

Reducing the Verbosity of Imperative Model Refinements by Using General-Purpose Language Facilities

Christopher Gerking¹(✉), David Schubert², and Ingo Budde²

¹ Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
christopher.gerking@upb.de

² Fraunhofer IEM, Paderborn, Germany

Abstract. Refinements are model transformations that leave large parts of the source models unchanged. Therefore, if refinements are executed outplace, model elements need to be copied to the target model. Refinements written in imperative languages are increasingly verbose, unless suitable language facilities exist for creating these copies implicitly. Thus, for languages restricted to general-purpose facilities, the verbosity of refinements is still an open problem. Existing approaches towards reducing this verbosity suffer from the complexity of developing a higher-order transformation to synthesize the copying code. In this paper, we propose a generic transformation library for creating implicit copies, reducing the verbosity without a higher-order transformation. We identify the underlying general-purpose language facilities, and compare state-of-the-art languages against these requirements. We give a proof of concept using the imperative QVTo language, and showcase the ability of our library to reduce the verbosity of an industrial-scale transformation chain.

Keywords: Model refinement · Implicit copy · Imperative languages

1 Introduction

In model-driven engineering (MDE), imperative transformation languages are widely used as vehicles for creating operational model transformations [4]. By including facilities known from general-purpose languages, imperative languages enable [26] the specification and execution of fit-for-purpose model transformations.

One major purpose of model transformation is *refinement* [25], with use cases such as desugaring [15], aspect weaving [17], or synthesis [7]. According to the terminology used in [20], a model refinement “preserves large parts of the source model and adds additional information”. Therefore, a refinement is *endogenous* because source and target models are based on the same metamodel. Furthermore, refinements in MDE often need to preserve the source models as primary, immutable artifacts. Thus, one possible approach is to create a temporary one-to-one copy of the source model (e.g., a file copy) in order to refine the copy using

an *inplace* transformation. However, this approach lacks end-to-end traceability and leads to redundant copies of the changing elements. Therefore, refinement transformations are often required to execute *outplace* [26] and need to copy the unchanged elements during the refinement.

This paper addresses the problem that imperative languages are increasingly verbose when used for outplace model refinements. The reason is that users need to copy the unchanged elements explicitly by iterating over all links of each element, resulting in a huge amount of *boilerplate* code that users must write and maintain. Nowadays, maintainability of model transformations is already considered as a critical factor [1]. A well-known approach for reducing this verbosity is the *implicit copy* design pattern [24], proposing a mechanism for creating copies implicitly. However, such a mechanism must be supported by the transformation language in use. Therefore, Lano and Kolaoudouz-Rahimi request “suitable language facilities to be present” [24]. On the one hand, languages such as ATL [18] or Epsilon Flock [29] provide special-purpose facilities for implicit copies. On the other hand, languages restricted to general-purpose facilities still suffer from an increased verbosity because users need to create all copies explicitly.

In literature, other approaches exist towards reducing this verbosity [12, 13, 22]. These related works regard the copying as *schematic-repetitive code* [30], and use a *higher-order transformation* (HOT, [32]) to synthesize the boilerplate code. However, such approaches suffer from the intrinsic complexity of developing a HOT [31], and from the maintenance burden of re-executing the HOT to update the boilerplate code in case of metamodel evolution [28].

In this paper, we demonstrate that general-purpose facilities are sufficient to realize the implicit copy pattern in imperative languages. Thereby, the language facilities requested by Lano and Kolaoudouz-Rahimi [24] are made explicit. We regard the copying as *generic code* [30] and propose a transformation library that provides implicit copies to reduce the verbosity of model refinements. We identify an amount of four underlying language facilities that enable our approach, and check state-of-the-art imperative languages against these facilities. Thereby, we assess their ability to realize our library. In contrast to related work, we propose a generic library that is reusable as-is for different metamodels, saving users from the effort of developing a HOT and maintaining the synthesized boilerplate code.

As a proof of concept, we realize our library using the QVT Operational Mappings language (QVTo, [27]), and showcase the ability of our library to improve the maintainability of an industrial-scale transformation chain.

In summary, this paper makes the following contributions:

- We provide a concept for an implicit copy library that reduces the verbosity of model refinements written in imperative transformation languages.
- We illustrate the required language facilities, and compare state-of-the-art imperative languages against these requirements.
- We give a proof of concept using the imperative QVTo language.

Paper Organization: Sect. 2 introduces a motivating example, before we discuss related work in Sect. 3. In Sect. 4, we present our implicit copy library and

compare existing languages against the required facilities. We use QVTo to provide a proof of concept in Sect. 5, before concluding in Sect. 6.

2 Motivating Example

As an illustrative example of a model refinement, we consider a statemachine dialect that requires *desugaring*, i.e., semantics-preserving normalization to a more basic syntactic form. In our dialect, every state of a statemachine may refer to a *do* activity which is a behavior executed periodically. The activity is released for execution in a fixed time interval (the period) and executes once per period. In Fig. 1a, we show an excerpt from a statemachine including a state A with a do activity named a() and a fixed period of 50 milliseconds.

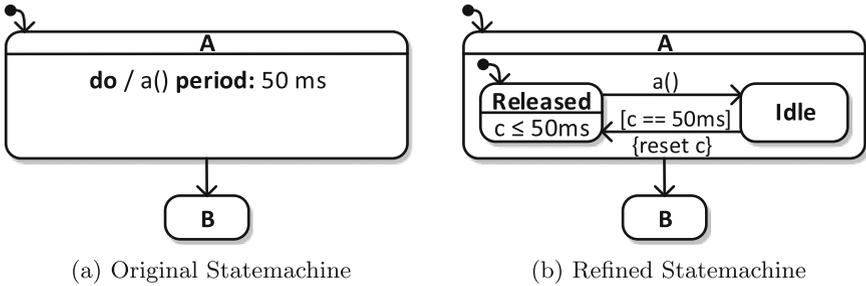


Fig. 1. Example refinement of statemachine models

When transforming such models to an execution platform, the declarative specification of a do activity needs to be refined to an equivalent operational form. For example, in Fig. 1b, we replaced the declaration of the do event by a submachine with two substates and one clock c , measuring the time that has already passed during the current period. Initially, the submachine is in state *Released*, denoting a situation in which the behavior $a()$ was released for execution, but has not yet been executed in the current period. At latest after 50 ms, the invariant $c \leq 50\text{ms}$ forces the submachine to execute the behavior $a()$ by switching to state *Idle*. In this state, the submachine waits until the clock constraint $c == 50\text{ms}$ is fulfilled, i.e., the end of the current period. At this time, the activity is released for execution again. Therefore, the submachine switches back to *Released* and resets the clock c to zero.

Our example refinement affects only states with do activities, whereas all other model elements (e.g., transitions) remain unchanged. Copying the unchanged model elements explicitly requires boilerplate code to iterate over all elements and copy all their links to other elements. In language families like UML with a multitude of different types included, the number of unchanged elements might increase drastically, and lead to an advanced verbosity of the boilerplate code for refinements. Thus, according to the *implicit copy* pattern,

the transformation language in use should provide a mechanism to specify explicitly only the refinement of states. All other unchanged model elements should be copied implicitly.

3 Related Work

In the following, we first review existing transformation approaches with respect to built-in refinement support (cf. Sect. 3.1). Then, we discuss approaches using a HOT for refinements in languages without built-in support (cf. Sect. 3.2).

3.1 Refinement as Built-in Language Facility

In Table 1, we check existing transformation approaches against the core requirements for model refinements described in Sect. 1. First, *outplace* transformations must be supported, as opposed to *inplace* transformations on a one-to-one copy of the source model. Second, languages need an *implicit copy* facility to reduce the verbosity of refinements. If such a facility is available, we also check whether end-to-end *traceability* between the source elements and their copies is provided, and whether it is possible to execute a *refinement* during the copy process, instead of creating a one-to-one copy.

The field of graph transformation (GT) provides approaches that are tailored to *inplace* transformations. However, languages such as Henshin [3] support *outplace* transformations as well, connecting models by means of a dedicated trace model that provides traceability between source and target elements. Whereas no implicit copy facility is given in general, Krause et al. [23] propose dynamically typed graph transformations that enable the specification of concise and generic rules for implicit copies¹.

In contrast, approaches based on triple graph grammars (TGG) address *outplace* transformations, connecting source and target models by means of a correspondence graph. The correspondence graph ensures traceability between source and target elements. Anjorin et al. [2] address refinement support for TGG by enabling specialization of basic transformation rules. However, in general, no mechanism for creating implicit copies is available.

The Epsilon Wizard Language (EWL, [21]) is a task-specific language for manipulating user-defined selections of model elements. EWL is restricted to *inplace* transformations. Therefore, no *outplace* transformations and no implicit copies are supported.

The task of refining a model can also be viewed as a special case of model migration to compensate metamodel evolution. In this field of model/metamodel co-evolution, a large body of knowledge is already existing and surveyed in [14, 16, 28]. In fact, the majority of approaches uses *outplace* transformations [16]. For example, Epsilon Flock [29] creates implicit copies using a conservative copy strategy. The approach provides traceability and allows to execute refinements as user-defined migrations during the copy process.

¹ <http://www.ckrause.org/2013/04/copying-emf-models-with-henshin.html>

Similar to Flock, ATL provides a *refining* execution mode that implicitly copies all the unchanged model elements from the source model to the target model. The refining mode provides traceability and also allows to execute refinements during the copy process because user-defined transformation rules override the default copy behavior.

The QVT standard provides languages with support for outplace transformations. Whereas QVTr supports no implicit copies, QVTo provides a copy mechanism in terms of its *deepclone* operation [27]. However, this operation does not provide traceability between the source elements and their copies, and does not allow to execute refinements during the copy process.

Table 1. Comparative overview on refinement support

	GT	TGG	EWL	Flock	ATL	QVTr	QVTo
Outplace	(✓) ^a	✓	✗	✓	✓	✓	✓
Implicit Copy	(✓) ^b	✗	✗	✓	✓	✗	✓
-Traceability	(✓) ^c	-	-	✓	✓	-	✗
-Refinement	✗	-	-	✓	✓	-	✗

^aOutplace is supported by graph transformation languages such as Henshin.

^bLanguages such as Henshin provide a trace model.

^cImplicit copies are supported by dynamically typed graph transformations [23]

3.2 Refinement Using Higher-Order Transformations

The work by van Gorp et al. [13] extends graph transformations with declarative copy annotations for elements to be copied. A HOT is used to transform these implicit annotations into executable graph transformations with an explicit copy behavior. The use of annotations reduces the verbosity of the transformations, and also allows to specify refinements to be executed during the copy process.

Unlike the above approach that uses graph transformations as inputs for the HOT, Goldschmidt et al. [12, 19, 20] generate a full set of copy rules from a specific metamodel, one rule for each type. The copy rules generated by the HOT are based on the declarative QVTr language. In order to execute refinements during the copy process, it is possible to override specific copy rules with custom refinement rules. Furthermore, the authors also discuss the possibility of using another HOT to weave the refinement rules into the copy rules, receiving a single comprehensive refinement transformation.

Whereas the above approaches focus on declarative languages, the work by Kraas [22] addresses an imperative transformation language in terms of QVTo, similar to the scope of this paper. The author uses a HOT to generate an imperative copy transformation for a specific metamodel. Similar to the declarative

case, one copy operation for each type inside the metamodel is generated. These default copy operations can be overridden by user-defined refinement operations.

A common drawback of the above approaches is that developing a HOT is generally considered a tedious task [31]. Furthermore, if the HOT can not be re-executed automatically, frequent metamodel evolution can easily evolve into serious maintenance efforts to update the generated code manually. Nevertheless, in this paper, we consider the HOT-based approach described by Kraas [22] as our baseline due to its focus on imperative transformations.

4 A Generic Library for Implicit Copies

In this section, we present our approach towards reducing the verbosity of imperative model refinements. In Sect. 4.1, we describe our conceptual approach based on a generic transformation library that provides implicit copies. In Sect. 4.2, we extrapolate the required language facilities, and compare state-of-the-art imperative languages against these requirements.

4.1 Conceptual Approach

Our approach enables model refinements based on general-purpose facilities of imperative transformation languages. Instead of using a HOT, we provide a generic library implementation that encapsulates the repetitive copying behavior, and is reusable by arbitrary user-defined refinement transformations.

As an example, Fig. 2a shows a metamodel excerpt for statemachines. We assume a type `Element` to be the implicit supertype of all model elements, similar to the `Object` class in Java. The only type affected by the normalization introduced in Sect. 2 is `State` because additional substates and transitions are added to replace the declarative `do Activity`. The elements of type `Transition` and any other types omitted in Fig. 2a should be copied without changes.

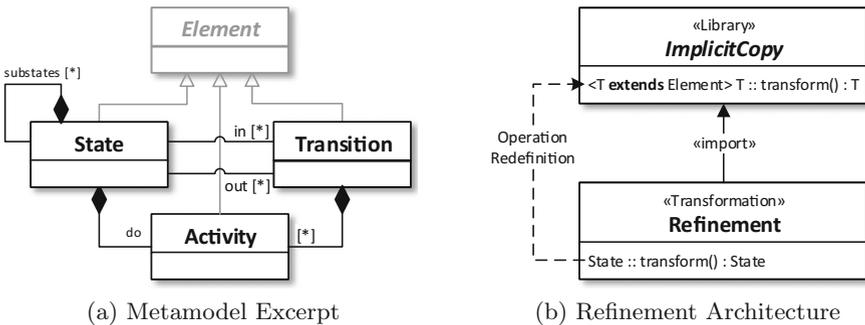


Fig. 2. Application scenario for model refinement

Figure 2b illustrates our approach at the architectural level. We introduce a transformation library named `ImplicitCopy` that can be imported by arbitrary `Refinement` transformations. Our library consists of a single operation named `transform` that receives an arbitrary model element. In its basic form, the operation returns a one-to-one copy of the element, including copies of all cross-linked elements. To make our operation applicable to arbitrary elements, the context type `T` (representing the received element) is a generic subtype of `Element`. Since `transform` returns a copy, `T` also acts as the return type. In Fig. 2b, we use a double colon to separate the operation’s context type from the operation name, whereas a single colon separates the operation name from its return type.

In our example, the `Refinement` transformation includes a redefinition of the generic `transform` operation for elements of type `State`. By returning a subtype of `Element`, the operation makes use of covariant return types. On the right-hand side of Fig. 3, we depict the pseudocode implementation. To avoid duplicate copies, our library holds a cache of copied elements. The cache is a map between the originals and their copies, providing the default `get` and `put` operations to obtain or add a copy. Using the cache, the `transform` operation first checks for an existing copy and, if present, returns it. Otherwise, it creates a new instance of the same type as the received element, which is denoted by `self`. The new instance is assigned to the `result` variable, and added to the `cache` immediately.

Subsequently, the operation reflectively traverses all `features` of the `self` element, representing links to other elements. For each `feature`, the operation obtains the current `values` (representing the linked elements), and invokes `transform` recursively on each of them. At this point, we apply the *recursive descent* design pattern for model transformations [24]. In order to avoid duplicate copies in a recursive invocation, it is of crucial importance that every `result` is added to the cache immediately after its creation. Thereby, we also ensure termination

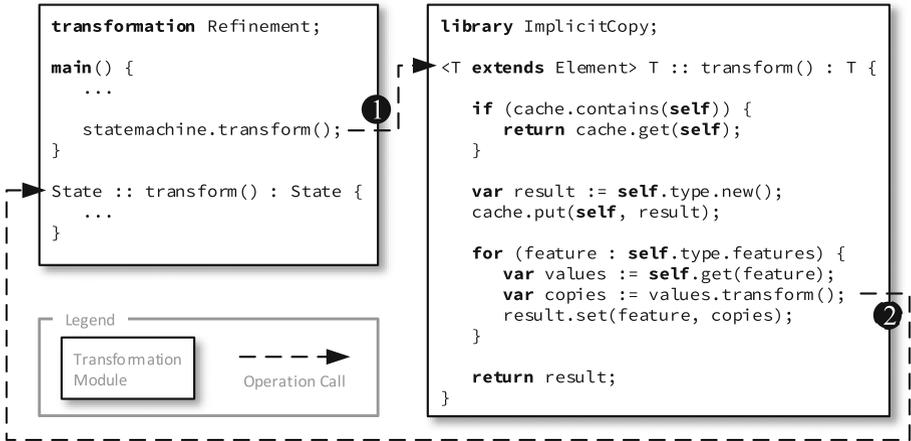


Fig. 3. Interaction of refinement transformation and implicit copy library

because the recursion takes place only during the first execution of the operation for a particular element. Finally, the resulting copies are assigned to the respective feature of the result, which is returned at the end of the operation.

On the left-hand side of Fig. 3, we depict a transformation using our library to refine statemachine models. From its main operation, the refinement invokes `transform` on a statemachine, as depicted by ❶. Accordingly, the library implementation of the operation executes. However, when invoking the operation recursively on a `State` contained in the statemachine, a virtual operation call depicted by ❷ ensures that the redefining implementation executes instead. Thereby, we ensure that refinements can be executed during the copy process.

4.2 Language Facilities

The concept described in Sect. 4.1 requires four core facilities of imperative languages in order to apply our approach successfully. In the following, we extrapolate these facilities (F1-F4) from our conceptual approach.

- (F1) **Module Superimposition** [33] is the ability to layer separate transformation modules by means of importing. Thereby, it is possible for arbitrary transformations to import and reuse our implicit copy library.
- (F2) **Dynamic Dispatch** of virtual operations is required to distinguish at runtime between the generic copying behavior implemented by our library, and the user-defined refinement behavior. The default copying operation needs to be polymorphic in order to support redefinitions for specific model

Table 2. Comparative overview on language facilities

	Xtend	K3	ATL	ETL	QVTo
Built-In Facility	✗	✗	✓	✗ ^a	✗ ^b
Module Superimposition (F1)	✓	✓	✓	✓	✓
Dynamic Dispatch (F2)	✓	✗ ^c	✓	✓	✓
Reflection (F3)	✓	✓	✗ ^d	✓	✓
Generic Return Types (F4)	(✓) ^e	✗ ^f	✗	✗	✗
Successful Implementation	✓	✗	✗	✓	✓

^aThe Epsilon family provides refinement facilities in terms of Epsilon Flock.

^bQVTo provides a *deepclone* operation that is restricted to one-to-one copies.

^cKermeta 3 does not support dynamic dispatching for methods of aspect classes.

^dATL provides no access to the reflective API of the EMF modeling framework.

^eXtend supports generic return types, but not in combination with dynamic dispatch.

^fKermeta 3 does not support generic return types for methods of aspect classes.

element types. At runtime, the executing implementation depends on the actual type of the element that the operation has been invoked on.

- (F3) **Reflection** of the underlying modeling framework. Given an element of a concrete type, the copy operation needs to create a new instance of the same type, which is only known at runtime. Furthermore, reflective access to all features of an element’s type is required.
- (F4) **Generic Return Types (optional)** to infer the type of a copied element statically. When invoking the copy operation on an element of a certain type, it is beneficial to obtain the identical return type at compile time. Otherwise, type casting is required to establish type compatibility.

In order to point out the generality of our approach, we attempted to implement our library in five state-of-the-art imperative transformation languages, namely Xtend, Kermeta 3 (K3), ATL, ETL, and QVTo, covering two of the most widely used de-facto standards [4]. In Table 2, we check these languages against the facilities F1-F4. We also indicate if an built-in facility is already provided, and if our concept could be implemented successfully.

5 Proof of Concept

We realized our concept of a generic implicit copy library using QVTo [27] as one of the languages that lacks a built-in facility, and provides the mandatory facilities F1-F3 identified in Sect. 4.2. QVTo enables the specification and execution of imperative model-to-model transformations, consisting of modules that can define operations and import other modules. In QVTo, mapping operations are used to transform a source element into a target element.

5.1 Library Implementation Using QVTo

We provide a reference implementation of our concept as a transformation library for Eclipse QVTo, which is based on the Eclipse Modeling Framework (EMF). Our library is ready to use and publicly available at an Eclipse update site². In QVTo, we use a mapping to implement the default `transform` operation. Since QVTo does not support generic return types (F4), we use the implicit supertype `Element` as both context and return type of our mapping. Below, we show the mapping declaration and its implementation in QVTo.

² <http://muml.org/implicitcopy/updates>

```

mapping Element ::= transform() : Element {
  init {
    var type := self.oclAsType(EObject).eClass();
    var factory := type.ePackage.eFactoryInstance;
    result := factory.create(type);
  }

  type.eAllReferences->select(modifiable())->forEach(r) {
    var values := self.oclAsType(EObject).eGet(r);
    var copies := values.oclAsType(Element).map transform
      ();
    result.oclAsType(EObject).eSet(r, copies);
  }
}

```

For the received element denoted by `self`, our mapping creates a new instance of the same type inside the `init` section, which allows to assign the instance to the predefined `result` variable. By using EMF's reflective API, we obtain the concrete `type` of the `self` element, and use the corresponding factory to create a new instance. QVTo does not require the manual implementation of the caching mechanism introduced in Sect. 4.1. Instead, a built-in traceability mechanism [27] records a trace link between the `self` and the `result` elements at the end of the `init` section. If the mapping is invoked on an element for which a trace record exists already, the copy will be obtained from the trace automatically without executing the mapping again.

Subsequently, we link the `result` to the copies of all cross-linked elements. For each modifiable reference `r` of the `type`, we obtain the collection of cross-linked `values` using EMF's `eGet` operation. We create the copies by casting each of the `values` to the required type `Element` and invoking `transform` recursively. Finally, we use a EMF's `eSet` operation to create the links between the `result` and the copies of the cross-linked elements.

5.2 Refinement Implementation Using QVTo

Transformations may use the identifier `org.muml.ImplicitCopy` to import our library. As an example, we illustrate how we implemented the normalization of `do` activities for statemachine models introduced in Sect. 2. Below, we show a QVTo transformation named `DoActivity` operating on statemachine models of type `SM`. In the main operation, the transformation starts the refinement by invoking the imported `transform` mapping on all root elements of type `StateMachine`.

For elements of type `State`, an additional mapping declared in the refinement transformation is automatically executed instead of the default copying operation. Inside this mapping, the state name as well as all incoming/outgoing transitions and embedded regions of the state are preserved by recursively invoking `transform`. Since the return type is `Element`, we use the `oclAsType`

operation to cast the copies to the required type. The do activity of a state is not preserved, as it is replaced by an additional region that is created by the mapping `DoActivity2Region` (omitted below). This new region contains the additional states and transitions described in Sect. 2.

```

import org.muml.ImplicitCopy;

transformation DoActivity(in source : SM, out target : SM);

main() {
    source.objects()[StateMachine]->map transform();
}

mapping State :: transform() : State {

    name := self.name;

    incoming := self.incoming->
        map transform().oclAsType(Transition);
    outgoing := self.outgoing->
        map transform().oclAsType(Transition);

    regions := self.regions->map transform().oclAsType(Region)
        ;
    regions += self.map DoActivity2Region();
}

```

5.3 Validation

The objective of our validation is to demonstrate the progress of our implicit copy library in terms of maintainability, compared to the baseline approach of using a HOT (cf. Sect. 3.2). To this end, we consider a refinement scenario in the context of MECHATRONICUML [5, 11], a model-driven software design method for cyber-physical systems. Since MECHATRONICUML combines numerous domain-specific languages, the resulting domain models are based on a large, industrial-scale metamodel. A key feature of MECHATRONICUML is domain-specific model checking of the system behavior [9], which is modeled using a real-time variant of UML statemachines. To this end, MECHATRONICUML uses model refinements to align its statemachines to the syntax of the model checker in use. This refinement is based on a QVTo transformation chain consisting of eight refinement steps for statemachine models [9]. Each step refines only a particular syntactic feature, namely (1) scaling of time units, (2) disambiguation of identifiers, (3) normalizing transition deadlines, (4) normalizing transitions of composite states, (5) normalizing of do activities, (6) flattening of hierarchical states, (7) normalization of entry/exit activities, and finally (8) normalizing of urgency properties of transitions.

Setting the Hypothesis. We argue that our approach outperforms the baseline by reducing the size of transformation code that requires maintenance in case of metamodel evolution. According to the quality model for QVTo proposed in [10], we use lines of code for the quantification of transformation size. In particular, we chose source lines of code (SLOC) as our base metric, which represents the overall number of lines of code, excluding whitespace and comments. On the basis of SLOC, we compute three different response variables:

1. *Overall code size* which is the total number of SLOC.
2. *Handwritten code size* as the number of SLOC that was written manually.
3. *Evolving code size* which is the number of SLOC that is metamodel-specific, including both handwritten and generated code.

Our evaluation hypothesis is that our approach improves the maintainability of the transformation chain by significantly reducing the *evolving code size* compared to the baseline approach of using a HOT.

Planning. To conduct our validation, we implement each of the eight refinement steps in four different stages, differing in what type of library is used to enable implicit copies. The stage *none* denotes a naive implementation stage without a copy library. All transformation steps implement the copying on their own, resulting in huge code duplication between the different steps. For the stage *boilerplate*, we avoid code duplication. Here, the copying is outsourced to a central library containing the handwritten boilerplate code. In contrast, the *HOT-generated* stage uses a HOT to generate this boilerplate code automatically. Finally, the *generic* stage denotes the approach of using an implicit copy library as proposed in this paper.

Validating the Hypothesis. We validate our hypothesis by traversing the four aforementioned stages of implementation. The stages *none* and *boilerplate* comprise manually implemented code only. For the stage *HOT-generated*, we manually implement a HOT in QVTo that traverses an arbitrary Ecore metamodel and generates the abstract syntax tree of a QVTo copy library. We refer the reader to [22] for structural details on the synthesized code. In the *generic* stage, we implement the implicit copy library as proposed in this paper.

After every stage, we count the SLOC of all transformation modules involved. The respective modules include (1) the eight refinement steps, (2) a copy library as used in all stages except *none*, and (3) a HOT as used only in the *HOT-generated* stage. Finally, on the basis of SLOC values for all modules and all different implementation stages, we calculate the response values for our variables by distinguishing between manually implemented and generated modules, and between metamodel-specific and metamodel-independent modules.

Analyzing the Results. In Table 3, we list the results of our validation procedure. For each implementation stage and each transformation module involved, we show the resulting number of SLOC. In the rightmost columns, we present the

resulting values of our response variables. In particular, we also show percentage values indicating the increase/decrease compared to the preceding implementation stage. According to our results, the *boilerplate* approach reduces all three variables by 71.3% compared to the type *none*. Whereas the *HOT-generated* library leads to a minor increase of the overall transformation size by 4.0% compared to the boilerplate approach, it further decreases the handwritten code size significantly by another 31.4%. With respect to the evolving code size, no difference compared to the *boilerplate* approach is detected because the additional HOT implementation is metamodel-independent. Finally, the *generic* approach reduces the overall code size by another 37.7%, and the handwritten code size slightly by 5.5%. Furthermore, we also detect a reduction of the evolving code size by 36.2%. According to this significant reduction, we regard our evaluation hypothesis as fulfilled.

Table 3. Resulting SLOC Values and Response Variables

Library Type	Time Units	Identifiers	Dead lines	Transitions	Do Activities	Hierarchy	Entry/Exit	Ur gency	Lib rary	HOT	Over all	Hand written	Evol ving
None	1062	1033	1198	1235	1281	1755	1082	1261	N/A	N/A	9907	9907	9907
Boilerplate	53	24	189	226	272	746	73	252	1009	N/A	2844 -71.3%	2844 -71.3%	2844 -71.3%
HOT-generated	53	24	189	226	272	746	73	252	1009	115	2959 +4.0%	1950 -31.4%	2844 ±0%
Generic	53	24	196	226	272	746	73	224	29	N/A	1843 -37.7%	1843 -5.5%	1814 -36.2%

Threats to Validity. A threat to the validity of our findings is that SLOC represents only the overall size of the transformation modules. Thus, it reflects the potential vulnerability to metamodel evolution, but does not capture the actual maintenance efforts over time in a real evolution scenario. Therefore, in case of infrequent metamodel evolution or limited change impact, the benefits of our approach might be less obvious. Furthermore, if it is possible to re-execute the HOT automatically (e.g., by means of continuous integration), the evolving code size might not be the crucial variable to measure maintenance effort. In this case, the handwritten code size is a more meaningful metric.

Another threat is that the HOT we developed is a model-to-model transformation and, therefore, generates an abstract syntax tree. Instead, using a model-to-text transformation to generate the concrete syntax is expected to be less cumbersome. However, since we restrict our validation to QVTo as a model-to-model approach, involving an additional model-to-text engine would lead to incomparability of the transformation size.

6 Conclusion and Future Work

In this paper, we present a transformation library that provides implicit copies for imperative transformation languages without special-purpose refinement

facilities. Thereby, we significantly reduce the verbosity of refinement transformations written in such languages restricted to general-purpose facilities. We also discuss the generality of the approach by elaborating the core facilities that imperative languages must provide in order to realize our approach. We give an overview on state-of-the-art imperative languages with respect to the core facilities. Thereby, we assess the applicability of our approach to these languages. Our proof of concept demonstrates how a reduced verbosity contributes to an improved maintainability of refinement transformations.

Our approach is beneficial for imperative transformation developers with recurring model refinement tasks, provided that their favored transformation language supports the identified facilities. If so, developing an implicit copy library will eventually save development efforts, compared to the approach of writing the boilerplate code manually for each refinement transformation. Furthermore, compared to the approach of using a HOT to synthesize the boilerplate code, our approach also saves maintenance efforts in scenarios with frequent meta-model evolution. Finally, language engineers might use our approach to provide transformation languages with a built-in standard library for implicit copies. For now, we make a ready-to-use library available to QVTo developers.

Future work on our approach includes a more fine-grained specification of refinements. Currently, refinement operations are specified per type and need to address all of the type's features, i.e., links to other types. In contrast, specifying refinements per feature is a promising extension to our approach. At the technological level, our library implementation in QVTo could benefit from generic return types for copy operations, as one of the language facilities identified in this paper. In order to avoid type casting in case of such generic, type-preserving operations, QVTo might adopt the `OclSelf` pseudo type proposed by Willink [34] as an extension to OCL. Finally, we plan to extend the scope of our work towards refining declarative transformation models [6], addressing also exogenous transformations. In particular, we intend to combine our approach with generic execution algorithms [8], assisting in situations where complex mapping models can not be executed by an algorithm, and therefore need to be refined manually using an imperative language.

Acknowledgments. The authors thank Marie Christin Platenius and Anthony Anjorin for helpful comments on earlier versions of the paper, and Mario Treiber for assisting in our validation.

References

1. Amstel, M.F., Brand, M.G.J.: Model transformation analysis: staying ahead of the maintenance nightmare. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 108–122. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-21732-6_8](https://doi.org/10.1007/978-3-642-21732-6_8)
2. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing triple graph grammars using rule refinement. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 340–354. Springer, Heidelberg (2014). doi:[10.1007/978-3-642-54804-8_24](https://doi.org/10.1007/978-3-642-54804-8_24)

3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-16145-2_9](https://doi.org/10.1007/978-3-642-16145-2_9)
4. Batot, E., Sahraoui, H.A., Syriani, E., Molins, P., Sboui, W.: Systematic mapping study of model transformations for concrete problems. In: MODELWARD 2016, pp. 176–183. SciTePress (2016)
5. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Schäfer, W., Meyer, M., Pohlmann, U.: The MechatronicUML method. In: ICSE Companion 2014, pp. 614–615. ACM (2014)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? transformation models!. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006). doi:[10.1007/11880240_31](https://doi.org/10.1007/11880240_31)
7. Denil, J., Cicchetti, A., Biehl, M., Meulenaere, P.D., Eramo, R., Demeyer, S., Vangheluwe, H.: Automatic deployment space exploration using refinement transformations. *Electronic Communications of the EASST* 50 (2011)
8. Freund, M., Braune, A.: A generic transformation algorithm to simplify the development of mapping models. In: MoDELS 2016, pp. 284–294. ACM (2016)
9. Gerking, C., Dziwok, S., Heinzemann, C., Schäfer, W.: Domain-specific model checking for cyber-physical systems. In: MoDeVVA 2015, pp. 18–27 (2015)
10. Gerpheide, C.M., Schiffelers, R.R.H., Serebrenik, A.: Assessing and improving quality of QVTo model transformations. *Software Qual. J.* **24**(3), 797–834 (2016)
11. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time UML designs. In: ESEC/FSE 2003, pp. 38–47. ACM (2003)
12. Goldschmidt, T., Wachsmuth, G.: Refinement transformation support for QVT relational transformations. In: MDSE 2008 (2008)
13. Gorp, P., Keller, A., Janssens, D.: Transformation language integration based on profiles and higher order transformations. In: Gašević, D., Lämmel, R., Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 208–226. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-00434-6_14](https://doi.org/10.1007/978-3-642-00434-6_14)
14. Hebig, R., Khelladi, D., Bendraou, R.: Approaches to co-evolution of metamodels and models: a survey. *IEEE Trans. Softw. Eng.* **43**(5), 396–414 (2016)
15. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. *Softw. Syst. Model.* **9**(3), 375–402 (2010)
16. Herrmannsdörfer, M., Wachsmuth, G.: Coupled evolution of software metamodels and models. In: Mens, T., Serebrenik, A., Cleve, A. (eds.) *Evolving Software Systems*, pp. 33–63. Springer, Heidelberg (2014)
17. Jézéquel, J.M.: Model driven design and aspect weaving. *Softw. Syst. Model.* **7**(2), 209–218 (2008)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. *Sci. Comput. Program.* **72**(1–2), 31–39 (2008)
19. Kapová, L., Goldschmidt, T.: Automated feature model-based generation of refinement transformations. In: SEAA 2009, pp. 141–148 (2009)
20. Kapová, L., Goldschmidt, T., Happe, J., Reussner, R.H.: Domain-specific templates for refinement transformations. In: MDI 2010, pp. 69–78. ACM (2010)
21. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the small with the Epsilon Wizard Language. *J. Object Technol.* **6**(9), 53–69 (2007)

22. Kraas, A.: Realizing model simplifications with QVT operational mappings. In: OCL 2014, pp. 53–62 (2014)
23. Krause, C., Dyck, J., Giese, H.: Metamodel-specific coupled evolution based on dynamically typed graph transformations. In: Duddy, K., Kappel, G. (eds.) ICMT 2013. LNCS, vol. 7909, pp. 76–91. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-38883-5_10](https://doi.org/10.1007/978-3-642-38883-5_10)
24. Lano, K., Kolahdouz Rahimi, S.: Model-transformation design patterns. *IEEE Trans. Softw. Eng.* **40**(12), 1224–1259 (2014)
25. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M.K., Syriani, E., Wimmer, M.: Model transformation intents and their properties. *Softw. Syst. Model.* **15**(3), 647–684 (2016)
26. Mens, T., van Gorp, P.: A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.* **152**, 125–142 (2006)
27. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. No. formal/15-02-01 (2015)
28. Paige, R.F., Matragkas, N.D., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. *J. Syst. Softw.* **111**, 272–280 (2016)
29. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C., Poulding, S.M.: Epsilon Flock: a model migration language. *Softw. Syst. Model.* **13**(2), 735–755 (2014)
30. Stahl, T., Völter, M.: Model-driven software development: technology, engineering, management. Wiley (2013)
31. Tisi, M., Cabot, J., Jouault, F.: Improving higher-order transformations support in ATL. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 215–229. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13688-7_15](https://doi.org/10.1007/978-3-642-13688-7_15)
32. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-02674-4_3](https://doi.org/10.1007/978-3-642-02674-4_3)
33. Wagelaar, D., van der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. *Softw. Syst. Model.* **9**(3), 285–309 (2009)
34. Willink, E.D.: Modeling the OCL standard library. *Electronic Communications of the EASST* 44 (2011)



<http://www.springer.com/978-3-319-61472-4>

Theory and Practice of Model Transformation
10th International Conference, ICMT 2017, Held as Part
of STAF 2017, Marburg, Germany, July 17-18, 2017,
Proceedings
Guerra, E.; van den Brand, M. (Eds.)
2017, XIV, 183 p. 73 illus., Softcover
ISBN: 978-3-319-61472-4