (follows), and that people that follow a person (followers). The follows - followers relationship is asymmetric: following a person does not imply that they must follow you. The model serves as input to the template in Listing 1.1. The generation of a person's follower list or following list represents a simple example of a process that would benefit from incremental generation. In the remaining sections, we demonstrate the use of signatures with our running example.



(a) Metamodel. (b) Example input model.

Fig. 1. Example input model.

3.2 Signature-Based Source Incrementality

Signatures are concise, lightweight proxies for templates that indicate whether or not a change to an input model will affect the output of a template. Signatures can be used to detect changes in input model(s) and limit the execution of a transformation to the templates that are affected by the change(s). Signatures represent a dynamic identifier for a model element in relation to a particular template that consumes data from the model element. The composition of a model element's signature depends on what attributes of the model element are accessed in a template. For example, in Figure 1, a person has *age* and *name* attributes. Consider the template shown in Listing 1.1, that is used to generate a person's followers and followed list from the person model in Figure 1. The template accesses the name attributes of the person, the person's followers and persons followed by the person. Therefore, the signature of *person* with respect to the template can be the values of the properties that are accessed by the template: $person.name$, $person.follows.name$ and $person.followers.name$. Note that the *age* attribute does not form any part of the signature, as this template is not dependent on the value of the any person's *age*. That is to say, a signature is a proxy for a specific template, and captures only those parts of the model that are used by that template.

```
1  [template public personToFile(p : Person)]
2  [file (p.name)/]
3  Person [p.name] has the following followers:
4  [p.followers.name] and follows the following persons:
5  [p.follows.name].
6  [/file]
7  [/template]
```

Listing 1.1. Simple template-based M2T transformation template specified in OMG MOFM2T syntax

We represent a signature as a sequence. Each element in the sequence is either a primitive value, or a further sequence. During the computation of a signature, each model element property value that makes up the composition of a signature is added to the signature sequence. For instance, considering the input model in Figure 1, suppose that a *person*'s signature is calculated from the name of person and the name attribute of person's followers and persons followed by the person. The initial signature for 'andy' using a flat structure will be the sequence: `{"Andy Brown", {"John Godwin"}, {"Simon Hayes"}}`. Note that the second and third elements of the sequence are also sequences because the follows and followers references are multi-valued. At the end of each transformation execution, the signatures (i.e., sequences) are persisted in non-volatile storage, such as a relational database or XML document.

During the first execution of an M2T transformation, a template's signature is evaluated each time the template is invoked, and the resulting signature value

is written to disk along with a unique identifier (typically, the model element's id) for the model elements that produced that signature value. In subsequent executions of the M2T transformation, the previous signature values are used in determining which templates need to be re-executed, and for which model elements. More specifically, a change in the model results in a signature value that differs from the equivalent signature value in the previous execution of a template on that model element. A signature value that has changed indicates that a template must be re-executed on a model element in order to propagate the change (to the generated text).

We have seen how signatures are stored and how they are used, but not how they are computed. The subsequent sections discuss two approaches to styles of signature which differ by the way in which that they are computed: automatic signatures and user-defined signatures.

3.3 Automatic Signatures

Automatic signatures are computed by concatenating the dynamic text-emitting sections of a template. The transformation engine strips a template of its static sections and invokes the template made up of only the dynamic sections, essentially capturing property accesses of model elements specified or expressed in templates. The templates are analysed at runtime. For example, the automatic signature computed by executing template in Listing 1.1 on 'andy' is the sequence: {"Andy Brown",{"John Godwin"},{"Simon Hayes"}}.

Automatic signatures require that a template is invoked at least once to re-compute a signature and compare the newly computed value to the signature value stored during the last successful transformation execution. Through an empirical study in our previous work [4], automatic signatures have been demonstrated to be an effective way of achieving source incremental transformations, leading to significant (30 - 50%) reduction in execution time of re-transformations.

```
1  [template public personToFile(p : Person)]
2  [file (p.name)/]
3  [p.name/]
4  [if (p.followers.isEmpty())]
5  has no followers
6  [else]
7  has some followers
8  [/if]
9  [/file]
10 [/template]
```

Listing 1.2. Example of a template-based M2T transformation, specified in OMG MOFM2T syntax.

Automatic signatures however do not always guarantee the correctness of re-generated text. For example, in Listing 1.2, the only dynamic text-emitting

section in the template emits the name of a *person*. Therefore, the signature of *person* is always *person.name*. However, this signature is not sensitive to all possible changes to *person* that could result in different text being generated, such as the followers reference becoming non-empty. Suppose that the model evolves such that person 'andy' has no followers. The signature of 'andy' will remain constant (`"Andy Brown"`) despite the change to the model, and the obvious need to re-generate the text file. The transformation engine using the automatic signature cannot detect the change, and thus, no template invocation is performed.

Templates that access properties in static sections, such as the one shown in Listing 1.2, tend to result in the computation of signature values that do no always accurately reflect a change in the model that necessitate re-generation of text.

4 User-defined Signatures

User-defined signatures give more control to the developer by allowing them to express the way in which a signature is computed. Ideally, a user-defined signature accesses precisely the same model elements (and precisely the same properties of those model elements) as the template for which the signature is a proxy. The responsibility for ensuring the signatures are representative of the templates rests with the transformation developer. Unlike automatic signatures, user-defined signatures give more control to the developer, and are more lightweight. As such user-defined signatures are heavily reliant on the developer's knowledge of the transformation.

For example, the transformation in Listing 1.2 which the automatic signature finds problematic can be addressed with the user-defined signature shown on line 2 in Listing 1.3. (Note that user-defined signatures necessitate extending the M2T language with an additional language construct). The user-defined signature is computed using the same parts of the model as the template, including whether or not the person has any followers. When the template is executed on person 'andy', the signature evaluates to `{"Andy Brown",false}`, which is a complete reflection of the property accesses made in the template. The signature expression instructs the transformation engine to evaluate the signature from the expression provided by the developer. In this case, the developer is careful to include all model element properties whose change are likely to result in re-execution of the template. The advantage of this approach is that signatures can include parts of the model that are accessed in static sections of templates, as well as those that are accessed in dynamic sections.

4.1 Drawbacks of User-defined Signatures

Despite the effectiveness of user-defined signatures at addressing the drawbacks of automatic signatures, they are not without their own drawbacks. In particular, user-defined signatures are prone to human error. For example, a transformation

```
1  [template public personToFile(p : Person)]
2  [signature : Sequence{p.name,p.followers.isEmpty()} /]
3  [file (p.name)/]
4  [p.name/]
5  [if (p.followers.isEmpty())]
6  has no followers
7  [else]
8  has some followers
9  [/if]
10 [/file]
11 [/template]
```

Listing 1.3. Example of user-defined signature in a template-based M2T transformation, specified in OMG MOFM2T syntax.

author might specify a signature expression that is incomplete. An incomplete signature expression omits at least one property access made in the template, and cannot be relied upon to produce signatures that are correct or that are true reflections of the property accesses made in a template. We address this challenges by applying runtime analysis of templates to provide helpful hints to the developer, which the developer can use to assess the correctness and completeness of their signature expressions.

Additionally, although user-defined signatures may appear to be simple, specifying signature expressions that are complete and correct reflections of model element property accesses in a template can be a onerous task, especially for complex templates involving large models. Furthermore, writing signature expressions for templates that access a large number of model element properties can result in very long lists of attributes in the signature expression, which may be unappealing to the developer and difficult to manage. Addressing this challenge remains as future work, but we anticipate providing built-in operations that make it easier for developers to declaratively express which parts of the input model should be traversed to compute a user-defined signature.

4.2 Runtime analysis for User-defined Signatures

Contemporary M2T languages limit the applicability of static analysis techniques to the languages, because most M2T languages are dynamically typed and support features such as dynamic dispatch [4]. Instead we have applied partial analysis of templates at runtime to determine model element properties accessed in a template. The property accesses made in the template then serve as useful hints to the developer for assessing the correctness and completeness of the specified signature expression composition. Property access hints are particularly useful during initial execution of a transformation, because the first transformation execution is not incremental, but the hints help the developer immediately assess the signature expression, perhaps still with little knowledge of the transformation. Therefore, on subsequent transformation executions, the developer is less concerned about the correctness of the signature expressions, provided the template has not been modified.

In addition to this, property access hints can capture model element property accesses used to control template execution flow. For instance, in the example shown in Listing 1.2, the runtime analysis will capture and suggest to the developer to include 'person.followers' in the signature expression.

5 Experimental Evaluation

To assess the performance benefits of user-defined signatures, and compare their performance with automatic signature generation and non-incremental transformation, we extend our experiment from our previous work [4]. This work used the Pongo[2] M2T transformation, which is implemented in EGL and used to generate data mapper layers for MongoDB relational databases. For this experiment, we generate Java code from the GmfGraph Ecore model obtained from the Subversion repository of the GMF team. GmfGraph Ecore model is a prime candidate for this experiment because it represents a project that has evolved independent of the signature implementation in EGL, which also means that our knowledge of GmfGraph in relation to the Pongo transformation templates is limited. User-defined signature was prototyped by extending the syntax of EGL with support for a user-defined "signature" expression per transformation rule. An example of user-defined signature expression is shown on Line 2 in Listing 1.3.

The results in Table 1 show the difference in the number of template invocations and total execution time between non-incremental transformation, and incremental transformation using automatic and user-defined signatures. Expectedly, due to the initial overhead of computing, processing, and storing signatures, the first execution of the transformation in incremental mode takes longer to execute than in non-incremental mode. However, in subsequent executions of the transformation, the incremental mode out-performs the non-incremental mode: on average execution of the incremental mode requires 66% of the time taken for non-incremental mode when using automatic signatures and 52% when using user-defined signatures. It was also observed that in two instances (versions 1.25 and 1.30 of the input model), the user-defined signatures resulted in more template invocations than the automatic signatures. This suggests that the automatic signatures, in these particular instances were insensitive to changes in the input model, and it also highlights the shortcoming (discussed in Section 3.3) of automatic signatures.

Furthermore, the results of the experiment indicate that signatures, generally, are effective means of achieving source incrementality. For instance, signatures allow the transformation engine to selectively invoke only the templates that are affected by changes to an input model. This was observed in versions 1.31 and 1.32 of the input model, when the transformation did not invoke any template using the signatures (both user-defined and automatic), whereas the non-incremental transformation invoked all the templates in the transformation.

[2] https://code.google.com/p/pongo/

		Non-Incremental		Incremental (Auto)		Incremental (User-def)	
Version	Changes (#)	Inv. (#)	Time (s)	Inv. (#)	Time (s; %)	Inv. (#)	Time (s; %)
1.23	-	72	2.16	72	2.69 (125%)	72	2.17 (100%)
1.24	1	73	1.93	1	1.38 (93%)	1	1.10 (57%)
1.25	1	73	1.89	**2**	1.47 (78%)	**4**	0.95 (50%)
1.26	1	74	2.09	1	**0.73 (35%)**	1	0.79 (38%)
1.27	10	74	1.89	44	1.27 (67%)	44	1.11 (59%)
1.28	10	74	2.16	44	1.63 (75%)	44	1.19 (55%)
1.29	14	74	2.05	14	1.67 (81%)	14	1.00 (49%)
1.30	24	77	2.21	**35**	1.45 (66%)	**37**	1.29 (58%)
1.31	1	77	2.13	0	0.97 (46%)	0	0.90 (42%)
1.32	1	77	2.13	0	0.81 (38%)	0	**0.48 (23%)**
1.33	3	79	1.88	3	0.71 (38%)	3	0.72 (38%)
			22.52		14.78 (66%)		11.70 (52%)

Table 1. Results of using non-incremental and incremental generation for the Pongo M2T transformation, applied to 11 historical versions of the GMFGraph Ecore model. (Inv. refers to invocations)

5.1 Discussion

The results from our experiments indicate that signatures are viable means of providing source incremental M2T transformations. For the Pongo transformation, we observed that the lowest execution time for the automatic signatures was as little as 35% of the execution time in non-incremental mode, and for the user-defined signatures, the execution time was as little as 23% of the execution time in non-incremental mode.

User-defined signatures often execute faster than automatic signatures. Apart from the overhead of storing, retrieving, and comparing signatures, automatic signatures also incur an additional overhead of invoking templates to calculate signatures. On the other hand, user-defined signatures are concise EOL[3] expressions that result in relatively small sized string values, compared to automatic signatures that contain all dynamic sections of a template.

6 Conclusion

We have proposed user-defined signatures, which provide source-based incremental M2T without the downsides of automatic signatures which were the subject of our previous work. We have illustrated, with the aid of an example, that user-defined signatures can effectively handle transformation templates that automatic signatures find problematic. Additionally, through empirical evaluation, we have showed that user-defined signatures are often more efficient than both non-incremental transformations, and incremental transformations that use automatic signatures.

In future work, we will make it easier for developers to specify user-defined signatures by providing built-in operations that traverse a subset of a model and collect appropriate signatures for all of the elements that have been traversed.

[3] http://www.eclipse.org/epsilon/doc/eol/

These operations will relieve the developer of the responsibility of specifying large or complex signatures. Additionally, we will extend our empirical evaluation to include several further M2T transformations that are used in industry, such as those found in the EMF[4] and GMF[5] projects.

Acknowledgements. This work was partially supported by the European Commission, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125).

References

1. Giese, H. and Wagner, R. Incremental Model Synchronization with Triple Graph Grammars. In *MoDELS*, pages 543–557. Springer, 2006.
2. K. Bennett and V. Rajlich. Software Maintenance and Evolution: a Roadmap. In *Proc. of the Conf. on the Future of Software Engineering*, pages 73–87. ACM, 2000.
3. A. Etien and C. Salinesi. Managing Requirements in a Co-evolution Context. In *RE, 2005. Proc.. 13th IEEE Int'l Conf. on*, pages 125–134. IEEE, 2005.
4. B. Ogunyomi et. al. On the Use of Signatures for Incremental Model-to-Text Transformation. In *MoDELS 2014, Valencia, Spain*, LNCS. Springer, to appear.
5. S. Sendall and W. Kozaczynski. Model transformation: The Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20(5):42–45, 2003.
6. D. Hearnden et al. Incremental model transformation for the evolution of model-driven systems. In *Proc. MoDELS*, LNCS, pages 321–335. Springer, 2006.
7. L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, 2008.
8. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[4] http://www.eclipse.org/modeling/emf/?project=emf
[5] http://wiki.eclipse.org/Graphical_Modeling_Framework/Models/GMFGen