

A BxtendDSL Solution to the TTC2023 Asymmetric and Directed Bidirectional Transformation for Container Orchestrations Case

Thomas Buchmann
thomas.buchmann@th-deg.de
Degendorf Institute of Technology
Degendorf, GER

ABSTRACT

This paper presents a solution to the Asymmetric and Directed Bidirectional Transformation for Container Orchestrations Case at TTC 2023 using BxtendDSL. BxtendDSL is hybrid language for bidirectional and incremental model transformations, allowing transformation developers to specify model transformations on the declarative and imperative level, allowing for maximum expressive power to tackle all possible transformation problems.

CCS CONCEPTS

• Software and its engineering → Domain specific languages.

KEYWORDS

container orchestration, bidirectional transformations, model merging, graphical models, YAML, BxtendDSL

ACM Reference Format:

Thomas Buchmann. 2023. A BxtendDSL Solution to the TTC2023 Asymmetric and Directed Bidirectional Transformation for Container Orchestrations Case. In *Proceedings of 15th Transformation Tool Contest (TTC'23)*. ACM, New York, NY, USA, 6 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

The transformation case addresses a real-world scenario, which is relevant e.g. for DevOps engineers. Leite et al. [11] define DevOps as a “collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions, while guaranteeing their correctness and reliability”. The still rising interest in DevOps lead to the creation of numerous domain-specific modeling notations, such as notations covering aspects of microservice architectures [13], DevOps processes [7], or multi-cloud applications [8].

DevOps relies heavily on tools that help automate deployment. Typically, those tools operate by reading a declarative description written in a structured format such as YAML¹. These structured formats conform to loosely defined schemas that often evolve from

¹<https://yaml.org/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
TTC'23, July 20, 2023, Leicester, UK

© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$0.00
<https://doi.org/XXXXXXX.XXXXXXX>

one version to another. An example is the Docker Compose file format².

The proposed case is based on a scenario inspired by Piedade et al. [12], that focuses on container orchestration using Docker Compose. To this end, an Ecore [14] model which constitutes the abstract syntax of a high-level graphical DSL for container orchestration is transformed into another Ecore model representing the abstract syntax of a Docker Compose YAML file. Changes of both models should be propagated back and forth at any time. Furthermore, changes in the high-level graphical DSL should be propagated to the YAML file, while retaining any elements that contain information that cannot be represented by the high-level DSL.

Altogether, this results in a directed, but asymmetrical transformation case. In this paper, we present our solution to the proposed transformation case using our hybrid language for bidirectional and incremental model transformations called BxtendDSL [3, 5, 6].

2 BXTENDDSL

BxtendDSL [3, 5, 6] is a state-based framework for defining and executing bidirectional incremental model transformations on demand that is based on EMF [14] and the programming language Xtend³. It builds upon Bxtend [4], a framework that follows a pragmatic approach to programming bidirectional transformations, with a special emphasis on problems encountered in the practical application of existing bidirectional transformation languages and tools.

When working with the stand-alone Bxtend framework, the transformation developer needs to specify both transformation directions separately, resulting in Bxtend transformation rules with a significant portion of repetitive code.

To this end, BxtendDSL adds a declarative layer on top of the Bxtend framework, which significantly reduces the effort required by the transformation developer. Figure 1 depicts the layered approach of our tool: First, the external DSL (BxtendDSL Declarative) is used to specify correspondences declaratively. Second, the internal DSL (BxtendDSL Imperative) is employed to add algorithmic details of the transformation that can not be expressed on the declarative layer adequately.

The handwritten code and the generated code are combined with framework code to provide for an executable transformation. The transformation developer is relieved from writing repetitive routine parts of the transformation manually using a code generator. The generated code ensures roundtrip properties for simple parts of the transformation. Since the declarative DSL usually is not expressive enough to solve the transformation problem at hand completely, the generated code must be combined with handwritten imperative

²<https://docs.docker.com/compose/compose-file/>

³<https://eclipse.dev/Xtext/xtend/>

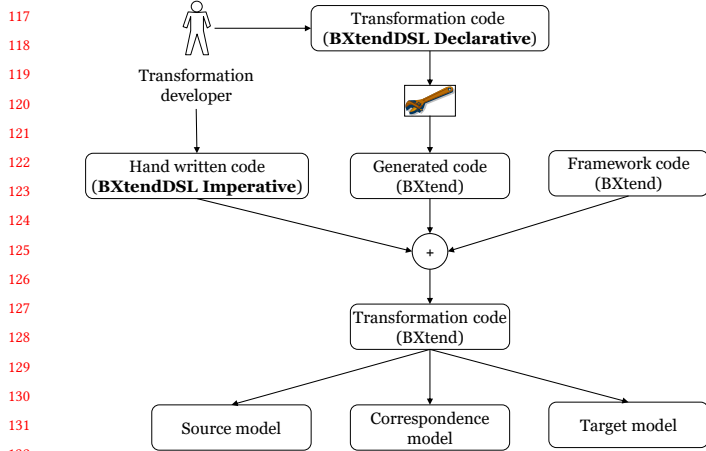


Figure 1: Layered approach used in BxtendDSL

code. Certain language constructs of the declarative DSL define the interface between the declarative and the imperative parts of the transformation. From these constructs, *hook methods* are generated, the bodies of which must be manually implemented. Hook methods are used, e.g. for implementing filters or actions to be executed in response to the deletion or creation of objects, etc.

Incremental change propagation relies on a persistently stored *correspondence model*, which allows for $m : n$ correspondences between source and target model elements. A powerful *internal DSL* may be used at the imperative level, to retrieve correspondence model elements associated with a given element from the source and target models, respectively. Please note that the transformation developer does not have to deal with managing correspondences at the declarative level, rather all the algorithmic details of managing the correspondence model are handled by our framework automatically.

3 SOLUTION

In this section, we explain the details of our BxtendDSL solution for the Asymmetric and Directed Bidirectional Transformation for Container Orchestration. We will discuss the different layers in separate subsections.

3.1 Declarative Layer

BxtendDSL code at the declarative layer is used to define transformation rules between elements of source and target models respectively. Listing 1 depicts the code for the transformation at the declarative layer.

```

155 1 sourceModel "http://york.ac.uk/ttc/containers/1.0.0"
156 2 targetModel "http://york.ac.uk/ttc/miniyaml/1.0.0"
157 3
158 4 rule Volume2MapEntry
159 5   src Volume v;
160 6   trg MapEntry me | filter;
161 7
162 8   v.name <--> me.key;
163 9
164 10 rule Image2MapEntry
165 11   src Image img;
166 12   trg MapEntry me | filter, creation;
167 13

```

```

175 14 img.image <--> me.value;
176 15
177 16 rule VolumeMount2Scalar
178 17   src VolumeMount vm;
179 18   trg Scalar sc | filter;
180 19
181 20   vm.path vm.volume --> sc.value;
182 21
183 22 rule Container2MapEntry
184 23   src Container c;
185 24   trg MapEntry me | filter;
186 25
187 26   c.name <--> me.key;
188 27   c.image c.replicas c.dependsOn {c.volumeMounts:
189     VolumeMount2Scalar} --> me.value {me.value:
190     VolumeMount2Scalar};
191 28   c.image c.replicas c.dependsOn <-- me.value;
192 29
193 30 rule Composition2Map
194 31   src Composition c;
195 32   trg Map m | filter, creation;
196 33
197 34 {c.nodes: Image2MapEntry, Container2MapEntry, Volume2MapEntry}
198   --> m.entries {m.entries: Image2MapEntry, Container2MapEntry
199     , Volume2MapEntry};
200 35 c.nodes <-- m.entries {m.entries: Image2MapEntry[img],
201     Container2MapEntry[c], Volume2MapEntry[v]};

```

Listing 1: BxtendDSL code at the declarative layer

The code at the declarative layer comprises the transformation rules for all required model elements. Each rule consists of *src* and *trg* patterns. The *trg* patterns contain modifiers, such as *filter* and *creation*, which result in the generation of hook methods. The implementation of the hook methods is described in Section 3.2. After declaring the *src* and *trg* patterns in each rule, the respective mappings are declared. Mappings may be bidirectional, depicted by the double arrow \leftrightarrow , or specified for a certain transformation direction, for example, forward (\rightarrow) or backward (\leftarrow). A very simple bidirectional mapping is depicted in Line 8 of Listing 1: the attribute name of a *Volume* is assigned to the attribute key of a *MapEntry* and vice versa.

Note that BxtendDSL was intentionally left incomplete when designed to keep the language as simple and as small as possible. To this end, we did not incorporate an expression language into BxtendDSL. Instead, we decided to apply the generation gap pattern [9] and generate hook methods that are called from the generated code at the respective steps during the transformation. Bodies for hook methods are supplied at the imperative layer using the Xtend programming language.

Consequently, a mapping that has one element on each side of the arrow may be transformed directly into executable code. Hook methods are generated for mappings with more than one element on one side of the arrow symbol. The mapping depicted in line 20 of Listing 1 is used to map the attributes *path* and *volume* to the attribute *value* of the target pattern. Because the declarative language does not comprise mechanisms to describe how the two attributes are mapped to a single attribute on the target side, a hook method is generated (cf., Listing 2).

The transformation specification on the declarative layer also comprises mappings of (containment) references between the elements of the source and target models. Lines 27 and 28 depict the respective mapping in the forward and backward directions. In the forward direction (Line 27), the cross reference *image*, attribute *replicas*, cross reference *dependsOn*, and containment reference *volumeMounts* of the class *Container* are mapped to the respective *MapEntry* in the *YAML* model. Please note that Bxtend rules are executed in

their textual order specified in the BxtendDSL declarative file. That is, rules `Volume2MapEntry`, `Image2MapEntry`, and `VolumeMount2Scalar` are executed before rule `Container2MapEntry`. Thus, we can be sure that all the elements required for this mapping actually exist and can be retrieved. The syntax of the mapping specified in line 27 contains curly brackets. This indicates that for this feature, the correspondence/trace model is accessed to obtain the respective model elements from the source and target models. The specification of this mapping also results in the generation of a hook method, which is used to describe all the algorithmic details to realize this mapping on the imperative layer.

Rule `Composition2Map` (c.f., Line 30-35 in Listing 1) maps the root elements of both models. These elements (transitively) contain all other model elements via references `nodes` and `entries`. To realize the transformation, this rule is executed after all other rules are executed to ensure that the respective model elements actually exist when they are assigned to the containment references of the source and target root elements. In the forward direction, `Images`, `Containers`, and `Volumes` are assigned to the respective entries of the target `Map`. Again, please note that curly brackets are used to access the correspondence model and retrieve the already existing target elements for `Images`, `Containers`, and `Volumes`. A similar mapping is required for the backward transformation (see Line 35 in Listing 1).

3.2 Imperative Layer

On the imperative layer, the bodies for hook methods must be supplied. This holds for the specification of modifiers (e.g., `filter` or `creation`), as well as for mappings where further information is required, which cannot be supplied using the declarative language.

Listing 2: Hook method for mapping attributes `path` and `volume` to `Scalar.value`

```
1 override protected valueFrom(String path, Volume volume) {
2     return new Type4value(volume.name + ":" + path)
3 }
```

Listing 2 shows the imperative code that is required to realize the mapping `vm.path vm.volume -> sc.value`, as depicted in Line 20 of Listing 1. The value attribute of the scalar is a concatenation of the name of the `Volume` and the `path`, separated by `":"`.

Please note that this rule does not specify the backward direction; rather, it is addressed in the imperative code for mapping `c.nodes <- m.entries ...` from the rule `Composition2Map` in Listing 1.

Listing 3 depicts the code required on the imperative layer to realize the rule `Image2MapEntry`, as specified in Lines 10-14 in Listing 1.

Listing 3: Imperative code for rule `Image2MapEntry`

```
1 override protected filterMe(MapEntry me) {
2     me.key == "image"
3 }
4
5 override protected onMeCreation(MapEntry me) {
6     me.key = "image"
7 }
8
9 override protected valueFrom(String image) {
10    new Type4value(MiniyamlFactory.eINSTANCE.createScalar() =>
11        [value = image])
12 }
13
14 override protected imageFrom(Value value) {
```

```
15     return new Type4image(((value as Scalar).value))
16 }
```

The implementation of the modifiers `filter` and `creation` is shown in Lines 1-7 of Listing 3. The modifiers result in the generation of methods `filterMe` and `onMeCreation`, and the transformation developer only needs to supply a body to realize the desired behavior. In this case, an `Image` from the container model is transformed into a `MapEntry` of the `YAML` model. The corresponding `MapEntry` has a key attribute with the value `"image"`. The filter is applied when transforming in backward direction and it is used to filter all `MapEntry` elements from the target model and only retrieve the ones whose key attribute contains the value `"image"`.

Methods `valueFrom` and `imageFrom` are hook methods that are generated from the mapping depicted in Line 14 of Listing 1. Note that in this case, the mapping only contains a single element on each side of the arrow, but the respective attributes are of different types. Thus, hook methods are required to specify how these types are mapped onto each other. In the forward direction (method `valueFrom`), the `"image"` String is transformed into a `Scalar`, where the attribute value is assigned to the `value` attribute of the scalar. The value of this attribute is returned in the backward direction.

Listing 4: Imperative code for mapping specified in Line 27 of Listing 1

```
1 override protected valueFrom(Image image, int replicas,
2     List<Container> dependsOn, List<Scalar> volSc,
3     Value oldValue) {
4     var entry = yamlFactory.createMap()
5     if (replicas > 1) {
6         val me = yamlFactory.createMapEntry() =>
7             [key = "replicas"
8              value = yamlFactory.createScalar() =>
9                  [value = "" + replicas]
10            ]
11     entry.entries += me
12 }
13 if (image != null)
14     entry.entries += (elementsToCorr.get(image).getTarget()
15         .get(0) as SingleElem).element as MapEntry
16 if (!dependsOn.isEmpty) {
17     val me = createMapEntry("depends_on")
18     val list = me.value as miniyaml.List
19     for (Container c : dependsOn)
20         list.values += yamlFactory.createScalar() =>
21             [s | s.value = c.name]
22     entry.entries += me
23 }
24 if (!volSc.empty) {
25     val me = createMapEntry("volumes")
26     val list = me.value as miniyaml.List
27     for (Scalar s : volSc) {
28         list.values += s
29     }
30     entry.entries += me
31 }
32 ...
33 new Type4value(entry)
```

Listing 4 depicts the imperative code that is required to realize the mapping of `image`, `replicas`, `dependsOn` and `volumeMounts` attributes and references of a `Container` to respective `MapEntry`s in forward direction. The imperative code contains several conditional blocks that need to be processed if certain conditions hold, for example, if the value of the integer attribute `replicas` is `> 1`. In this case, a new `MapEntry` with appropriate key-value pairs is created and added to the parent `Map`. If additional `MapEntry`s are required for

dependencies and volumes, they are also created using this hook method. Respective entries are then also added to the map, which is then returned at the end of the hook method.

4 EVALUATION

The aim of the proposed transformation case is to answer research questions concerning conciseness (i.e., how much specification effort is required to solve this case with current bx tools), preservation of information that cannot be mapped on the other model, and scalability (i.e., how well the proposed solution scales with increasing model sizes).

To this end, the transformation is classified according to the evaluation criteria discussed in the following subsections.

4.1 Correctness

The correctness of the transformation may be verified using two different and supplied comparators. The *MiniYAMLComparator* ignores the order of elements in the respective models, whereas the *MiniYAMLExactComparator* also considers the ordering of elements.

MiniYAMLComparator. The BXtendDSL solution passes all supplied tests (forward, backward, incremental forward) for the benchmark testsuite using the *MiniYAMLComparator*; for example, we achieved 100% correctness in this case.

MiniYAMLExactComparator. For the *MiniYAMLExactComparator*, the BXtendDSL solution passes eight out of nine batch forward tests, seven of eight batch backward tests, and four of five incremental forward tests, which leads to a correctness rate of 86.4%. The following test cases fail, due to incorrect ordering of elements in a multi-valued reference: `completeModel` in both `BatchForward` and `BatchBackward`, as well as `updateReplicas` in `IncrementalForward`

4.2 Conciseness

To measure the conciseness of the transformation specification, nodes in the respective AST/ASG of the languages used should be counted. To this end, solution developers are required to provide specific implementations of the AST-counter. In our case, an AST-counter for the BXtendDSL language and an AST-counter for the Xtend programming language are required. However, an additional problem remains. First, BXtend does not provide static libraries; rather, framework code is generated specifically for each transformation. That is, the project contains a significant portion of the generated Xtend and Java code, which must be excluded when the AST nodes are computed. Furthermore, large parts of the Xtend code that are used to implement hook methods on the imperative layer are generated as well. Consequently, the transformation developer is required to specify only the bodies of the respective languages. However, an AST-counter works on a valid source code, that is, an Xtend class that has no compile errors and counts all nodes present in the class. This would lead to incorrect results because large parts of the generated code would be considered. From our understanding, however, conciseness should only take the parts of the code into account that the transformation developer must supply to make the transformation work. To this end, we decided to provide conciseness information using the LOC metrics defined

in [1] and [2]. We further split up the numbers into code required on the declarative and the imperative layer respectively. Table 1 presents the results.

	BXtendDSL Declarative	BXtendDSL Imperative
Lines of code	32	202
Number of words	94	686
Number of characters	862	5967

Table 1: Size of the transformation definitions of both solutions

The results clearly indicate that a significant portion of the transformation was specified on the imperative layer. This is due to the asymmetric nature of the transformation case, which cannot be handled adequately in the declarative layer of BXtendDSL. However, the resulting transformation specification is still concise if we compare it to similar transformation cases such as AST2Dag [10].

4.3 Least Change

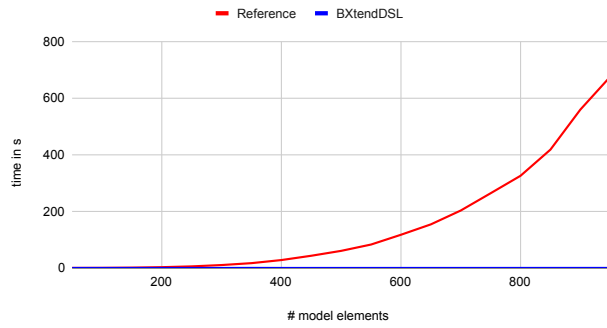
Beyond correctness, the transformation should preserve additional information in the YAML file that cannot be expressed in the Containers model. When executing the transformation in the two modes (*MiniYAMLComparator*, which checks the general correctness, and *MiniYAMLExactComparator*, which also considers the order of elements in the YAML model), we observe that the transformation can preserve the additional information specified in the YAML file, but not in the exact order of elements. Test cases specified in class `IncrementalForward` were considered for this test. While the BXtendDSL solution achieves 100% accuracy for Mode 2, it passes four out of five tests for Mode 1, resulting in an accuracy of 80% when the exact order of the elements matters.

4.4 Scalability

To measure how well the transformation scales to models with increasing numbers of containers, volumes, and images, different scalability tests have been executed in the forward and backward directions, as well as both in batch and incremental situations. We attempted to compare the results of BXtendDSL to the provided reference implementation, but unfortunately, the backward measurements failed with errors on our computers. To this end, only the plots for batch forward and incremental forward transformations contain data for the reference implementation as well. All scalability tests were performed on the same computer in isolation to avoid side effects. A desktop PC with an AMD Ryzen 7 3700x CPU was used, running at a standard clock of 3.60 GHz, with 32 GB of DDR4 RAM and Microsoft Windows 11 64-bit as the operating system. We used Java 13.0.2, Eclipse 4.27.0, and EMF version 2.33.0, to compile and execute Java code for the scalability test suite. Each test was repeated five times, and the median measured time was computed.

For each test, we used the provided class for scalability measurements, which created models of increasing sizes up to 1000 elements. BXtendDSL proves to scale very well with increasing model sizes, as depicted by the plots for batch forward (c.f., Figure 2), incremental forward (c.f., Figure 3), batch backward (c.f., Figure

Batch Forward: Reference Implementation and BxtendDSL



Batch Forward: Reference Implementation and BxtendDSL

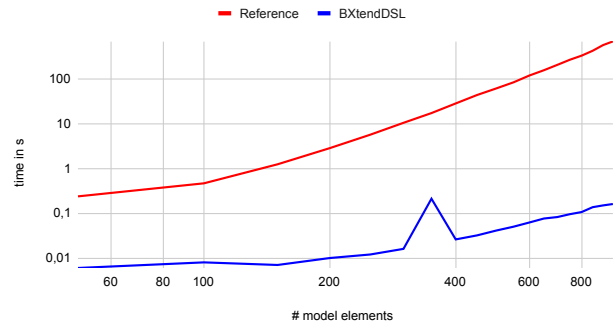
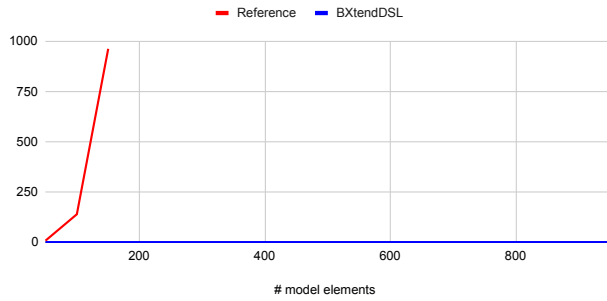


Figure 2: Forward batch transformation: Linear/linear scale (left) and log/log scale (right)

Incremental Forward: Reference Implementation and BxtendDSL



Incremental Forward: Reference Implementation and BxtendDSL

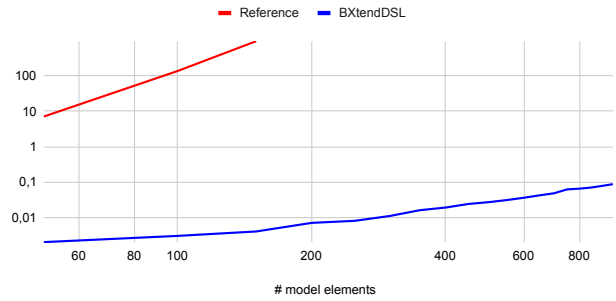
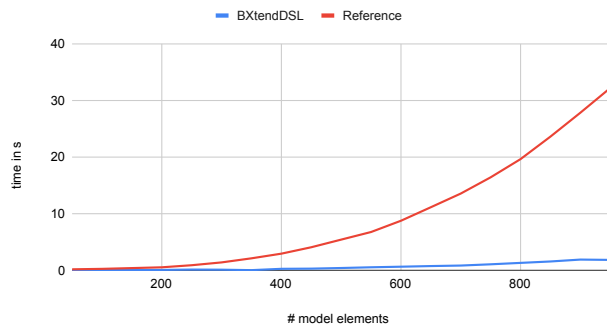


Figure 3: Forward incremental transformation: Linear/linear scale (left) and log/log (right)

Batch Backward: Reference Implementation and BxtendDSL



Batch Backward: Reference Implementation and BxtendDSL

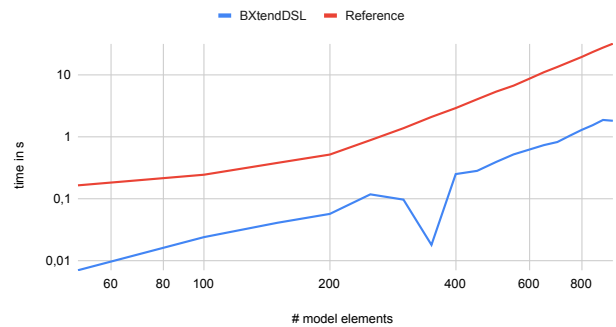
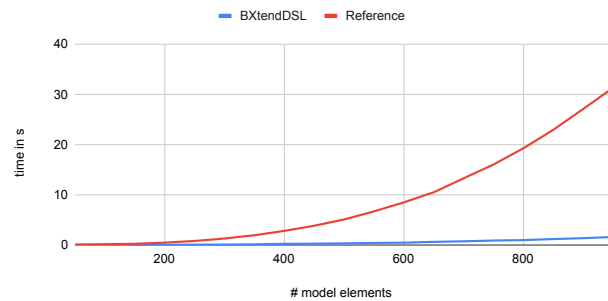


Figure 4: Backward batch transformation: Linear/linear scale (left) and log/log (right)

4), and incremental backward (c.f., Figure 5). Please note that two plots are given per figure: one with linear scaling of the x and y axes and the other with logarithmic scaling. The linear plot is meant to provide a realistic impression for the actual complexity curve of the BxtendDSL solution compared to the reference implementation.

The logarithmic plots help zoom into finer details for smaller models (practically invisible in the linear plot), and zoom out for larger models so even large differences in runtime can still be presented qualitatively.

Incremental Backward: Reference Implementation and BxtendDSL



Incremental Backward: Reference Implementation and BxtendDSL

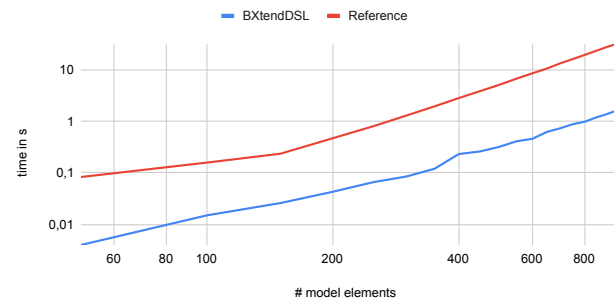


Figure 5: Backward incremental transformation: Linear/linear scale (left) and log/log (right)

5 CONCLUSION

The BxtendDSL solution provided for the asymmetric and directed bidirectional transformation for the container orchestration case has proven to be concise, sufficiently correct, and scalable. Because BxtendDSL allows us to specify details of the transformation on both declarative and imperative levels, the transformation developer may choose (almost) freely which programming paradigm is best suited for the transformation problem at hand. A combination of both results in high expressive power while simultaneously maintaining low specification effort at the same time. The transformation case revealed small bugs in the code generation engine, which was used to generate executable code from declarative specifications. Thus, minor tweaks of the generated code are required. These issues have already been addressed, and will be incorporated into BxtendDSL in the upcoming release.

REFERENCES

- [1] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. 2017. The Families to Persons Case. In *Proceedings of the 10th Transformation Tool Contest*, Vol. 2026. CEUR-WS.org, Marburg, Germany, 27–34. <http://ceur-ws.org/Vol-2026/paper2.pdf>
- [2] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling* 19, 3 (May 2020), 647–691. <https://doi.org/10.1007/s10270-019-00752-x>
- [3] Matthias Bank, Thomas Buchmann, and Bernhard Westfechtel. 2021. Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021*, Slimane Hammoudi, Luis Ferreira Pires, Edwin Seidewitz, and Richard Soley (Eds.). SCITEPRESS, 15–27. <https://doi.org/10.5220/0010188200150027>
- [4] Thomas Buchmann. 2018. Bxtend - A Framework for (Bidirectional) Incremental Model Transformations. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*. 336–345. <https://doi.org/10.5220/0006563503360345>
- [5] Thomas Buchmann, Matthias Bank, and Bernhard Westfechtel. 2022. BxtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language. *J. Syst. Softw.* 189 (2022), 111288. <https://doi.org/10.1016/j.jss.2022.111288>
- [6] Thomas Buchmann, Matthias Bank, and Bernhard Westfechtel. 2023. BxtendDSL at Work: Combining Declarative and Imperative Programming of Bidirectional Model Transformations. *SN Comput. Sci.* 4, 1 (2023), 50. <https://doi.org/10.1007/s42979-022-01448-8>
- [7] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. 2020. DevOpsML: towards modeling DevOps processes and platforms. In *Proceedings of the 23rd*

ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. ACM, Virtual Event Canada. <https://doi.org/10.1145/3417990.3420203>

- [8] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. 2018. CloudMF: Model-Driven Management of Multi-Cloud Applications. *ACM Transactions on Internet Technology* 18, 2 (May 2018). <https://doi.org/10.1145/3125621>
- [9] Martin Fowler. 2011. *Domain-Specific Languages*. Addison-Wesley. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html
- [10] Oliver Hacker and Thomas Buchmann. 2023. NICE: A Flexible Expression Language. In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2023, Lisbon, Portugal, February 19-21, 2023*, Francisco José Dominguez Mayo, Luis Ferreira Pires, and Edwin Seidewitz (Eds.). SCITEPRESS, 63–74. <https://doi.org/10.5220/0011712700003402>
- [11] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojevic, and Paulo Meirelles. 2020. A Survey of DevOps Concepts and Challenges. *Comput. Surveys* 52, 6 (Nov. 2020). <https://doi.org/10.1145/3359981>
- [12] Bruno Piedade, João Pedro Dias, and Filipe F. Correia. 2022. Visual notations in container orchestrations: an empirical study with Docker Compose. *Software and Systems Modeling* 21, 5 (Oct. 2022), 1983–2005. <https://doi.org/10.1007/s10270-022-01027-8>
- [13] Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2021. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. *SN Computer Science* 2, 6 (Sept. 2021). <https://doi.org/10.1007/s42979-021-00825-z>
- [14] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF Eclipse Modeling Framework* (2nd ed.). Addison-Wesley, Boston, MA.