Softw Syst Model (2019) 18:1305–1344
https://doi.org/10.1007/s10270-017-0602-0

REGULAR PAPER

End-to-end model-transformation comprehension through fine-grained traceability information

Victor Guana[1] · **Eleni Stroulia**[1]

Received: 11 October 2016 / Revised: 27 February 2017 / Accepted: 7 May 2017 / Published online: 1 June 2017

Abstract The construction and maintenance of model-to-model and model-to-text transformations pose numerous challenges to novice and expert developers. A key challenge involves tracing dependency relationships between artifacts of a transformation ecosystem. This is required to assess the impact of metamodel evolution, to determine metamodel coverage, and to debug complex transformation expressions. This paper presents an empirical study that investigates the performance of developers reflecting on the execution semantics of model-to-model and model-to-text transformations. We measured the accuracy and efficiency of 25 developers completing a variety of traceability-driven tasks in two model-based code generators. We compared the performance of developers using ChainTracker, a traceability analysis environment developed by our team, and that of developers using Eclipse Modeling. We present statistically significant evidence that ChainTracker improves the performance of developers reflecting on the execution semantics of transformation ecosystems. We discuss how developers supported by off-the-shelf development environments are unable to effectively identify dependency relationships in nontrivial model-transformation chains.

Keywords Model-transformation comprehension · Transformation comprehension · Traceability analysis · Software maintenance · Development environments

Communicated by Prof. Lionel Briand.

✉ Victor Guana
guana@ualberta.ca

Eleni Stroulia
stroulia@ualberta.ca

1 Department of Computing Science, University of Alberta, Edmonton, AB, Canada

1 Introduction

Model-to-model (M2M) and model-to-text (M2T) transformations can be used to solve complex software engineering tasks. These include, but are not limited to, code generation from high-level specifications [1,2], database schema migration[3,4], code analysis and verification [5,6] and deployment automation [7,8].

Due to the complex execution semantics of model-transformation languages, the construction and maintenance of model transformations pose multiple challenges to novice and expert developers [9,10]. A key challenge involves tracing dependency relationships between artifacts of a transformation ecosystem [11]. In effect, traceability information is required to assess the impact of metamodel and transformation evolution [12–14], to debug complex transformation expressions [15], to identify transformation-refactoring opportunities [16,17], and to determine the metamodel coverage of complex transformation ecosystems [18–21]. Moreover, collecting traceability information becomes considerably more difficult when transformation ecosystems include model-transformation chains (MTCs) [22,23].

In order to increase the adoption of model-driven engineering practices, we need development environments specifically tailored to support the construction of model transformations [9,22]. This is particularly relevant in the context of modern software developers, that continuously experiment with development technologies, and that quickly abandon tools with no evident economic return [24,25]. Unfortunately, most of the existing development environments for model transformations have been tailored for model-driven engineering experts, thus limiting their adoption among the general software engineering community [26].

In [27], we introduced a model-transformation analysis technique to gather fine-grained traceability links. In [28,29], we introduced ChainTracker, a traceability collection and analysis environment that enables developers to explore the execution semantics of model-to-model, and model-to-text transformations. The main goal of this paper is to evaluate the usability of ChainTracker using Eclipse Modeling as the industry baseline.

ChainTracker is built on top of a generalizable traceability analysis technique for transformation technologies that use the Object Constraint Language (OCL) [30] as underlying model manipulation formalism. ChainTracker considers model-to-model and model-to-text transformations individually, and in nontrivial transformation chains.

The analysis environment includes interactive traceability visualizations, and projectional code editors targeted at novice transformation developers. Furthermore, it provides features such as binding filters and code highlighters that support experienced developers assessing the impact of metamodel changes, and more effectively debugging nontrivial transformation expressions. Currently, ChainTracker is capable of analyzing ATL [31] model-to-model transformations, and Acceleo [32] model-to-text transformations, two widely adopted model-transformation technologies in industry and academic environments.

Although model transformations can be used to tackle numerous software engineering tasks, over the last 10 years *model-based code generation* has been the flagship paradigm used to promote their adoption among the general software engineering community. Model-based code generators integrate model-to-model and model-to-text transformations to build applications that systematically differ from each other. More often than not, code generators use multistep transformation chains to translate high-level system specifications, captured by domain-specific languages, into executable artifacts, i.e., code and deployment scripts [33,34].

Like all software, model-based code generators are bound to evolve [35]. Evolutionary changes in model-based code generators can be classified in two scenarios of evolution: *metamodel evolution* and *platform evolution* [35]. In the metamodel evolution scenario, changes to underlying domain languages are required to improve their expressiveness, and to better capture information relevant to the to-be-constructed systems. In the platform evolution scenario, changes to generated artifacts are required to meet new requirements not captured by the generation infrastructure. Such modifications also include code refactoring for design improvement, performance tuning for mission critical systems, energy consumption optimization, and bug fixes [36]. In both scenarios of evolution, model-to-model and model-to-text transformations potentially need to be modified in order to reflect changes in a systematic way.

In the last few years, tools have been proposed to automatically assess the impact of changes in model transformations [11,37]. However, automatically performing adaptive changes in a transformation ecosystem is extremely hard. This task must consider both the purpose of the modification (i.e., updative, adaptive, performance, corrective or reductive) and its technical aspects (i.e., the when, where, what and how of changes) [38]. In this study, we hypothesize that enabling developers to interactively explore the execution semantics of a transformation ecosystem can significantly improve the performance of developers reflecting on its design and evolution. In this context, developers need to interpret the execution semantics of transformations in order to answer traceability-driven questions such as:

- **T1.** *What is the order of precedence for the correct execution of the transformations in my ecosystem?*
- **T2.** *How well is the information captured by the metamodels used by the transformations in my ecosystem?*
- **T3.** *What transformation bindings intervene in the generation of this line of code?*
- **T4.** *What metamodel elements are derived using this element or property?*
- **T5.** *Are there unused rules or binding expressions in the transformations that comprise my ecosystem?*

This paper presents an empirical study that investigates the performance of developers when reflecting on the execution semantics of model-to-model and model-to-text transformations. In effect, we measured the accuracy and efficiency of developers when asked to identify dependency relationships between transformation artifacts using ChainTracker. Furthermore, we compared their performance with that of developers using Eclipse Modeling. We intend to investigate two research questions:

- **RQ1**: Do developers using ChainTracker identify metamodel and generation dependencies in transformation ecosystems more accurately and efficiently than those using Eclipse Modeling?
- **RQ2**: Do the size and complexity of transformation ecosystems affect the effectiveness of ChainTracker in helping developers identifying their metamodel and generation dependencies?

We found that when using Eclipse Modeling, most developers could not effectively identify metamodel dependencies defined in nontrivial model-to-model transformations. Furthermore, we observed that developers were unable to precisely pinpoint dependencies between metamodels and generated textual artifacts in the context of model-to-text transformations. Our study also revealed that developers often fail to identify chained upstream and downstream metamodel dependencies in both linear and multibranched model-transformation chains. Moreover, we found that ChainTracker's interactive visualizations and projectional editors considerably improve the performance of developers reflecting on the execution semantics of transformation ecosystems.

The remainder of this paper is structured as follows. Section 2 reviews related work on traceability collection and visualization techniques. Section 3 presents a detailed description of the ChainTracker analysis environment. Section 4 introduces the hypotheses of our empirical study. Section 5 discusses our experimental protocol. Section 6 presents the results of our study. Section 7 discusses our findings, and key take-home messages. Section 8 presents our threats to validity. Section 9 summarizes our contributions and discusses our future avenues of research.

2 Related work

The term *traceability* is not regarded very strictly by authors in the model-driven engineering community [14]. We elaborate on the traceability definition proposed by Winkler et al. [14] in which traceability is understood as "*the ability* to collect traceability links from a set of transformation expressions." Traceability links are understood as *the relations* between a set of artifacts in a transformation ecosystem. These artifacts include a transformation's codebase, its source and target metamodels, corresponding model instances, and potential generated artifacts. Moreover, a traceability link is understood as a "dependency relationship between two artifacts *a1* and *a2*, in which *a2* relies on the existence of *a1*, or that changes in *a1* potentially result in changes in *a2*" [14,39,40]. We consider the term *transformation ecosystem* as the set of artifacts that comprise one or multiple transformations that work cooperatively in a model-based software engineering tool.

2.1 Extracting traceability information

Surveys such as in [41], and more recently in [14,42] have compiled the existing work toward collecting traceability links from model-to-model and model-to-text transformations. Current analysis techniques conceive traceability links as dependency relationships between a variety of transformation artifacts, in different levels of granularity, and conforming to different levels of abstraction.

Authors such as in [43–47] have proposed traceability analysis techniques for model-to-model transformations. They conceive traceability links as dependency relationships between the models used by a transformation and those produced after its execution, i.e., traceability at the model level of abstraction. Yet other researchers such as in [48–50] conceive traceability links as the symbolic dependencies between the metamodels used by a transformation, and its corresponding binding expressions, i.e., traceability at the metamodel level of abstraction. Most the aforementioned techniques identify dependency relationships by means of a) instrumenting and executing the transformations under analysis, thereby obtaining traceability information as a byproduct of a transformation execution itself, or b) comparing the source and target models of a transformation in order to infer its execution mechanics. Other traceability techniques such as in [51–54] investigate how to collect traceability links in model-to-text transformations. They conceive traceability links as relationships of dependency between metamodels, code templates, and generated textual artifacts.

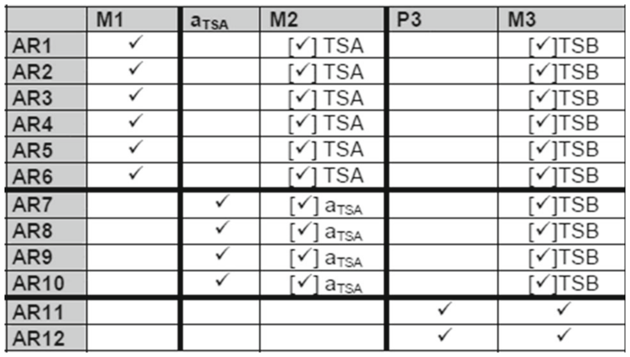
To the best of our knowledge, there are no proposals that provide a unified traceability visualization technique for heterogeneous transformation compositions, i.e., transformation chains that combine both model-to-model and model-to-text transformations. Moreover, none of the existing techniques provide analysis capabilities to identify end-to-end fine-grained traceability links. Access to fine-grained traceability links significantly increases the developers' ability to reflect on the execution semantics of complex transformation ecosystems (Sect. 7).

2.2 Visualizing traceability links

Multiple techniques have been proposed to diagrammatically depict traceability information in software systems. Most of these techniques have been developed in the field of requirements engineering [14]. They have inspired little, but precious work on visualizing traceability information in the context of model-driven engineering. According to Wieringa [55], traceability visualizations can be categorized in three main groups: matrices, cross-references, and graph-based representations. Let us briefly discuss each one of them and provide examples of traceability visualizations that follow their design guidelines.

2.2.1 Matrix representations

Traceability matrices portray traceability links between a two-dimensional set of software artifacts. They follow a grid-based layout in which rows and columns capture information about two families of related entities. Primitive traceability matrices represent the existence of a dependency relationship between two artifacts by placing a mark, such as a black box, in their corresponding intersecting cell [14]. Almeida et al. in [56] use a matrix-based representation to study the conformance relationships between the implementation of a model-to-model transformation and its application domain (Fig. 1). Traceability matrices provide developers with little information about the type of relationship that a traceability link represents. However, enhancements can be made to matrices in order to enrich the information that they convey [55]. For example, matrices can be made interactive as to



	M1	a_{TSA}	M2	P3	M3
AR1	✓		[✓] TSA		[✓]TSB
AR2	✓		[✓] TSA		[✓]TSB
AR3	✓		[✓] TSA		[✓]TSB
AR4	✓		[✓] TSA		[✓]TSB
AR5	✓		[✓] TSA		[✓]TSB
AR6	✓		[✓] TSA		[✓]TSB
AR7		✓	[✓] a_{TSA}		[✓]TSB
AR8		✓	[✓] a_{TSA}		[✓]TSB
AR9		✓	[✓] a_{TSA}		[✓]TSB
AR10		✓	[✓] a_{TSA}		[✓]TSB
AR11				✓	✓
AR12				✓	✓

Fig. 1 Traceability cross-table used in [56] to relate the models, application requirements, and transformations scripts of an ecosystem

allow navigation to specific linked artifacts, such as using pop ups [57] or color encoded properties [58].

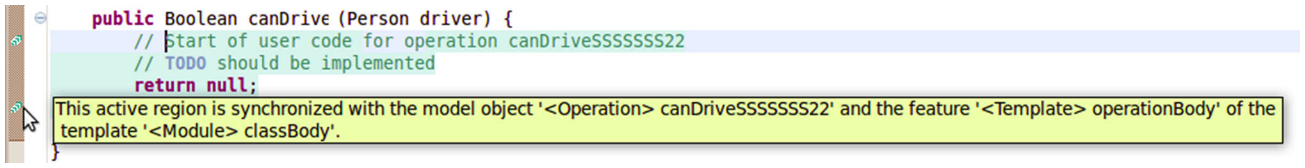
Traceability matrices are easy to understand by expert and novice developers. However, they have several limitations when used to represent the traceability links in a transformation ecosystem. A matrix representing the symbolic dependencies between the source and target metamodels of a transformation can be extremely cluttered and overwhelmingly large. Furthermore, the size of such traceability matrix will depend on the size of each underlying metamodel, and the complexity of the transformations under analysis. Research has shown that large traceability matrices become unreadable very quickly [59,60], and that their two-dimensional nature makes them unsuitable to represent n-ary traceability links, or links between hierarchical artifacts [14]. This is a common scenario in the context of transformation ecosystems, in which artifacts such as metamodels and transformations have hierarchal structures.

2.2.2 Cross-reference representations

Traceability links can be expressed as cross-references embedded in the artifacts of a software system [63]. Cross-references can be represented using natural language, or using interactive referencing features such as hyperlinks. In their most simple form, cross-references can be found as in-line documentation in source code and design documents. In the context of rule-based transformation languages such as ATL, RubyTL [64] and ETL [65], it is a widespread practice to document the source code of every transformation rule with cross-referencing notes, e.g., *"This rule transforms element A into element B"* or *"This rule uses helper X."*

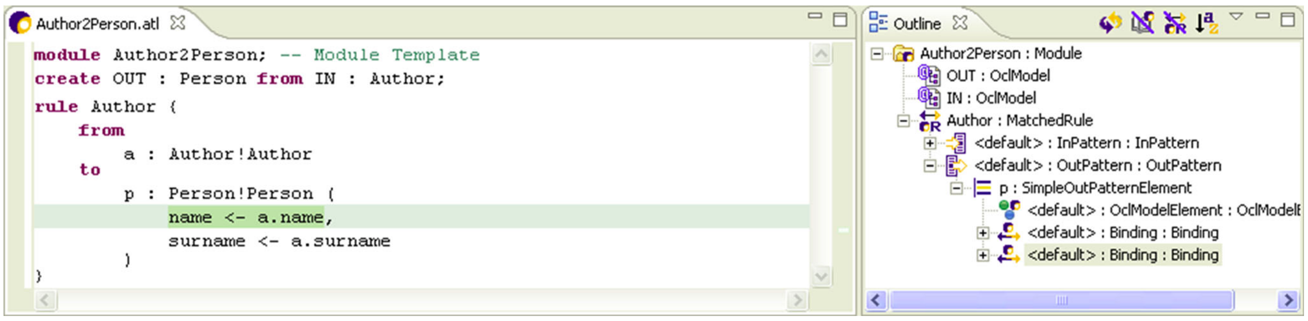
Hyperlinks enable developers to navigate through the traceability links of a given artifact, in order to switch between their different contexts. A limitation of this approach is that hyperlinks can only reveal localized outgoing and upcoming traces between two artifacts [14]. Cross-referencing is very common in modern integrated development environments. For example, the Eclipse plug-ins for transformation technologies such as ATL and Acceleo allow developers to use interactive code editors and navigate through their execution dependencies via cross-referencing hyperlinks. Developers can click on the procedural calls between transformation rules in order to have access to their definition from outlying segments of code. Similarly, Eclipse plug-ins enable developers to explore detailed information about individual metamodel elements, by means of hyperlinks that open views with detailed listings describing their relationships and properties (Figs. 2, 3).

Even though cross-reference representations allow the navigation of interdependent traceability links, they do so at the cost of limiting the scope of its views to one single artifact



```
public Boolean canDrive (Person driver) {
    // Start of user code for operation canDriveSSSSSSS22
    // TODO should be implemented
    return null;
This active region is synchronized with the model object '<Operation> canDriveSSSSSSS22' and the feature '<Template> operationBody' of the
template '<Module> classBody'.
}
```

Fig. 2 Acceleo Eclipse plug-in: in-line cross-referencing editor [61]



Author2Person.atl

```
module Author2Person; -- Module Template
create OUT : Person from IN : Author;
rule Author {
    from
        a : Author!Author
    to
        p : Person!Person (
            name <- a.name,
            surname <- a.surname
        )
}
```

Outline

```
Author2Person : Module
    OUT : OclModel
    IN : OclModel
    Author : MatchedRule
        <default> : InPattern : InPattern
        <default> : OutPattern : OutPattern
            p : SimpleOutPatternElement
                <default> : OclModelElement : OclModelE
                <default> : Binding : Binding
                <default> : Binding : Binding
```

Fig. 3 ATL Eclipse plug-in: cross-referencing multipanel editor [62]

at the time. This makes cross-referencing a poor alternative to portray global dependency views between artifacts in model-transformation chains. Furthermore, using cross-reference representations to visualize n-ary links is highly impractical [14]. Representing n-ary traceability links is a fundamental requirement in model-transformation ecosystems. Complex ecosystems usually involve multiple fine-grained artifacts with multiple outgoing and upcoming dependency relationships. As a concrete example, please consider the dependency relationships between a model-to-text transformation and a generated segment of code. A metamodel element can be used in the generation of multiple lines of code, and a line of code may be the result of querying multiple metamodel elements in a single binding expression [52].

2.2.3 Graph-based representations

Most artifacts in model-based software engineering tools are represented using both graphical and textual concrete syntaxes, e.g., a metamodel can be studied in its textual structured form, or as a class diagram that captures its elements and properties. The dual nature of artifacts in transformation ecosystems makes diagrams and general graph-based representations the most common mechanism to represent their traceability information [14]. Let us now review current graph-based approaches to represent traceability information in transformation ecosystems.

Falleri et al. [43] propose an imperative language to collect traceability links from model-to-model transformations. It represents traceability information as a bipartite graph in which nodes represent individual model elements, and edges their dependency relationships. This technique supports traceability analysis in Kermeta [66], a transformation language developed by the same authors. Falleri et al. use Graphviz [67] in order to create a simple visual representation of their trace graph. It is important to mention that due to the static nature of the visualization, no additional information can be obtained by means of interacting with it.

In [45], Von Pilgrim et al. present a traceability tool for ATL, and MTF.[1] Similar to Falleri's technique, this tool conceives traceability links at the model level of abstraction . Furthermore, it supports the collection and visualization of model-to-model transformation chains. Authors use GEF3D [69] to visualize a collection of overlapped 2D class diagrams linked by edges in a 3D space (Fig. 4). Each layer of the visualization captures a diagram corresponding to the models resulting from the execution of each shackle of a transformation chain. The 3D visualizations presented in this tool have numerous scalability issues. Considering that model instances may contain several elements, handling the visualization of large class diagrams is challenging in terms

[1] http://goo.gl/YcWHNX.

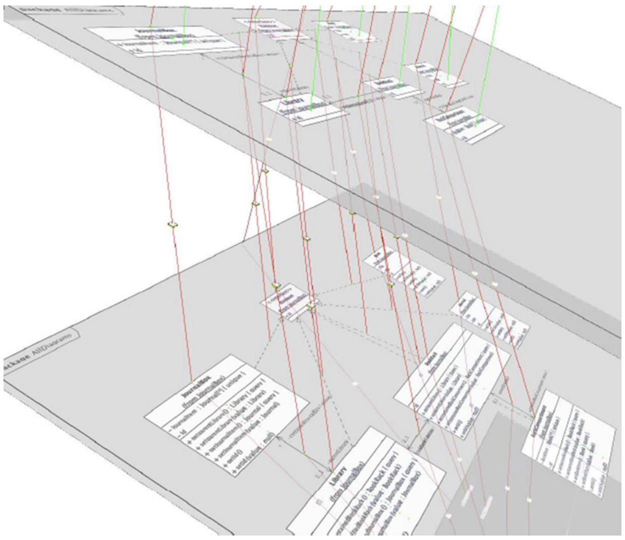


Fig. 4 Traceability 3D visualization created using GEF3D [45]

of memory space [70]. Furthermore, in terms of usability, research has shown that large class diagrams pose significant cognitive challenges to developers when filtering, isolating, and summarizing information [71,72], which is exacerbated by the 3D overlapping nature of the proposal.

Van Amstel et al. [50,68] present a tool that gathers and visualizes traceability information in the context of ATL model-to-model transformations. In [50], the tool includes features to collect symbolic dependencies between the artifacts of model-to-model transformation chains. More recently, in [68] Van Amstel et al. presented an extension of their work targeted at identifying traceability links at the model level of abstraction. Both tools use TraceVis [73] to visualize traceability information. They highlight the hierarchical structure of the transformation expressions that determine the presence of symbolic dependencies between source and target metamodel elements, e.g., grouping them in *helpers, matched rules, lazy matched rules, unique lazy matched rules*, and *called rules*. TraceVis supports hierarchical edge bundling which makes both tools highly scalable in front of large ecosystems [74]. A similar visualization approach based on TraceVis is presented by Di Rocco et al. in [48].

Santiago et al. introduce iTrace [47], a framework for the management and analysis of traceability information in model-driven engineering. iTrace identifies traceability links as symbolic dependencies between metamodel elements given ATL model-to-model transformations. It offers two dashboards to the end user, namely the overview dashboard, and workload dashboard.

The overview dashboard presents a tabular view that summarizes the metamodel elements used by a transformation. The workload dashboard presents information about

Table 1 Traceability visualization techniques for model transformations and transformations compositions

	Transformation type			Level of abstraction	
Visualization technique	M2M	M2T	MTC	Meta-level	Model-level
Matrix-based					
Almeida et al. [56]	■				■
Santiago et al. [47]	■		■	■	
Cross-reference based					
ATL Eclipse plug-in [62]	■			■	■
Acceleo Eclipse plug-in [61]		■		■	■
Graph-based					
Falleri et al. [43]	■				■
Van Amstel et al. [50]	■			■	
Van Amstel et al. [68]	■		■		■
Santiago et al. [54]		■			■
Von Pilgrim et al. [45]	■		■		■
Di Rocco et al. [48]	■			■	

a transformation's runtime behavior, including the number of elements processed by each of its transformation rules. In [54], iTrace was extended to support the visualization of model-to-text transformations using a multipanel editor. The editor includes information such as the model elements used by a model-to-text transformation, and the textual artifacts derived from its execution. Unfortunately, iTrace has not been designed to support the visualization of model-transformation chains; it considers traceability links in model-to-model and model-to-text separately, and in different levels of abstraction. Furthermore, iTrace portrays traceability links using interactive two-dimensional tables, thus suffering from the limitations of matrix-based representations discussed in Sect. 2.2.1.

Table 1 summarizes the traceability visualization techniques discussed in this section. It is important to mention that none of the reviewed techniques proposes a traceability visualization for model-to-text transformations as a part of a model-transformation chain. Furthermore, to the best of our knowledge, none of the traceability collection techniques has been empirically validated with nontrivial case studies. They are usually presented using pedagogical examples of small complexity that do not necessarily reflect on challenges of collecting and representing traceability information in complex transformation ecosystems. In effect, most of the reviewed proposals provide traceability collection and visualization techniques that only deal with coarse-grained traceability links, e.g., between source and target metamodel elements (in case of model-to-model transformations), and metamodel elements and generated files (in case of model-to-text transformations). This limits their usability in the context of transformation design and evolution scenarios that require fine-grained traceability links, such as when reflecting on the dependency relationships between the properties of metamodel elements in an ecosystem. More importantly, even though most proposals claim that their traceability collection and visualization techniques help developers to build and maintain transformation ecosystems in a more effective or efficient fashion, none of them has been empirically validated in controlled experiments with real developers. Moreover, none of the tools reviewed in this section is publicly available for download to be studied or compared by other research teams. To the best of our knowledge, ChainTracker is the first traceability collection and visualization technique to be formally evaluated with real developers using nontrivial case studies.

3 ChainTracker

ChainTracker collects and visualizes traceability links from rule-based model-to-model, and template-based model-to-text transformations. Furthermore, it collects traceability links from individual transformations, and model-transformation chains. It takes as input one or multiple model-transformation scripts, along with their corresponding source and target metamodels, and optionally model instances that conform to these metamodels. Our traceability analysis technique combines static and dynamic analysis strategies to gather fine-grained traceability links [28]. In the case of model-to-model transformations, ChainTracker considers traceability links as symbolic dependencies between three main artifacts, a) a transformation's source metamodel, b) a transformation's target metamodel, and c) a transformation's binding expressions. Likewise, ChainTracker considers traceability in model-to-text transformations as symbolic dependencies between a) a transformation's source metamodel, b) a transformation's binding expressions, and c) the lines of text generated after its execution.

A transformation ecosystem can be understood as implementation units with runtime behaviors, or as a collection of static elements that are bound to each other at development time. The ChainTracker analysis environment presents two views of a transformation ecosystem using an interactive multi-view approach. According to Clements et al. [75], a view can be understood as the *representation of a set of system elements and the relationships associated with them*. Each view defines the concrete and abstract syntaxes of the elements and relationships of the system. The goal of having multiple views for a software system is to enable developers to think about the architecture of a software system in multiple ways simultaneously. In [76], Soni et al. present the four types of views that allow developers to reflect on a system's architecture using complementary perspectives:

1. The *conceptual* view describes a system in terms of its major design elements and relationships.
2. The *module* view presents a system as set of implementation or functional units.
3. The *execution* view reflects on the runtime behavior and interactions of a system.
4. The *code* view portrays how a system's implementation units relate to non-software elements in its environment.

ChainTracker proposes a visualization technique for the *module* and *execution* views of a transformation ecosystem. It enables developers to visualize the different artifacts that comprise an ecosystem and its execution mechanics, as well as to use projectional code editors to simultaneously explore its underlying transformation scripts. Let us now present a small case study that will serve as a running example to explore ChainTracker's views and their aggregation in an unified analysis environment.

3.1 Motivating example: bank to credit report

The *Bank to Credit Report* example is comprised by a simple transformation chain. It takes as input a model describing the credit card transactions conducted by the customers of a bank. It then derives a simple report that summarizes the overall credit status of each of the bank's customers. The metamodels Bank and Report capture the specification of the source and target domains of the transformation chain (Fig. 5). The case study is composed by one model-to-model and one model-to-text transformation, namely Bank2Report (Listing 1) and Report2HTML (Listing 2), respectively.

The Bank2Report (M2M) transformation is comprised by two matched rules and three helpers. The main purpose of the helpers is to aggregate a customer's total credit (Listing 1, line 25), and total balance (Listing 1, lines 28 and 31).

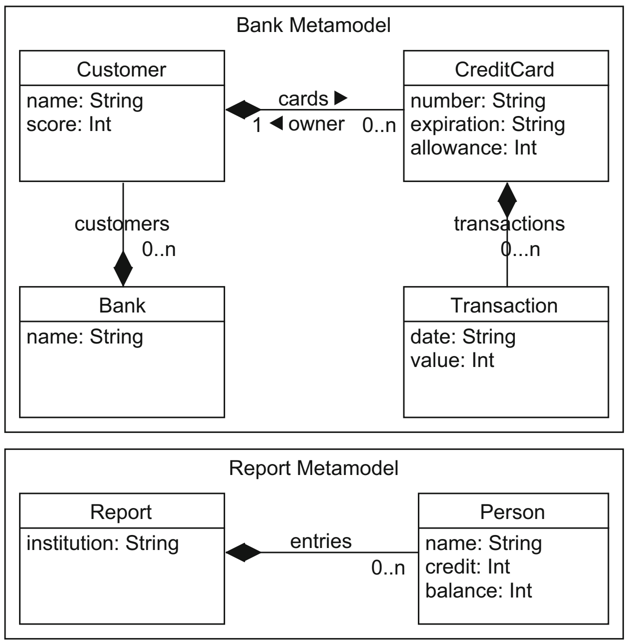


Fig. 5 Bank and report metamodels

```
1  module Bank2Report;
2  create OUT : Report from IN : Bank;
3
4  rule Main{
5      from
6          b: Bank!Bank
7      to
8          r: Report!Report(
9              institution <– b.name,
10             entries <– b.customers
11         )
12 }
13
14 rule Customer2Person{
15     from
16         c: Bank!Customer
17     to
18         p: Report!Person(
19             name <– c.name,
20             credit <– c.getTotalCredit(),
21             balance <– c.getTotalSpent()
22         )
23 }
24
25 helper context Bank!Customer def : getTotalCredit() : Integer
26 =self.cards–>collect(c|c.allowance).sum();
27
28 helper context Bank!Customer def : getTotalSpent() : Integer
29 =self.cards–>collect(c|c.getBalance()).sum();
30
31 helper context Bank!CreditCard def : getBalance() : Integer
32 =self.transactions–>collect(t|t.value).sum();
```

Listing 1 Bank2Report Model-to-Model Transformation

The Report2HTML (M2T) transformation generates a textual report summarizing the credit card portfolio of a bank. The Report2HTML transformation uses an iterator that produces an HTML table with the total credit and outstanding balance of each customer (Listing 2, lines 8–16).

```
1 [comment encoding = UTF−8 /]
2 [module generateHTML('http://ualberta.ssrg.report')]
3 [template public generateHTML(aReport : Report)]
4 [comment @main/]
5 [file ('report.html', false, 'UTF−8')]
6 <h2> [aReport.institution/] Customer Credit Report </h2>
7 <table border='''' style="with:50%">
8 [for (p : Person | entries)]
9       <tr>
10          <td>
11              <p><h4>[p.name/]</h4></p>
12              <p>Total Credit:[p.credit/]</p>
13              <p>Balance:[p.balance/]</p>
14          </td>
15      </tr>
16 [/for]
17 </table>
18 [/file]
19 [/template]
```

Listing 2 Report2HTML - Model-to-Text Transformation

3.2 The ChainTracker analysis environment

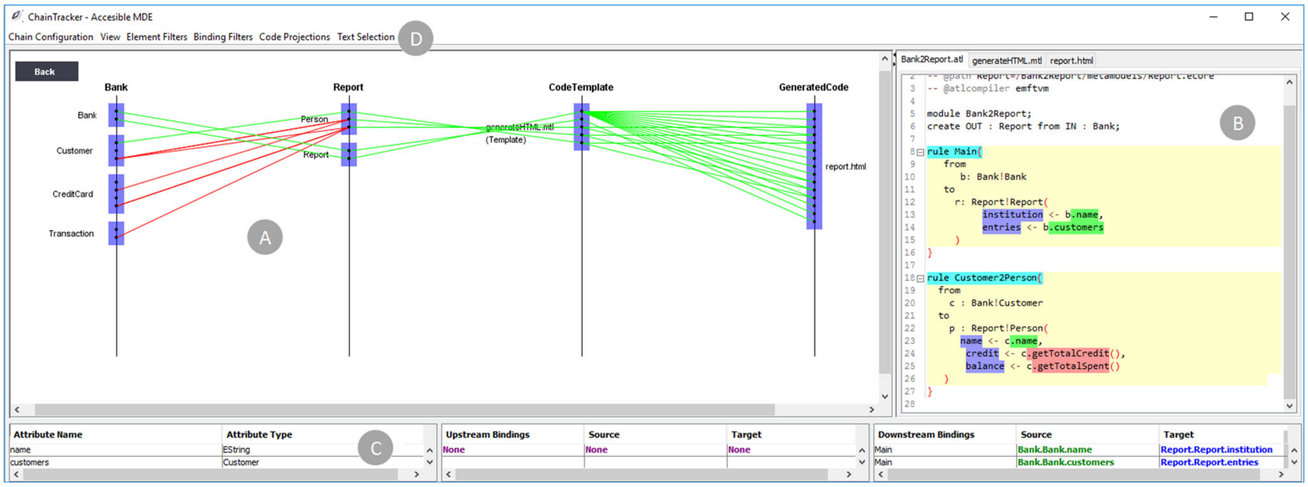
The ChainTracker analysis interface is divided in three main areas: the *transformation visualizations*, the *projectional code editors*, and the *contextual tables* (Fig. 6A–C, respectively). Each area of the analysis environment is synchronized with each other to provide developers with an unified and context-aware experience. ChainTracker helps developers to investigate how source metamodel elements and their attributes are transformed into different intermediate metamodels, and into final textual files. ChainTracker not only allows developers to explore the visualizations of model-to-model and model-to-text transformations, but also to project their information onto the analyzed scripts using highlighters. Furthermore, if concrete model instances are provided to the analysis environment, ChainTracker is capable of executing transformation ecosystems, in order to include generated textual files in the traceability analysis and visualization process. Let us briefly discuss each one of the areas of the environment using our *Bank to Credit Report* example, and the traceability-driven questions listed in Sect. 1.

3.2.1 The transformation visualizations

ChainTracker provides two different types of transformation visualizations: the *overview visualization* (Fig. 7) and the *branch visualization* (Fig. 8). Developers can switch between visualization types using ChainTracker's command menu (Fig. 6D)

The *overview visualization* presents a *module view* of the ecosystem under analysis. This view enables developers to abstract the complexity of individual and isolated transformation scripts, into a single picture that summarizes its compositional structure. It follows a graph-based approach in which blue nodes represent the source and target artifacts of a transformation step. In the case of model-to-model transformations, blue nodes portray source and target metamodels. In model-to-text transformations, blue nodes portray textual templates and generated textual artifacts such as code. Edges in this visualization diagrammatically depict dependencies between the steps of a transformation chain. As a concrete example, Fig. 7 portrays the *overview visualization* of the *Bank to Credit Report* example.

The *overview visualization* can be used to quickly obtain information about the order of precedence of the transformation in a complex transformation ecosystem (T1). Developers can click on the edges of the visualization to obtain information boxes with details about the ecosystem's transformation scripts, code templates, and generated textual artifacts. For example, Fig. 7A presents information corresponding to the transformation rules contained in the Bank2Report transformation (Listing 1, line 14 and 4, respectively), Fig. 7B portrays the templates comprised by the Report2HTML transformation (Listing 2, line 3), and Fig. 7C presents the list of the generated files derived after its execution, i.e., *report.html*.



```
-- @path Report=/Bank2Report/metamodels/Report.ecore
-- @atlcompiler emftvm

module Bank2Report;
create OUT : Report from IN : Bank;

rule Main{
   from
      b: Bank!Bank
   to
     r: Report!Report(
          institution <- b.name,
          entries <- b.customers
      )
}

rule Customer2Person{
   from
     c : Bank!Customer
   to
     p : Report!Person(
       name <- c.name,
        credit <- c.getTotalCredit(),
        balance <- c.getTotalSpent()
    )
}
```

Attribute Name	Attribute Type
name	EString
customers	Customer

Upstream Bindings	Source	Target
None	None	None

Downstream Bindings	Source	Target
Main	Bank.Bank.name	Report.Report.institution
Main	Bank.Bank.customers	Report.Report.entries

Fig. 6 ChainTracker—main screen

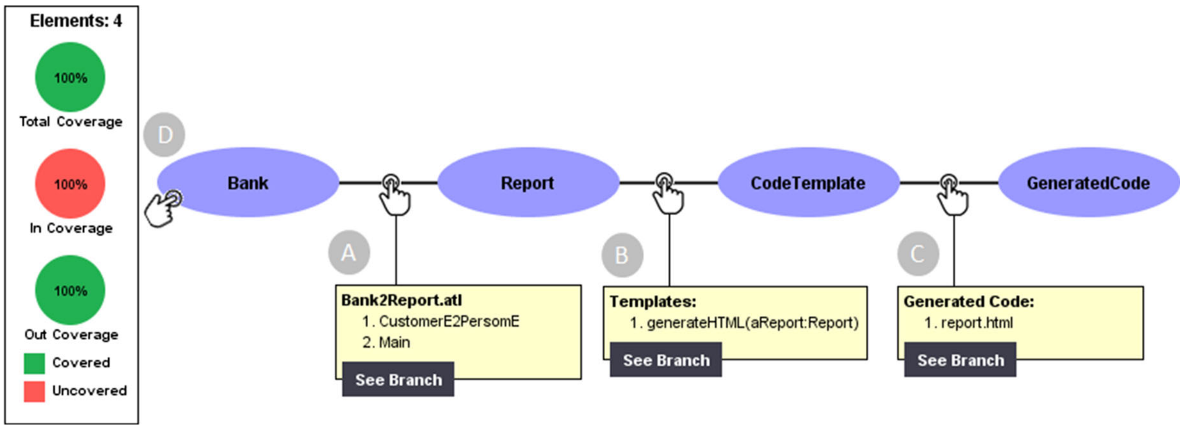


Fig. 7 ChainTracker—overview visualization

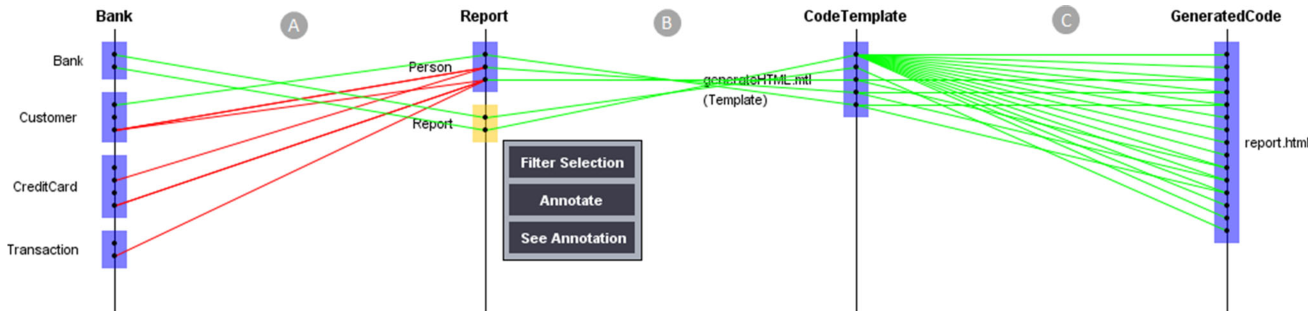


Fig. 8 ChainTracker—branch visualization

In this small example, determining the high-level compositional structure of a transformation ecosystem is a trivial task. However, as we present in Sect. 7, as the number and size of transformation scripts increases, so does the difficulty of understanding how they operate and execute collaboratively.

ChainTracker automatically identifies the dependencies between transformation scripts and determines its order of precedence in the case of model-transformation chains. The *overview visualization* provides quick insights on the branching structure of a composition, by creating a graphical representation of its major implementation units. Summarizing information about the ecosystem's high-level structure enables developers to assess and, potentially, optimize its overall design and correctness [77].

In order to study how well the information captured by the metamodels of an ecosystem is used by its transformations (T2), developers can click on the nodes of the overview visualization to determine their coverage throughout the different steps of a transformation chain (Fig. 7D). Coverage metrics provide insights about how well the information captured by a metamodel is used by the transformations of an ecosystem. In effect, coverage information is a vital component when reflecting about the quality of a transformation composition, or when assessing the impact of evolutionary changes. In ChainTracker, coverage metrics are summarized in contextual pie charts that contain in- and out-coverage metrics. The in-coverage metric reports the percentage of elements in a metamodel that are effectively targeted by the bindings defined in the transformations of an ecosystem. The out-coverage metric represents the percentage of elements in a metamodel used by transformation bindings to either generate textual artifacts, or to derive intermediate models in a multistep transformation chain. This information might lead developers to remove unused elements from a metamodel, or to take advantage of their semantic value and include them in the scope of a transformation. Maximizing metamodel coverage makes transformations less convoluted and less error prone, while, at the same time, freeing metamodels from unused semantic constructs [18].

In the *Bank to Credit Report* case study, the out-coverage of the Bank metamodel is 100% (Fig. 7D). This means that all the elements of the Bank metamodel are used by the Bank2Report transformation. Conversely, the in-coverage metric for the same metamodel is 100% uncovered since it is the root of the transformation chain, and no binding targets its elements.

The *branch visualization* presents an *execution view* of the transformation ecosystem under analysis (Fig. 8). The goal of the *branch visualization* is to present information about the fine-grained traceability links that exist throughout a transformation ecosystem. This visualization aims at portraying all the symbolic dependencies that individual binding expressions establish between metamodel elements and their properties, textual templates, and potential generated artifacts.

The *branch visualization* follows a graphical notation inspired by parallel-coordinate visualizations for hyper-dimensional data [78]. Parallel coordinates have been widely used to represent the relationships between multidimensional datasets. They are commonly used to represent relationships between continuous numerical variables grouped in interdependent classes.[2] In the context of transformation ecosystems, the artifacts in each step of a transformation chain, including the resulting files obtained after its execution, can be considered as an interdependent set of categorical information. Indeed, each set is interconnected by means of transformation expressions that bind their comprising elements.

The ultimate purpose of a transformation chain is to reduce, split, merge, or augment the information provided as input to systematically transform it, into one or multiple output representations that conform to a textual or metamodel syntax. In transformation ecosystems that involve model-to-model and model-to-text transformations, we can find at least three semantic dimensions in the transformation process: metamodels, textual templates, and generated textual artifacts. The *branch visualization* captures each step of a transformation composition and portrays its corresponding artifacts in individual coordinate dimensions.

Figure 8 portrays the *branch visualization* of the *Bank to Credit Report* case study. Each semantic dimension of its underlying transformation chain is represented using vertical lines. Depending on the nature of each transformation step, its corresponding graphical notation contains a different set of artifacts. In the case of model-to-model transformations, vertical lines contain blue boxes that portray the elements of its source and target metamodels; black dots inside these boxes represent their corresponding properties (Fig. 8: Bank and Report metamodels). For model-to-text transformations, vertical lines contain blue boxes that represent individual template modules, and black dots portray binding statements embedded in templates that access the properties of a metamodel element (Fig. 8: *generateHTML.mtl*). Moreover, in the context of generated textual artifacts, each blue box represents a generated text file, and black dots individual lines of text generated inside the file (Fig. 8: *report.html*). Indeed, blue boxes are multi-purpose visual elements for artifacts that have hierarchical structure, such as metamodels (that contain elements and properties), templates (that contain template modules and individual binding statements), and generated textual files (that include non-variable text snippets, and variable generated lines of text). The *branch visualization* combines the multidimensional properties of parallel-coordinate visualizations, and the scalability and filtering power of hierarchical edges [74] to visualize adjacency relations in hierarchical data.

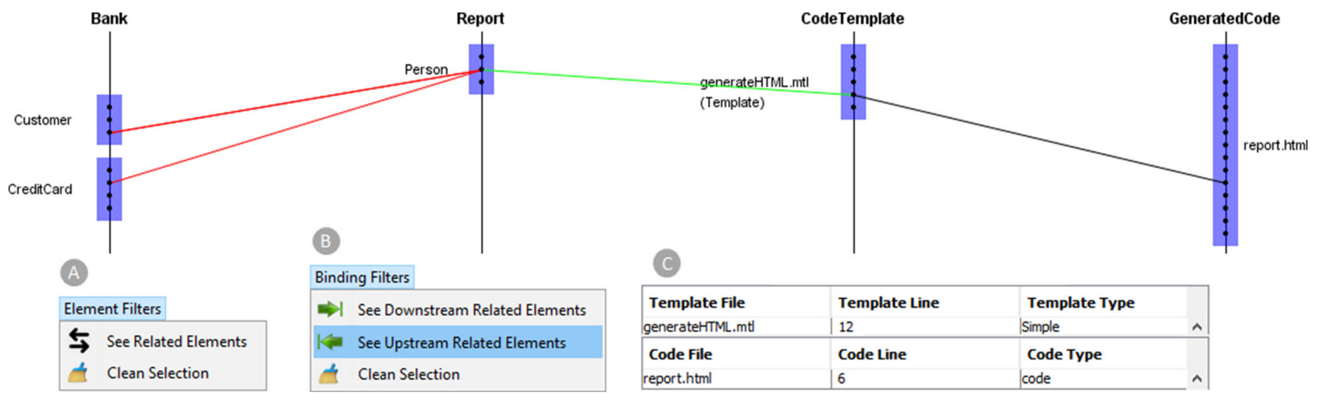
Figure 8 also presents the different types of edges used to represent traceability links in a transformation ecosystem. These include *model-to-model transformation bindings* (Fig. 8A), *model-to-text transformation bindings* (Fig. 8B) and *generation bindings* (Fig. 8C). Model-to-model and model-to-text bindings represent metamodel query expressions in transformation scripts. Bindings represent all the potential fine-grained traceability links between artifacts of the transformation ecosystem. They dictate how the properties of a metamodel element are used in order to derive a target metamodel property (in the case of model-to-model transformations) or a line of text (in the case of a model-to-text transformation). Furthermore, generation bindings are understood as the runtime dependencies between a model-to-text transformation and a generated textual file. They represent the dependency relationships between a template that queries a set of metamodel properties, and the lines of text generated after its execution.

The *branch visualization* can be used to grain access to end-to-end traceability information of a transformation chain. It can be used to identify the metamodel and binding expressions that intervene in the generation of a line of code (T3). Furthermore, this visualization enables developers to identify upstream and downstream metamodel dependencies in isolated or composed model-to-model transformations (T4).

In order to help developers to more accurately identify such dependencies, ChainTracker distinguishes between two types of bindings, namely *explicit* and *implicit* bindings [28] (Fig. 8, green and red edges, respectively). Explicit bindings reflect on the dependency relationships caused by assignment expressions in a transformation. Implicit bindings portray dependency relationships given by intermediate expressions that manipulate, constrain, or navigate the structure of a metamodel in order to realize the intent of an assignment. Recently, researchers such as in [21] have considered implicit bindings as the "footprints" of a transformation expression. In the context of ATL and Acceleo transformation technologies, these expressions are specified using OCL [79].

Distinguishing between explicit and implicit bindings enables developers to study the coarse-grained dependencies when assessing the impact of changes in a transformation ecosystem. Furthermore, it also helps them to consider individual statements in complex expressions and potentially discovers fine-grained artifacts that are indirectly bound by navigation or constrain expressions.

[2] Multiple examples of parallel-coordinate visualizations are available at https://syntagmatic.github.io/parallel-coordinates/.



Template File	Template Line	Template Type
generateHTML.mtl	12	Simple

Code File	Code Line	Code Type
report.html	6	code

Fig. 9 ChainTracker—branch visualization filtered

In the *Bank to Credit Report* example, multiple implicit bindings exist between the properties of the Customer and Credit source elements, and the properties of the Person target element (Fig. 8). The implicit bindings can be observed in OCL expressions (Listing 1, lines 29 and 32) where the properties *"cards"* and *"transactions"* are used in nested `collect` statements to compute the outstanding balance of a Customer. In Sect. 6, we present evidence on how making aware developers of implicit bindings help them to more accurately predict the impact of changes in transformation compositions.

ChainTracker includes two main filtering mechanisms to isolate elements and bindings of interest, namely *element filters* and *binding filters*. Using *element filters* (Fig. 9A), developers can select multiple metamodel elements to study all of its dependencies. Furthermore, using *binding filters* (Fig. 9B) developers can select one binding in the visualization and isolate all of its upstream- or downstream-related elements.

Figure 9 presents the *branch visualization* of the *Bank to Credit Report* case study. It portrays the result of applying a binding filter to find the symbolic origins of a generated line of code. In this case, the developer is able to see the origins of the generated *line of code 6* in *report.html*. This line is generated by the *template line 12* in the *generateHTML.mtl* module, by querying the property *"name"* of the element Person in the Report metamodel. This property is, in turn, derived by querying the properties *"cards"* and *"allowance"* of the Customer and CreditCard elements, respectively (see Listing 1, line 29).

As metamodels and transformations grow in size and complexity, so does the number of artifacts that need be represented in the ecosystem's views. As a result, our parallel-coordinate implementations can potentially become cluttered when dealing with extremely large ecosystems. We have adopted a parallel-coordinate arrangement similar to the one presented in [80]. Each categorical dimension is placed equidistant to each other and perpendicular to the x-axis. Furthermore, each category is allocated space proportional to the number of its comprising elements. We have found that for most of the ecosystems available in the literature, our current rendering parameters make visualizations scalable and easy to understand. Nevertheless, ChainTracker allows developers to manually modify the distance between dimensions in order to increase the usability of its visualizations with larger ecosystems. As suggested in [80], strategies such as the *theory of envelopes* can be used to implement automatic scaling mechanisms for parallel coordinates. We plan to include such capabilities in future releases of the tool.

3.2.2 The ChainTracker projectional code editors

It is important to mention that none of the existing approaches allow developers to project information obtained from their interaction with the transformation visualizations onto transformation editors. This fundamentally limits their usability in the context of supporting developers developing and maintaining transformation ecosystems. In [81], Myers et al. explore how conventional human–computer interaction (HCI) methods can help researchers to better understand the need of developers when dealing with complex software artifacts. Particularly, Myers et al. have found that a key use for code visualizations is to guide developers to the right code to look at, instead of being an aid to understanding on their own. Following this principle, ChainTracker enables developers to use transformation visualizations to understand their static and dynamic characteristics, as well as to project information they have discovered in the visualizations onto textual editors, and vice versa.

ChainTracker offers a code-projection menu where developers can find three types of projections, namely *downstream projections*, *upstream projections*, and *single binding projections* (Fig. 6D). In all cases, developers can select artifacts in the visualizations, in order to find their original textual representation. To understand their usage, let us briefly examine four examples.

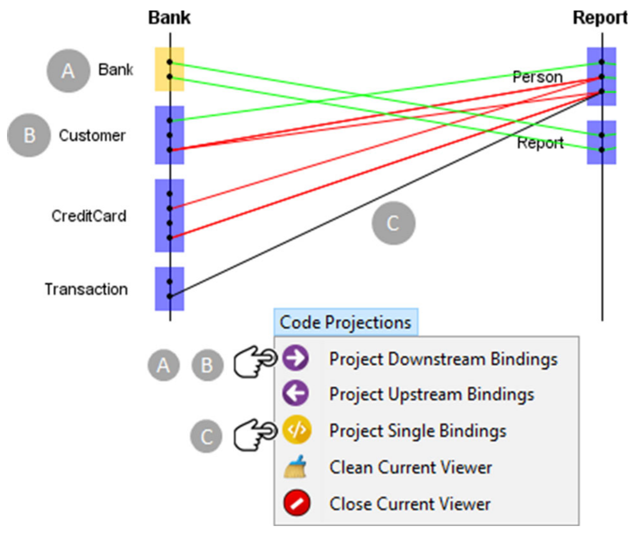
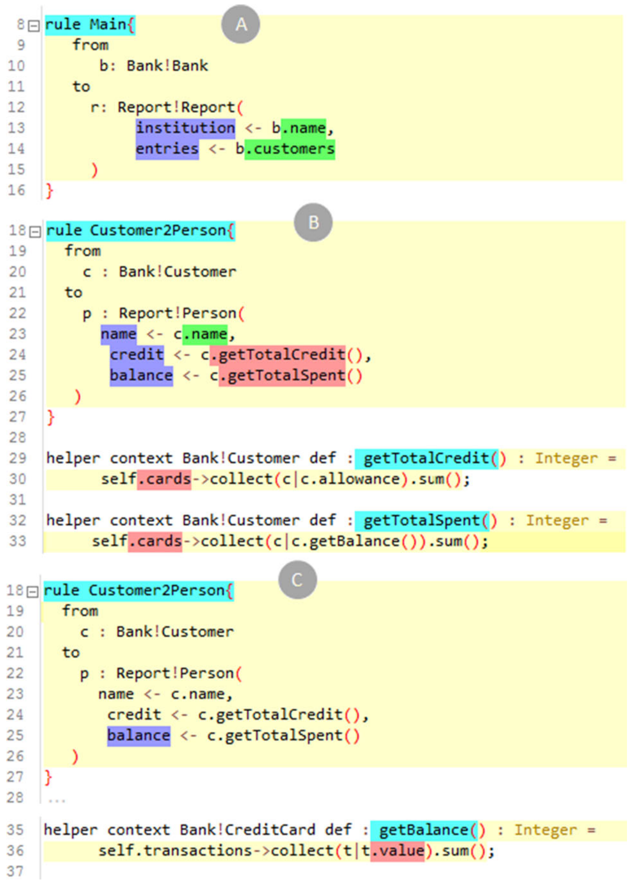


Fig. 10 ChainTracker—binding projection menu

1. Figures 10A and 11A showcase the use of *downstream projections* in order to isolate the textual representation of the bindings associated with the Bank element of the Bank metamodel. In this case, all the related bindings are located in the Bank2Report (M2M) transformation (Listing 1). Projections that affect model-to-model transformation expressions are highlighted by first locating the rule where each binding is located (yellow shadow), its name (cyan), the property targeted by the binding (blue), and its related statements (green and red for explicit and implicit bindings, respectively).
2. Figures 10 and 11B present the result of applying *downstream projections* to obtain the textual representation of bindings related to the Customer element of the Bank metamodel. A similar result will be obtained when applying *upstream projections* to elements of the Report metamodel.
3. Figures 10C and 11C present the result of applying a *single binding projection* in order to isolate the expressions that realize the binding between the *"balance"* property of the Person element, and the *"value"* property of the Transaction element. ChainTracker identifies procedural calls as implicit bindings and helps developers to follow their execution flow using code projections.
4. Figure 12 depicts the result of selecting a model-to-text binding (A), and projecting it onto the textual editor (B). In this particular case, the editor showcases the contents of the Report2HTML (M2M) transformation (Listing 2). It highlights the binding expression that uses the property *"name"* of the Person element in the Report metamodel. This expression generates a portion of the report table (C). It is important to mention that generation bindings can also be subject of code projections onto generated textual files.



```
rule Main{
   from
      b: Bank!Bank
   to
     r: Report!Report(
          institution <- b.name,
          entries <- b.customers
     )
}
```

```
rule Customer2Person{
  from
    c : Bank!Customer
  to
    p : Report!Person(
      name <- c.name,
       credit <- c.getTotalCredit(),
       balance <- c.getTotalSpent()
    )
}

helper context Bank!Customer def : getTotalCredit() : Integer =
      self.cards->collect(c|c.allowance).sum();

helper context Bank!Customer def : getTotalSpent() : Integer =
     self.cards->collect(c|c.getBalance()).sum();
```

```
rule Customer2Person{
  from
    c : Bank!Customer
  to
    p : Report!Person(
      name <- c.name,
       credit <- c.getTotalCredit(),
       balance <- c.getTotalSpent()
    )
}
...
helper context Bank!CreditCard def : getBalance() : Integer =
      self.transactions->collect(t|t.value).sum();
```

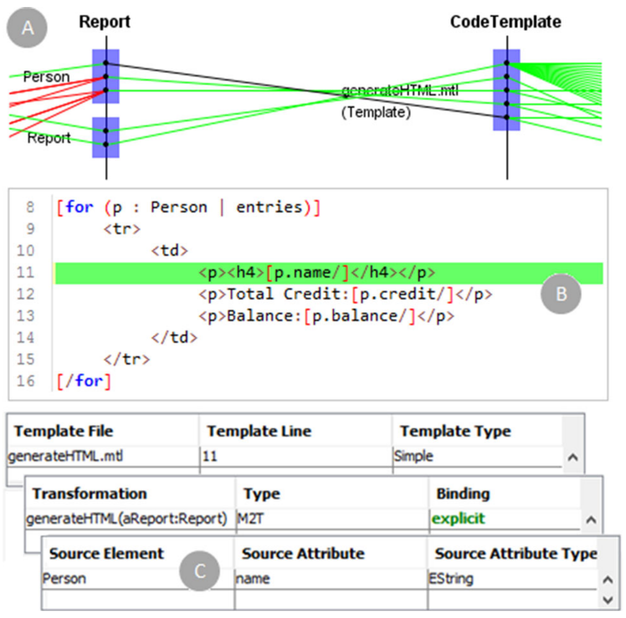
Fig. 11 ChainTracker—M2M binding projections

Code projections help developers to navigate the visualizations and map their semantics onto the expressions that they represent. They also provide a straightforward strategy to find unused code in transformation scripts, so they can be refactored or deleted (T6). In this context, a portion of a transformation script can be considered orphan if it remains clear after projecting all of its associated bindings from the branch visualization.

ChainTracker also includes a reverse code-projection feature. It enables developers to study templates and generated files in their textual form, in order to find a line of interest and investigate its precise graphical representation (Fig. 6D: Text Selection). A more elaborated example of this feature can be found in https://guana.github.io/chaintracker/tutorial.html.

3.2.3 The ChainTracker contextual tables

One of the strategies that ChainTracker uses to minimize the amount of information displayed in its visualizations is to omit details about the names and properties of the metamodel elements and binding expressions. However, this information might be of high value to developers debugging complex transformation expressions. The contextual tables present



```
[for (p : Person | entries)]
    <tr>
        <td>
            <p><h4>[p.name/]</h4></p>
            <p>Total Credit:[p.credit/]</p>
            <p>Balance:[p.balance/]</p>
        </td>
    </tr>
[/for]
```

Template File	Template Line	Template Type
generateHTML.mtl	11	Simple

Transformation	Type	Binding
generateHTML(aReport:Report)	M2T	explicit

Source Element	Source Attribute	Source Attribute Type
Person	name	EString

Fig. 12 ChainTracker—M2T binding projections

information that reveal details not available in the visualizations of the environment (Figs. 9, 12C).

Developers can select a metamodel element in the visualizations, and the three contextual tables will display information about the name and type of its properties, related upstream bindings (bindings that have as target the selected element), and related downstream bindings (bindings that use the selected element as the source for the creation of another element). Developers can also select individual bindings to obtain information about the transformation where the binding is located. When a binding of any nature is selected, the three contextual tables will portray information regarding the name of its parent script, generation module, and bound element properties. If the selected binding is a model-to-text binding, information regarding its location inside its corresponding template script will be presented along with its expression type, i.e., binding due to a simple query, conditional expression, or loop expression.

4 Study design

In this section, we summarize the design of our study, including our hypotheses, dependent and independent variables, the characteristics of the participants, and the tasks that participants completed using Eclipse Modeling and ChainTracker as tool treatments.

4.1 Hypotheses

In this study, we hypothesize that enabling developers to interactively explore the execution semantics of a transformation ecosystem can significantly improve their performance when reflecting on an ecosystem's metamodel and generation dependencies. In order to investigate our research questions, we outlined two individual null hypotheses.

- H_{01}: Developers spend an equal amount of time identifying metamodel and generation dependencies in model transformations using ChainTracker and Eclipse Modeling editors.
- H_{02}: Developers provide equally correct answers identifying metamodel and generation dependencies in model transformations using ChainTracker and Eclipse Modeling editors.

4.2 Objects

The objects of our study are two model-based code generators developed in our research laboratory: ScreenFlow and PhyDSL.

ScreenFlow is a design environment for mobile application storyboards. It enables developers to quickly translate user interface sketches into application skeletons, including interface navigation logic. ScreenFlow consists of a textual domain-specific language, and a linear model-transformation chain that includes one model-to-model transformation, and two model-to-text transformations written in ATL and Acceleo, respectively (Fig. 13). ScreenFlow is designed for novice android application developers and for rapid software prototyping environments, such as hackathons. A complete description and demo video of ScreenFlow can be found at https://guana.github.io/screenflow.

PhyDSL [82,83] is a game engine and authoring environment for mobile 2D physics-based games. It consists of a textual domain-specific language for gameplay design, and a multibranched transformation chain that takes high-level gameplay specifications and translates them into executable code for mobile devices. PhyDSL's transformation chain includes four model-to-model transformations implemented using ATL, and four template-based model-to-text transformations written in Acceleo (Fig. 14). PhyDSL is currently used by the Faculty of Rehabilitation Medicine at the University of Alberta, the Knowledge Media Design Institute at the University of Toronto, and the Sapporo Medical Uni-

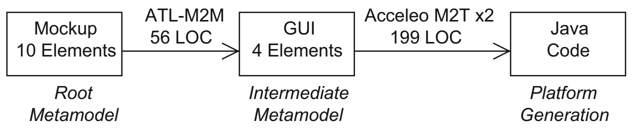


Fig. 13 ScreenFlow's linear model-transformation chain: 1 model-to-model (M2M) ATL transformation comprised by 4 matched rules and 1 helper, and 2 model-to-text (M2T) Acceleo transformations containing 43 binding expressions

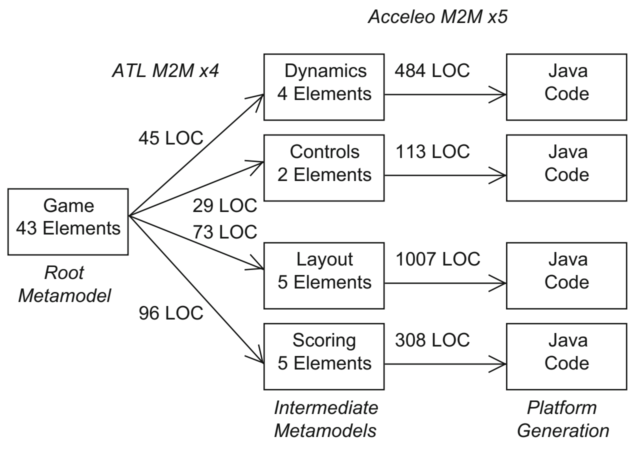


Fig. 14 PhyDSL's multibranched model-transformation chain: 4 model-to-model (M2M) ATL transformations comprised by 14 matched rules and 2 helpers, and 4 model-to-text (M2T) Acceleo transformations containing 172 binding expressions

versity in Japan, to create cost-effective mobile games for rehabilitation therapy [84]. More information about PhyDSL can be found at https://guana.github.io/phydsl/. The PhyDSL and ScreenFlow source code, and corresponding Chain-Tracker visualizations can be found at https://github.com/guana/chaintracker-eval.

4.3 Dependent variables

Considering the hypotheses H_{01} and H_{02}, our experiment has two dependent variables on which our treatments are compared:

- Var_A: Time developers spend solving each task.
- Var_B: Developers' accuracy solving each task.

4.4 Independent variables

The four independent variables of this study are summarized in Table 2. Variables $VarCT_1$ and $VarEM_1$ represent the tasks designed to evaluate the performance of developers working in the context of ScreenFlow, and $VarCT_2$ and $VarEM_2$ in the context of PhyDSL, using ChainTracker (CT) and Eclipse Modeling (EM), respectively.

4.5 Tasks

In this study, we measured the performance of developers when asked to identify dependency relationships between artifacts in two transformation ecosystems of different complexity. In order to do so, we created a set of task templates that aim at understanding how developers identify dependency relationships at different levels of granularity, using different tool treatments. These tasks are grouped in five main

Table 2 Study independent variables

Object system	Tasks CT	Tasks EM
Object 1: ScreenFlow	$VarCT_1$	$VarEM_1$
Object 2: PhyDSL	$VarCT_2$	$VarEM_2$

families, namely, a) determining metamodel coverage and expression location, b) identifying metamodel dependencies in model-to-model transformations, c) identifying metamodel dependencies in model-to-text transformations, d) identifying generation dependencies in model-to-text transformations, and e) identifying generation dependencies in model transformation chains (Sects. 4.5.1–4.5.5). The proposed question templates can be used to replicate this study with different object systems. "Appendix A" presents the questionnaires that instantiate the proposed templates in the context of PhyDSL and ScreenFlow. Let us briefly present each family of tasks.

4.5.1 Determining metamodel coverage and expression location

The goal of this family of tasks is to investigate how developers identify the major components of a transformation ecosystem, and measure their performance when inferring its high-level compositional structure. Tasks in this family use the following templates:

- Are there any unused elements in the *[metamodel-name]* metamodel? if so which ones?
- What transformation rule contains the binding expression *[query-expression]*?
- What transformation script contains the *[rule-name]* rule?
- What files does the *[template-name]* template generate?

4.5.2 Identifying metamodel dependencies in M2M transformations

This set of tasks requires developers to identify the dependencies that exist between the source and target metamodels of a single model-to-model transformation. This family of tasks is divided in two categories, namely element- and property-level dependency tasks.

More often than not, property-level dependencies are localized in nontrivial binding expressions that realize the intent of a transformation script, these include metamodel navigation statements or procedural calls to helpers. Tasks in this family are also distinguished by the direction of the dependencies that need to be identified. While some tasks require developers to identify upstream dependencies, other tasks investigate their performance identifying downstream dependencies. Tasks in this family are specified using the following templates:

- What metamodel elements are used in the creation of the *[metamodel-name ! element-name]* element? (element upstream dependencies)
- What metamodel elements are created using the *[metamodel-name ! element-name]* element? (element downstream dependencies)
- What metamodel elements are created using the property *[property-name]* of the *[metamodel-name ! element-name]* element? (property downstream dependencies)

4.5.3 Identifying metamodel dependencies in M2T transformations

This family of tasks investigates how developers identify upstream dependencies in model-to-text transformations. Concretely, they ask developers to determine the metamodel elements required for the execution of one or multiple bindings expressions in a model-to-text transformation. Furthermore, they ask developers to evaluate whether such elements have upstream dependencies in model-to-model transformations that are potentially linked in previous steps of a transformation chain. Tasks in this family follow the template:

- Considering the entire transformation chain, what metamodel elements does the template line *[line-number]* in *[template-name]* depend on?

4.5.4 Identifying generation dependencies in M2T transformations

This set of questions requires developers to identify dependencies between generated textual artifacts, e.g., code, and their originating model-to-text transformations. They are presented to the participants using the following template:

- What template lines in *[template-name]* are used in the generation of line *[line-number]* in *[generated-file-name]*?

4.5.5 Identifying generation dependencies in MTCs

This collection of tasks investigate how developers identify the dependencies of a generated textual artifact in a holistic fashion. They ask developers to determine all the metamodel elements and properties that intervene in the generation of one or multiple lines in a generated textual artifact. They are also divided in two categories namely, element- and property-level dependency tasks. The tasks in this family use the following templates:

- Considering the entire transformation chain, what metamodel elements does the generation of line *[line-number]* in *[generated-file-name]* depend on?
- Considering the entire transformation chain, what metamodel properties does the generation of line *[line-number]* in *[generated-file-name]* depend on?

4.6 Detailed hypotheses

Taking into account our two high-level null hypotheses and our two object systems, this study tries to reject four detailed hypothesis:

$$H_0Var_{A1} : \tilde{V}ar_A CT_1 = \tilde{V}ar_A EE_1$$
$$H_0Var_{A2} : \tilde{V}ar_A CT_2 = \tilde{V}ar_A EE_2$$
$$H_0Var_{B1} : \tilde{V}ar_B CT_1 = \tilde{V}ar_B EE_1$$
$$H_0Var_{B2} : \tilde{V}ar_B CT_2 = \tilde{V}ar_B EE_2$$

Hypotheses H_0Var_{A1} and H_0Var_{A2} compare the median time spent by developers solving the proposed tasks in single and multibranched transformation chains, respectively *(i.e., developers spend an equal amount of time solving questions using ChainTracker and Eclipse editors for single and multibranched transformation chains)*. Moreover, hypotheses H_0Var_{B1} and H_0Var_{B2} compare the median accuracy (in terms of task solution correctness) of developers solving the proposed tasks on single and multibranched transformation chains, respectively *(i.e., developers provide equally correct answers using ChainTracker than they do using Eclipse editors for single and multibranched transformation chains).*

4.7 Participants

This study involved 25 software engineers with an average of 7.5 years of software development experience. All participants were enrolled in a professional masters program which includes an intensive course on software engineering automation using model-transformation languages. The participants had an average of 6 months of training in rule- and template-based model-transformation languages. Their training also included Eclipse Modeling as their main development environment for model transformations. All of the participants reported having used Eclipse in their professional development practice, and Eclipse Modeling in the context of their graduate course in model-transformation technologies. None of the participants had experience with model-transformation technologies in an industrial setting. Furthermore, none of the participants had previous knowledge of the case studies used in this study. Our pool of participants is representative of a community in which developers have only introductory training on model-transformation technologies, yet considerable experience in the general software engineering field.

4.8 Data analysis

Given the sample size and the non-normal distribution of the data collected in this study, we adopted a Mann–Whitney "U" nonparametric test to investigate our hypothesis propositions. The Mann–Whitney compares the median of the observations for datasets with pronounced outliers. This makes the test a suitable analysis tool for small unpaired datasets with skewed distributions. All of our hypotheses were evaluated using a two-tailed version of test. In this study, we consider an alpha level with a *p* value lower than 5%; thus, we consider an acceptable probability of 0.05 for Type-I error, i.e., rejecting the null hypothesis when it is true.

5 Protocol

The protocol of the study was divided in three sessions conducted in the course of one week (Fig. 15). Session 1 (training session) involved an introductory tutorial on the features of ChainTracker. Sessions 2 and 3 involved two independent working sessions in which 25 participants solved the tasks assigned for the object systems of the study.

5.1 Training

The training session was structured in two 60 minute parts divided by a 15 minute break. The first half of the session involved a tutorial on the use of ChainTracker. The second half consisted of a laboratory workshop that provided hands on experience with the analysis environment.

The 25 participants received a presentation that introduced a case study of similar complexity to the one used in Sect. 3.1. The case study was used to explore the features of ChainTracker and the graphical notation of its visualizations. Additionally, participants completed a worksheet with questions conforming to the templates described in Sect. 4.5. Two research assistants helped participants to solve questions as they completed the guide. The training material and case study can be found at https://guana.github.io/chaintracker/tutorial.html.

At the end of the first session, the 25 participants were randomly divided in two groups of 15 and 10 participants, namely, Group A and B, respectively. Group A was assigned to the working session A, and Group B to the working session B.

5.2 Working sessions

Participants assigned to Group A were randomly divided in two subgroups of 7 and 8 participants, namely Group A1 and A2, respectively. Similarly, participants in Group B were randomly divided in two subgroups of 5 participants, namely Group B1 and B2.

Each subgroup was assigned to individual computer laboratories with virtual machines containing ChainTracker and Eclipse Modeling with ATL and Acceleo plug-ins. Participants in Working Sessions A and B were assigned ScreenFlow and PhyDSL as their case studies, respectively.

At the beginning of each session, participants received a 15-minute presentation on the major components of their corresponding case studies. This presentation included 10 minutes of questions and final setup. All participants received a printed copy of the metamodels that comprise their assigned case study. Finally, the participants in Groups A1 and B1 were instructed to solve 25 tasks using ChainTracker, and participants in Groups A2 and B2 using Eclipse Modeling ("Appendix A").

Both working sessions had a maximum time restriction of 2.5 hours. Each computer laboratory was supervised by a research assistant who was available to answer high-level questions about both Eclipse and ChainTracker, as well as the setup of the virtual machines. At the end of both working sessions, participants were rewarded with a gift card equivalent to $25CAD for their time.

5.3 Data collection

The working sessions were instrumented with a survey application developed by our research team https://www.github.com/guana/surveygen. Each participant logged into the survey application using a unique ID assigned before the beginning of the session. The application presents one task

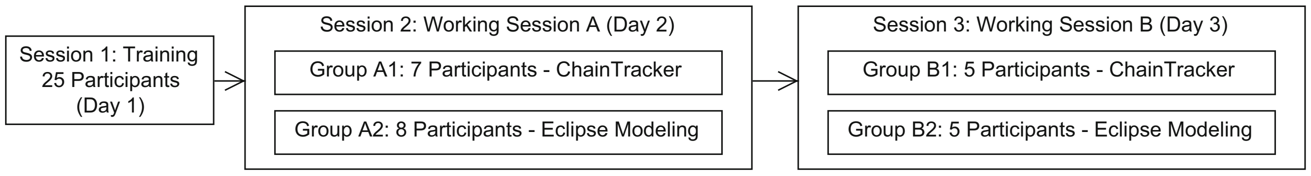

Session 1: Training
25 Participants
(Day 1)
Session 2: Working Session A (Day 2)
Group A1: 7 Participants - ChainTracker
Group A2: 8 Participants - Eclipse Modeling
Session 3: Working Session B (Day 3)
Group B1: 5 Participants - ChainTracker
Group B2: 5 Participants - Eclipse Modeling

Fig. 15 Study protocol: Session 1—Training, Session 2—Working Session A, and Session 3—Working Session B

at the time until all tasks in the questionnaire are answered. The application does not allow participants to skip tasks. Furthermore, it does not allow participants to return to a task once it has been completed. Our survey application measures the total time spent by the participant in each task. The total time is calculated as the time between a task is presented to the participant, and the time a valid answer is submitted.

The survey application is capable of presenting three types of questions, namely, multiple choice questions, list-based questions, and multiple selection questions. Each type captures the participant's answers using different mechanisms. Multiple choice questions receive one answer from a predefined set of options. List-based questions receive one or multiple open-ended answers, and multiple selection questions receive one or multiple answers from a predefined set of options. Our survey application stores the results of a session in a remote server using a REST API.

It is important to mention that tasks in each questionnaire were organized to make the experience of the participants engaging and balanced throughout the entire session. Similar tasks (i.e., tasks belonging in the same family) were distributed throughout the questionnaires, in almost regular intervals, and their level of difficulty was considered to avoid having collections of consecutive task with disproportionate complexity level. We categorized the difficulty of the each task in three increasing levels: easy, medium, and hard based on our observations during the preliminary pilot studies. The questionnaires took this classification into account, and they do not present more than two hard questions or more than three medium questions, one after the other. Since participants only use one tool treatment throughout the course of the study, counterbalancing was not sought.

In order to quantify the developers' accuracy for each task, we designed a point-based scoring mechanism similar to the one used [85] and [86]. Each task in the questionnaire is scored individually. For multiple choice questions, each participant is given a single point if the selected answer is correct. In the case of multiple selection questions, one point is given for each correct option selected. Furthermore, half a point is taken for every incorrect option selected, as well as for every correct option missed. Similarly, for list-based questions one point is given for correct answers, and half a point is taken for incorrect or missed options.

6 Results

In this section, we present the results of our study. We use summary tables to present the performance of developers in each family of tasks ("Appendix B").

6.1 Determining metamodel coverage and expression location

The goal of this family of tasks is to investigate the performance of developers identifying the major components of a transformation ecosystem, and the high-level compositional structure of its underlying transformations. Concretely, these tasks require developers a) to identify the files generated by a given model-to-text transformation, b) to locate individual binding expressions in the transformations of an ecosystem, and c) to evaluate the coverage of an ecosystem's metamodels. Tables 5 and 6 ("Appendix A") summarize the results for this family of tasks.

When identifying the files generated by a given model-to-text transformation in ScreenFlow (Q9 and Q10, Table 5), developers were on average 63% more efficient with Eclipse than with ChainTracker. This effect is less pronounced (22%) in the case of developers working on PhyDSL (Q9 and Q10, Table 6). However, there is no substantial difference regarding the accuracy of both groups of developers.

When asked to isolate individual binding expressions in ScreenFlow (Q1 and Q2, Table 5), developers supported by Eclipse were on average 29% more efficient than those using ChainTracker. In the context of PhyDSL, no performance difference was observed. Furthermore, there is no difference regarding the accuracy of developers in the ecosystems under study.

Finally, for the tasks of metamodel coverage assessment, developers using ChainTracker were on average 46% more efficient and 72% more accurate that those using Eclipse in ScreenFlow (Q17 and Q18, Table 5). On cursory examination, developers using Eclipse seem to be more efficient assessing metamodel coverage in PhyDSL (Q17 and Q18, Table 6). However, considering their lower accuracy, we believe this group of developers only submitted partial answers. Indeed, participants using ChainTracker were on average 200% more accurate assessing the coverage of metamodels in PhyDSL.

There is no statistically significant evidence to reject any of our hypothesis propositions for developers completing this family of tasks.

6.2 Identifying metamodel dependencies in M2M transformations

This set of tasks investigates the performance of developers identifying the upstream and downstream dependencies between a collection of metamodels, in the context of individual model-to-model transformations. The results for this family of tasks are divided in two levels of granularity, namely identifying *element-level dependencies* (Tables 7, 8), and identifying *property-level dependencies* (Tables 9, 10, "Appendix A").

ChainTracker improved the accuracy of developers assessing *element-level dependencies* in the context of individual model-to-model transformations (Q3, Q4, Q11, and Q12, Tables 7, 10). Developers using ChainTracker were on average 83% more accurate, and 48% more efficient than those using Eclipse, in both ecosystems under analysis.

Considering our accuracy-related hypotheses, we can reject H_0Var_{B1} (Q11: $p = 0.0310$, and Q12: $p = 0.0006$), and H_0Var_{B2} (Q11: $p = 0.0310$, Q4: 0.0072, Q3: $p = 0.0065$, and Q12: $p = 0.0065$) for developers identifying *element-level dependencies*. We can also reject our efficiency-related hypothesis H_0Var_{A1} (Q11: $p = 0.0021$, Q4: $p = 0.0139$, and Q12: $p = 0.0012$) in the case of developers working on ScreenFlow. Indeed, participants using ChainTracker were on average 36% faster than those supported by Eclipse in PhyDSL. However, there is not statistically significant evidence to reject H_0Var_{A2}.

Finally, participants using ChainTracker were significantly more accurate than those using Eclipse when determining *property-level dependencies* (Q23 and Q13, Tables 9, 10), almost 9 times more accurate in fact. We have statistically significant evidence to reject our accuracy-related hypotheses, namely H_0Var_{B1} (Q23: $p = 0.0046$ and Q13: $p = 0.0491$) and H_0Var_{B2} (Q23 $p = 0.0412$ and Q13 $p = 0.0393$) for developers identifying *property-level dependencies*.

6.3 Identifying metamodel dependencies in M2T transformations

This family of tasks requires developers to identify the metamodel elements consumed by one, or multiple, binding expressions in a model-to-text transformation, and to evaluate whether these elements have upstream dependencies in a model-transformation chain. Tables 11 and 12 ("Appendix A") summarize the relevant results.

In the context of ScreenFlow, developers using ChainTracker were on average 55% more accurate than those using Eclipse (Q5, Q15, and Q22, Table 11). In the case of PhyDSL, the ChainTracker advantage is even more pronounced. Developers using ChainTracker were 90% more accurate than those using Eclipse (Table 12).

We found statistically significant evidence to reject our accuracy-related hypotheses H_0Var_{B1} (Q5: $p = 0.0119$ and Q22: $p = 0.0025$), and H_0Var_{B2} (Q5: $p = 0.0072$, Q15: $p = 0.0097$ and Q22: $p = 0.0117$) for developers identifying metamodel dependencies in model-to-text transformations. There is no significant difference in the efficiency of developers completing this family of tasks.

6.4 Identifying generation dependencies in M2T transformations

This set of tasks requires developers to identify dependencies between individual lines of text, e.g., code, and their originating model-to-text transformations. Tables 13 and 14 ("Appendix A") summarize the relevant results.

For this family of tasks, developers using ChainTracker were on average 19% more efficient and 94% more accurate than those using Eclipse (Q6, Q16, and Q25). Although this difference is statistically significant for Q25 in ScreenFlow, and Q6 in PhyDSL, there is no significant evidence to reject any of our hypothesis propositions for developers completing this family of tasks.

6.5 Identifying generation dependencies in MTCs

This collection of tasks investigates how developers identify the upstream dependencies of a generated textual artifact, i.e., the metamodel elements needed for the generation of one or multiple lines of code. It is important to mention that this family of tasks inquires about the metamodel dependencies throughout an entire transformation chain. The performance measurements for this family of tasks are also divided in two levels of granularity, namely identifying *element-level dependencies* (Tables 15, 16), and identifying *property-level dependencies* (Tables 17, 18, "Appendix A").

When identifying *element-level dependencies* in ScreenFlow, developers using ChainTracker were on average 62% more accurate than developers using Eclipse. This effect is significantly more pronounced in PhyDSL where developers were 100% more accurate (Q7, Q8, Q19, Q20, and Q21, Tables 15, 16). This observation is statistically significant for all the tasks under consideration.

In ScreenFlow, developers using ChainTracker were 66% more efficient than those using Eclipse. This finding is statistically significant for all tasks in the ScreenFlow ecosystem. In PhyDSL, we observed that even though the median time spent by developers in these tasks is lower for developers using ChainTracker, the difference is significant only in Q8.

We found statistically significant evidence to reject our accuracy-related hypotheses, namely H_0Var_{B1} (Q7: $p = 0.0032$, Q8: $p = 0.0073$, Q19: $p = 0.0443$, Q20: $p = 0.0007$, and Q21: $p = 0.0007$), and H_0Var_{B2} (Q7: $p = 0.0094$, Q8: $p = 0.0072$, Q19: $p = 0.0066$, Q20: $p = 0.0055$, and Q21: $p = 0.0055$) for developers identifying end-to-end *element-level dependencies*. Furthermore, we can reject our efficiency-related hypothesis H_0Var_{A1} (Q7: $p = 0.0093$, Q8 $p = 0.0012$, Q19: $p = 0.0037$, Q20: $p = 0.0003$, and Q21: $p = 0.0003$).

As we mentioned in Sect. 3, identifying end-to-end *property-level dependencies* in a transformation chain requires developers to find all the metamodel element properties that are explicitly or implicitly used in the generation of a given line of code. When tracing *property-level dependencies*, ChainTracker developers were on average 68% more accurate than those using Eclipse in both ecosystems under study (Q14, and Q24, Tables 17 and 18).

Similar to element-level dependencies, we observed that identifying end-to-end *property-level dependencies*, in both linear and multibranched transformation chains is a challenging task. In this context, participants using Eclipse obtained a median accuracy score of 0.5 and 1.0 for tasks with maximum attainable scores of 8.0 (Q14 in ScreenFlow) and 4.0 (Q14 in PhyDSL), respectively.

Indeed, we can reject our accuracy-related hypotheses, namely H_0Var_{B1} (Q14: $p = 0.0090$, and Q24: $p = 0.04871$), and H_0Var_{B1} (Q14: $p = 0.0066$ and Q24: $p = 0.0248$) for developers identifying end-to-end *property-level dependencies*.

With respect to our efficiency-related hypothesis, we found statistically significant evidence that ChainTracker outperforms Eclipse Modeling in helping developers analyzing end-to-end *property-level dependencies*. However, we believe that most of the time, observations cannot be fairly analyzed considering that most developers using Eclipse were highly inaccurate. We require further analysis and additional empirical information to validate our hypotheses for developers completing this type of tasks.

7 Discussion

7.1 Determining metamodel coverage and expression location

Developers using ChainTracker performed less efficiently than developers using Eclipse in locating individual binding expressions across the ecosystems under study. At the same time, ChainTracker developers were considerably more accurate and more efficient when determining the coverage of their metamodels. None of these results, however, are statistically significant.

Tasks Q1 and Q2 require developers to isolate individual binding expressions in the ecosystems under study. Furthermore, Tasks Q9 and Q10 require developers to identify the files generated by a given model-to-text transformation (Tables 3, 4). We noticed that developers using Eclipse relied on the pattern-matching features of the environment to complete these tasks. As shown in Tables 5 and 6, this strategy proved very effective. Eclipse developers are between 24% and 51% more efficient than those supported by ChainTracker in Q1 and Q2, and between 22% and 63% more efficient in Q9 and Q10. As ChainTracker does not offer pattern-matching capabilities, developers had to manually explore the transformations of the ecosystems to complete the aforementioned tasks.

Tasks Q17 and Q18 require developers to determine the coverage of two metamodels in their respective ecosystems, namely Mockup and GUI for ScreenFlow, and PhyDSL and Dynamics for PhyDSL (Tables 3, 4). Eclipse developers had to manually explore all the upstream and downstream bindings that operate on each metamodel to evaluate its coverage. In ScreenFlow, they had to examine 56 lines of code, corresponding to its single model-to-model transformation (Fig. 13). In PhyDSL, they had to study a total of 243 lines of code, corresponding to its four model-to-model transformations (Fig. 14). Moreover, the Mockup and GUI metamodels of ScreenFlow have a total of 14 metamodel elements, and the PhyDSL and Dynamics metamodels of PhyDSL have 47 elements. Indeed, manually examining large transformation codebases, looking for the usage of dozens of metamodel elements, is a daunting, if not impossible, task.

Participants using ChainTracker were not required to manually study transformation codebases to identify uncovered metamodel elements. Coverage information is presented in the *overview visualization* using pie charts that summarize the in- and out-coverage metrics for all of the metamodels of an ecosystem. Furthermore, using ChainTracker's *branch visualization*, developers were able to identify metamodel elements that have no bindings attached to them, which makes uncovered elements easy to identify (Fig. 8). This information can be effectively used to precisely remove unused metamodel elements and properties, and to identify portions of code that are never executed.

7.2 Identifying metamodel dependencies in M2M transformations

7.2.1 Element-level dependencies

Developers using ChainTracker are significantly more accurate and more efficient than those using Eclipse at identifying *element-level dependencies* in model-to-model transformations. In effect, identifying downstream dependencies is significantly more difficult to developers, than pinpointing their upstream counterparts.

We believe that most of the positive impact that ChainTracker had on the accuracy of developers stems from the usability of its *branch visualization*, which presents a unified view of the traceability links contained in the transformations of an ecosystem.

Identifying downstream dependencies (Q4 and Q12) in rule-based transformations is fundamentally more difficult than identifying upstream dependencies (Q2 and Q11, Tables 3, 4). To complete both sets of tasks, developers need to examine the transformations and interpret their execution semantics. In order to identify upstream dependencies, developers only need to examine the binding expressions located in the matched rule corresponding to the element of interest. These expressions determine the metamodel elements that are used for its creation. In contrast, developers looking for an element's downstream dependencies require to pinpoint all the binding expressions located in potentially multiple transformation rules, used to create an undetermined number of target elements. In both cases, developers need to consider all the implicit and explicit expressions that use the element of interest to either constrain their execution, or gain access to other metamodel elements.

ChainTracker interactive filtering capabilities proved effective in isolating information corresponding to individual metamodel elements, and their related binding expressions. This enabled developers to analyze the downstream and upstream dependencies despite of the size of the visualizations. Furthermore, ChainTracker enables developers to quickly identify implicit and explicit bindings in complex OCL expressions. Our empirical observations revealed that the automatic reasoning of model transformations, and the quick access to fine-grained traceability links, enabled developers to identify dependency relationships in a more efficient and accurate manner. As shown in Tables 7 and 8, developers using Eclipse had a very low accuracy in tasks that included the analysis of expressions with multiple implicit bindings (Q4, and Q12).

7.2.2 Property-level dependencies

Identifying *property-level dependencies* in model-to-model transformation is particularly difficult due to the large search spaces that need to be explored to complete this task. ChainTracker developers were significantly more accurate in identifying *property-level dependencies* than participants using Eclipse. This phenomenon is more pronounced as the complexity of the ecosystem increases.

Similarly to tasks that require developers to identify downstream element dependencies, identifying *property-level dependencies* requires developers to manually interpret the binding expressions of all the transformations in an ecosystem. We observed that Eclipse users used its pattern-matching capabilities to find all the expressions that used a property of interest. This strategy is somehow effective in the case of properties with very distinctive names, such as in the case of *"fromScreen"* in Q23 (Table 3). However, for properties with common names, such as *"value"* or *"name"* (Q23 and Q13 in Table 4) the text-based search approach proved ineffective.

7.3 Identifying metamodel dependencies in M2T transformations

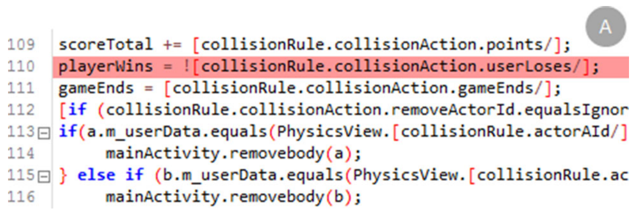
Developers using ChainTracker were significantly more accurate than those using Eclipse in identifying upstream metamodel dependencies in model-to-text transformations. This effect is more pronounced in PhyDSL, which suggests that the complexity of these tasks, and the usefulness of ChainTracker, increases in multibranched transformation chains.

Developers using Eclipse appeared to be more efficient than those using ChainTracker when addressing Q5, Q15, and Q22. However, they were much less accurate. Due to the small set of participants in each working session, we are unable to isolate developers that use Eclipse and that obtained high accuracy scores, in order to make a fair statistical analysis of our time-dependent hypotheses.

It is important to note that developers using ChainTracker obtained a perfect accuracy score for all task in this family. Conversely, Eclipse developers had a sparse accuracy distribution (Table 11). This suggests that Eclipse developers struggle to identify upstream dependencies in multistep transformation chains, even when these are due to simple binding expressions, e.g., Q15 in ScreenFlow.

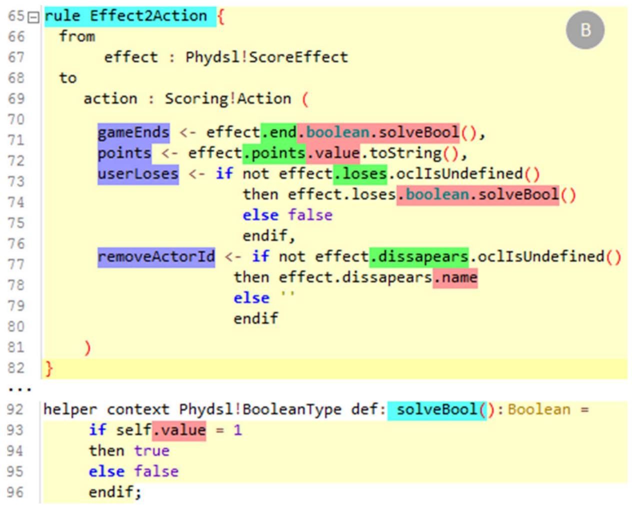
As a concrete example of the challenges that developers face completing this family of tasks, let us briefly explore Q15 in the context of PhyDSL (Table 4). This task requires developers to find the element dependencies of the expressions located in *line 110* of the *generateScoring.mtl* transformation (Fig. 16A). Furthermore, developers need to determine whether there are additional metamodel dependencies due to potential upstream model-to-model transformations in PhyDSL (Fig. 17B).

Figure 18 presents the ChainTracker visualization, relevant to task Q15 for PhyDSL. Line 110 in *generateScoring.mtl* uses two attributes corresponding to two elements



```
109  scoreTotal += [collisionRule.collisionAction.points/];
110  playerWins = ![collisionRule.collisionAction.userLoses/];
111  gameEnds = [collisionRule.collisionAction.gameEnds/];
112  [if (collisionRule.collisionAction.removeActorId.equalsIgnor
113  if(a.m_userData.equals(PhysicsView.[collisionRule.actorAId/]
114      mainActivity.removebody(a);
115  } else if (b.m_userData.equals(PhysicsView.[collisionRule.ac
116      mainActivity.removebody(b);
```

Fig. 16 PhyDSL—generateScoring.mtl M2T



```
rule Effect2Action {
  from
      effect : Phydsl!ScoreEffect
  to
    action : Scoring!Action (

      gameEnds <- effect.end.boolean.solveBool(),
      points <- effect.points.value.toString(),
      userLoses <- if not effect.loses.oclIsUndefined()
                   then effect.loses.boolean.solveBool()
                   else false
                   endif,
      removeActorId <- if not effect.dissapears.oclIsUndefined()
                  then effect.dissapears.name
                  else ''
                  endif
    )
}
...
helper context Phydsl!BooleanType def: solveBool():Boolean =
    if self.value = 1
    then true
    else false
    endif;
```

Fig. 17 PhyDSL—`Effect2Action` and `solveBool()` M2M rules

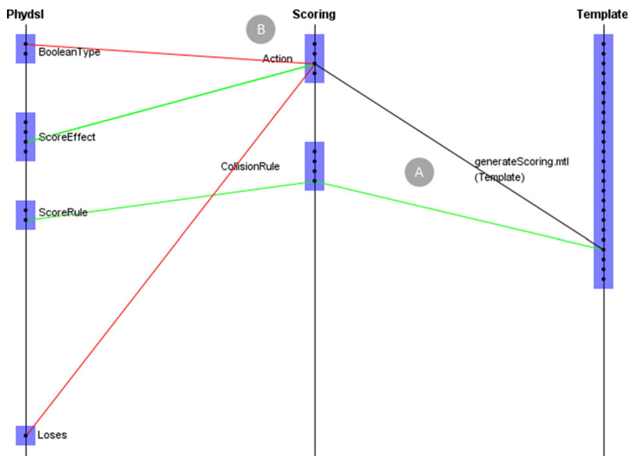


Fig. 18 PhyDSL—Task Q15 ChainTracker Visualization

in the Scoring metamodel, namely Action and CollisionRule (Fig. 18A). This line generates a portion of code that implements the scoring mechanisms of the video games generated by PhyDSL. The Action element is created by the `Effect2Action` rule (Fig. 17). This rule uses the `solveBool()` helper to complete its transformation intent. As shown in Fig. 18B, the Action element has multiple implicit dependencies given by binding expressions that include procedural calls, and metamodel navigation statements. Due to the complexity of the binding expressions, manually locating the usage of the metamodel of interest, and interpreting their corresponding dependency relationships are a challenging task.

Our observations during the study suggest that ChainTracker's *branch visualization*, and its code-projection capabilities, enabled developers to gain end-to-end traceability information, and to quickly filter upstream metamodel dependencies in both ecosystems under analysis.

7.4 Identifying generation dependencies in M2T transformations

We observed that participants using Eclipse use the Acceleo Profiler [61] in order to find generation dependencies in the model-to-text transformations under study. The Acceleo Profiler enables developers to debug model-to-text transformations, and to identify the binding expressions responsible for the generation of a particular line of code. On the other hand, ChainTracker developers were able to select portions of generated code and obtain their corresponding generation dependencies using its reverse code projection capabilities. The projections not only highlighted the model-to-text bindings that originated the selected portions of code, but provided quick access to the metamodel elements and properties, used for their generation.

ChainTracker developers were on average 19% more efficient identifying generation dependencies in model-to-text transformations. We believe this is because reverse code projections do not require to interactively execute transformations to study their bindings. Instead, they offer a self-contained view that can be studied as a whole, regardless of the complexity of the transformations under analysis, or the number of steps required for their execution. Our observations during the study suggest that interactive traceability visualizations are as useful as off-the-shelf transformation debuggers to identify generation dependencies in model-to-text transformations.

7.5 Identifying generation dependencies in MTCs

ChainTracker developers were significantly more accurate and more efficient at identifying end-to-end generation dependencies. Fifty percent of Eclipse developers obtained accuracy scores below their observed medians, which in most cases was 50% less of the maximum attainable score.

ChainTracker provides a significant advantage in the accuracy of developers interpreting binding expressions in both model-to-model and model-to-text transformations. The average median difference with respect to the accuracy of developers is more pronounced for those working on PhyDSL. Our results suggest that the benefits of using traceability visualization techniques are magnified in scenarios where the ecosystems are composed in nontrivial transformation chains. It is important to mention that all developers using ChainTracker had a perfect accuracy score when identifying *element-level generation dependencies* (Tables 15, 16), and *property-level generation dependencies* (Tables 17, 18).

We believe that the effectiveness of developers using ChainTracker to identify multistep generation dependencies is due to the quick access that they have to implicit and explicit binding information. We found that not only the *branch visualization* is highly useful to developers, but also the active use of editors that allow the projective interactions between graphical representations and transformation codebases.

8 Threats to validity

Construct validity *(Do we measure what is intended?)* In this study, we measured the performance of developers identifying dependency relationships in model-to-model and model-to-text transformations. We compared two tools namely, ChainTracker and Eclipse Modeling. We used two model-based code generators of different complexity to investigate the effect of their size on the usability of ChainTracker. Moreover, we understand developers' performance in terms of the time they take completing each task and their solution correctness.

We have developed an in-house survey application that presents participants with tasks that reflect on their ability to identify metamodel and generation dependencies at different levels of granularity, and between different artifacts of a transformation ecosystem. Our survey application has been extensively tested and did not present any failures during the execution of the study. Furthermore, we have carefully instantiated our question templates in the context of the two case studies under consideration. The correctness of each expected solution has been validated by three model-transformation experts with 6, 1, and 2 years of experience in model-transformation technologies, respectively. The questions were designed in an iterative fashion, looking for representative tasks of different complexity inside each of the case studies.

An early version of the protocol used for this empirical study was presented in the 1st International Workshop on Human Factors in Modeling collocated at the MODELS conference in 2015 [87]. The protocol of the study and the proposed task templates were discussed and reformulated based on feedback gained in informal meetings with industry and academic practitioners. We chose Eclipse Modeling as an industry baseline given that it is the official, and most popular, development environment for both ATL and Acceleo technologies. None of the alternative traceability visualization and collection frameworks reviewed in Sect. 2 are available to the public. We do not see any significant threats to the construct validity of this study.

Internal validity *(Are there unknown factors which might affect the outcome of the experiments?)* The limited number of participants and their heterogeneous expertise on model-driven development technologies are concern for the internal validity of the study. However, this study was conducted with a pool of participants with a broad industrial development experience, and an intensive 6-month course in model transformation technologies. Considering that model-transformation languages are a fairly new technology and are yet to be adopted by the software engineering community at large, our pool of participants is representative of most model-driven engineering practitioners.

We are aware that the learning curve of ChainTracker and Eclipse Modeling may impact the developers performance. In order to minimize the impact of this threat to validity, we included an introductory tutorial in the training session of our protocol (Sect. 5.1). The first half of the training session involved a tutorial on the use of ChainTracker. The second half consisted of a laboratory workshop that provided hands on experience with the analysis environment. The training session was structured in two 60-minute parts divided by a 15-minute break. Our protocol did not include an introductory tutorial on Eclipse Modeling. In effect, all participants received 6 months of training in Eclipse Modeling as a part of their graduate course, which included by weekly hands on tutorials. Furthermore, all participants reported having used Eclipse as a development environment in professional and academic settings.

Participants were not allowed to return to a task once it was completed. This strategy might fail to account for the exploratory nature of model-transformation comprehension. Developers might want to review previously answered questions based on understanding gained throughout the course of the working sessions. Limiting our survey application to a strictly linear answering mechanism was motivated by our desire to precisely measure the efficiency of developers solving individual tasks. We minimize this threat to validity in two ways. First, each of our working sessions includes a 15-minute presentation that explores the metamodels and transformations of their corresponding case studies. Additionally, a 10-minute window was allocated to allow further discussion on the implementation details of each ecosystem. Second, our questionnaires minimize the overlapping between segments of code that need to be analyzed throughout each session. More sophisticated mechanisms are needed to allow a more flexible answering strategy, thus increasing the generality of our results. However, the strategy used in this paper is realistic in the context of state-of-the-art program comprehension studies, that measure the efficiency of developers using linear questionnaires, such as in [88,89] and [90].

Our survey application presents three types of questions, namely, multiple choice questions, list-based questions, and multiple selection questions. We are aware that multiple choice and multiple selection questions may provide hints to participants, guiding them to investigate specific artifacts, thus reducing the precision of our performance measurements. In order to minimize this threat to validity, multiple choice and multiple selection questions presented a comprehensive list of artifacts needed to be considered in each task at hand. As a concrete example, tasks with the general form *"what metamodel elements are used in the creation of the [metamodel-name ! element-name] element"* included all of the metamodel elements found in the corresponding ecosystem under analysis. This effectively avoids drawing the attention of participants to specific artifacts, as well as narrowing their search to specific segments of code. An example of how this is presented to developers can be found here: https://github.com/guana/chaintracker-eval.

Concretely, 22 out of 25 questions in both questionnaires are multiple choice or multiple selection questions. The remaining 3 questions are list based, which receive one or multiple open-ended answers. List-based questions were used in tasks that require developers identifying bindings in model-to-text transformations. All multiple choice and multiple selection questions in the ScreenFlow questionnaire included a comprehensive list of their potential answers, i.e., metamodel elements and properties, transformation rules, and generated files. In the case of the PhyDSL, 20 questions provide all possible answers. The remaining 2 questions (which require the selection of metamodel element properties) provide a reduced, yet large number of answer possibilities. We do not believe that the nature of our type questions provided significant hints to developers during the completion of our study.

The study was divided in three sessions that took place over the span of a week. Our protocol was designed to minimize the fatigue of developers and allowed them to review the training material in between sessions. We believe this can potentially increase the developers' familiarity with the proposed families of tasks. Finally, during the last two years, we have iterated over ChainTracker's graphic user interface. We have conducted informal focus groups in order to make its features accessible and intuitive for developers. We have integrated the lessons learned in the tool demo sessions where ChainTracker has been showcased, i.e., the International Conference on Software Maintenance and Evolution (ICSME) in 2014 [27], the International Conference on Model Driven Engineering Languages and Systems in 2015 [29], and the IBM Technology Showcase in 2015.

External validity *(To what extend is it possible to generalize the findings?)* The case studies of our study are two model-based code generators implemented using ATL, a rule-based *model-to-model* transformation language, and Acceleo, a template-based *model-to-text* transformation technology. Therefore, any conclusions drawn from this study cannot be fully generalized to the performance of developers solving software engineering tasks on other model-transformation technologies. However, both Acceleo and ATL are widely adopted by academic and industry practitioners. More importantly, both languages are aligned to the OMG's *Query/View/-Transformation (QVT)* standard for model-to-model transformations [91], and the *Model to Text Transformation Language (MOF)* standard for model-to-text transformations [92], respectively. Therefore, the observations of this study can potentially be generalized to developers completing the same set of tasks, in ecosystems of similar size and complexity, and built using languages that comply with the same set of standards.

The case studies used in this study were developed in a research environment. We cannot claim that the results in this study can be generalized to industrial ecosystems. However, both case studies have been used in real software construction scenarios. Furthermore they both have being through development cycles that included platform and metamodel evolution scenarios, in order to meet the requirements of different clients. Considering the scope of our research, and the limited availability of industrial transformation ecosystems, we believe that the results of this study provide substantial insights on how developers trace and pinpoint metamodel and generation dependencies between the artifacts of a transformation ecosystem.

Even though the case studies considered in this study are model-transformation chains, the structure of the questionnaires included tasks that investigated the performance of developers dealing with single transformation steps of both model-to-model and model-to-text nature. Indeed, the results of this study can be generalized to the performance of developers dealing with transformations used in isolation, as well as in nontrivial transformation chains.

9 Conclusions and future work

In this paper, we present an empirical study that investigates the performance of developers when reflecting on the execution semantics of model-to-model and model-to-text transformations. We measured the accuracy and efficiency of developers when identifying metamodel and generation dependencies between transformation artifacts using ChainTracker, our traceability collection and analysis environment for model transformations, and we compared their performance against that of developers using Eclipse Modeling.

The hypothesis motivating our work has been that enabling developers to interactively explore the execution semantics of a transformation ecosystem can significantly improve developers' performance when reflecting on an ecosystem's design and evolution. ChainTracker was designed, based on this hypothesis, as a traceability collection and analysis environment to enable developers to explore the static and dynamic aspects of model-to-model, and model-to-text transformations. ChainTracker provides transformation visualizations, projectional code editors, and contextual tables. Each area of the analysis environment is synchronized with each other and provides interactive features that in conjunction help developers to reflect about the ecosystem execution. ChainTracker combines static and dynamic analysis strategies to gather fine-grained traceability information, including dependency relationships caused by both explicit and implicit binding expressions.

To evaluate ChainTracker's usefulness in supporting developers, we designed a collection of tasks that aim at understanding how developers reflect on the execution semantics of a transformation ecosystem. These tasks were grouped in five main families, namely, a) determining metamodel coverage and expression location, b) identifying metamodel dependencies in model-to-model transformations, c) identifying metamodel dependencies in model-to-text transformations, d) identifying generation dependencies in model-to-text transformations, and e) identifying generation dependencies in model-transformation chains. Regarding our two main research questions, let us briefly summarize our findings.

RQ1: *Do developers using ChainTracker identify metamodel and generation dependencies in transformation ecosystems more accurately and efficiently than those using Eclipse Modeling?*

We found that developers using ChainTracker had statistically significantly higher performance identifying metamodel dependencies in model-to-model transformations than those supported by Eclipse Modeling. Developers using ChainTracker were more accurate determining how transformation bindings create implicit and explicit dependency relationships at both metamodel element, and property levels of granularity.

Due to the lack of pattern-matching capabilities in ChainTracker, developers performed worse in finding individual transformation expressions. We believe supporting this type of tasks is very useful to developers detecting and removing duplicate bindings, i.e., code clones, as well as identifying transformation-refactoring opportunities. Indeed, our future work includes extending ChainTracker with pattern-matching capabilities to efficiently support developers locating bindings of interest in large transformation scripts.

We found that the performance of Eclipse developers was much worse for tasks that require identifying downstream metamodel element dependencies, than upstream dependencies, in model-to-model transformations. This is mostly due to the large code spaces that need to be manually analyzed if not supported by automatic reasoning tools. The impact of ChainTracker on the performance of developers was significantly higher when they were required to analyze model-transformation chains, and to determine end-to-end metamodel dependencies caused by the interdependent execution of model-to-model and model-to-text transformations.

Manually examining large transformation codebases, looking for the usage of dozens of metamodel elements, is a daunting if not impossible task. Developers were unable to accurately determine the coverage of metamodels in non-trivial transformation ecosystems using Eclipse Modeling. ChainTracker developers performed better, even if not at a statistically significantly level.

Our empirical observations suggest that interactive traceability visualizations are as useful as runtime transformation debuggers, to identify generation dependencies in model-to-text transformations. In the case of model-transformation chains, developers using ChainTracker were able to identify element- and property-level generation dependencies in transformation chains more efficiently and more accurately than developers using Eclipse. It is important to mention that developers using Eclipse performed very poorly identifying generation dependencies in model-transformation chains. Most of them obtained accuracy scores below the observed medians for this type of task, which overall was less than 50% of the maximum attainable score.

RQ2: *Do the size and complexity of transformation ecosystems affect the effectiveness of ChainTracker in helping developers identifying their metamodel and generation dependencies?*

We found that for all the families of tasks evaluated in this study, their complexity is considerably exacerbated by the size of the transformations, and the number of metamodel elements in an ecosystem. Particularly, for families of tasks that require the evaluation of model-transformation chains, the impact of ChainTracker on the performance of developers was positively more pronounced in the case of PhyDSL, a multibranched transformation chain.

We believe that the overall higher performance of developers using ChainTracker is due to the support that it provides identifying implicit and explicit bindings in complex OCL expressions. Considering that most developers are used to the execution semantics of imperative programming languages, ChainTracker considerably lowers the cognitive challenges that they face when getting used to declarative programming semantics. In effect, the active use of editors that allow two-way interactions between visualizations and transformation codebases proved effective to study information corresponding to metamodel and generation dependencies.

In this study, we gained important insights about the performance of developers using ChainTracker and Eclipse Modeling. However, more research is needed in order to understand how developers use the different features of ChainTracker. We plan to use eye-tracking technology to investigate which are the most valuable features of Chain-

Tracker. Particularly, we want to study the role that the contextual tables on the resolution of complex software engineering tasks.

Our future research agenda also includes an empirical study to emulate metamodel and platform evolution scenarios. We want to investigate how developers complete traceability-driven questions to fix, synchronize, and optimize the design of evolving transformation ecosystems using ChainTracker. Furthermore, with the advance of scalable of model-driven engineering, i.e., BigMDE [93], and the emerging fields of parallel model transformation execution, we believe ChainTracker can be extended with additional views to help developers study the performance of complex transformation ecosystems, as well as their physical runtime allocation.

Acknowledgements This work was supported by The Killam Trust, NSERC (the Discovery and the IRC program), the GRAND NCE and IBM Canada. The development of PhyDSL was supported by the AGE-WELL NCE. We would like to thank Kelsey Gaboriau and Vina Nguyen for their support during the development of ChainTracker. We would also like to thank Kelly Garces for her generous help during the execution of this study.

Appendix A: Questionnaires

See Tables 3 and 4.

Table 3 Session A: ScreenFlow Questionnaire

N.	Question
Q1	What ATL script contains the "Trigger2Button" rule?
Q2	In Mockup2GUI.atl, what transformation rule contains the binding expression `isMain<-mockscreen.main.toString().endsWith("true")`?
Q3	What metamodel elements are used in the creation of the GUI!Application element?
Q4	What metamodel elements are created using the Mockup!TriggerSection element?
Q5	Considering the entire transformation chain, what metamodel elements does the template line 24 in generateControlles.mtl depend on?
Q6	What template lines in generateControlles.mtl are used in the generation of line 21 in PlayerActivity.java?
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 4 in login.xml depend on?
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 34 in AndroidManifest.xml depend on?
Q9	What files does the generateControllers.mtl template generate?
Q10	What files does the generateViews.mtl template generate?
Q11	What metamodel elements are used in the creation of the GUI!Screen element?
Q12	What metamodel elements are created using the Mockup!ScreenSection element?
Q13	What metamodel elements are created using the property "name" of the Mockup!Screen element?
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 17 in login_activity.xml depend on?
Q15	Considering the entire transformation chain, what metamodel elements does the template line 19 in generateViews.mtl depend on?
Q16	What template lines in generateControllers.mtl are used in the generation of line 37 in LoginActivity.java?
Q17	Are there any unused elements in the Mockup metamodel? If so, which ones?
Q18	Are there any unused elements in the GUI metamodel?
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in AndroidManifest.xml depend on?
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 38 in LoginActivity.java depend on?
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in login_activity.xml depend on?
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 41–44 in generateControllers.mtl depend on?
Q23	What metamodel elements are created using the property "fromScreen" of the Mockup!Transition element?
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 8 in login_activity.xml depend on?
Q25	What template lines in generateViews.mtl are used in the generation of line 27 in AndroidManifest.java?

Table 4 Session B: PhyDSL Questionnaire

N.	Question
Q1	What ATL script contains the "Effect2Action" rule?
Q2	In Game2Layout.atl, what transformation rule contains the binding expression `isBall<-r.actorDefinition.first().isBall.boolean.solveBool()`?
Q3	What metamodel elements are used in the creation of the Scoring!TouchRule element?
Q4	What metamodel elements are created using the PhyDSL!Coordinate element?
Q5	Considering the entire transformation chain, what metamodel elements does the template line 148 in generateDynamics.mtl depend on?
Q6	What template lines in generateScoring.mtl are used in the generation of line 102 in ScoringManager.java?
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 100 in ScoringManager.java depend on?
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 220 in DrawingHelper.java depend on?
Q9	What files does the generateLayout.mtl template generate?
Q10	What files does the generateGraphics.mtl template generate?
Q11	What metamodel elements are used in the creation of the Scoring!CollisionRule element?
Q12	What metamodel elements are created using the PhyDSL!Ends element?
Q13	What metamodel elements are created using the property "name" of the PhyDSL!Actor element?
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 76 in ScoringManager.java depend on?
Q15	Considering the entire transformation chain, what metamodel elements does the template line 110 in generateScoring.mtl depend on?
Q16	What template lines in generateLayout.mtl are used in the generation of line 264 in PhysicsView.java?
Q17	Are there any unused elements in the PhyDSL metamodel? If so, which ones?
Q18	Are there any unused elements in the Dynamics metamodel? If so, which ones?
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 141 in MainActivity.java depend on?
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 29 in ControlManager.java depend on?
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 281 in PhysicsView.java depend on?
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 104–108 in generateControls.mtl depend on?
Q23	What metamodel elements are created using the property "value" of the PhyDSL!BooleanType element?
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 18 in ControlManager.java depend on?
Q25	What template lines in generateControls.mtl are used in the generation of line 99 in ControlManager.java?

Appendix B: Result tables

See Tables 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.

Result summary tables are divided in two main areas. The first area presents the p values corresponding to the statistical evaluation of our hypotheses. We present individual p values for each of the dependent variables under consideration, namely the time and accuracy of developers completing a task. The second area compares the mean, standard deviation (SD), and median of the dependent variables across treatments. Furthermore, summary tables include the maximum attainable score for each task, and two box and whisker diagrams that portray the distribution of the recorded measurements for both variables under consideration. In all diagrams, green boxes portray the distribution of measurements for participants using ChainTracker (CT), and purple boxes for participants using Eclipse (EC).

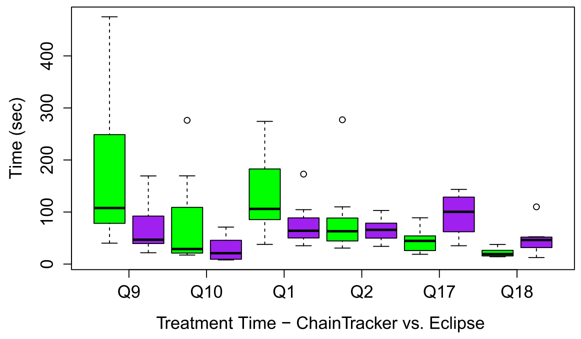
Table 5 Session A (ScreenFlow)—results: determining metamodel coverage and expression location

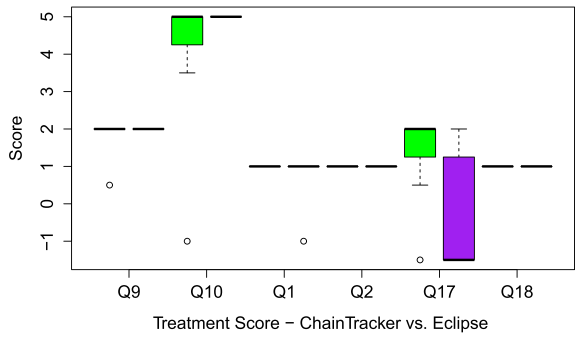
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q9	What files does the generateControllers.mtl template generate?	0.3496	0.0721
Q10	What files does the generateViews.mtl template generate?	0.1423	0.2319
Q1	What ATL script contains the 'Trigger2Button' rule?	0.4227	0.1206
Q2	In Mockup2GUI.atl, what transformation rule contains the binding expression `isMain`...?	1.0000	0.9551
Q17	Are there any unused elements in the Mockup metamodel?	**0.0066***	**0.0093***
Q18	Are there any unused elements in the GUI metamodel?	1.0000	**0.0400***

Treatments Time Mean and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q9	182.14	157.90	68.00	48.21
Q10	82.71	100.89	28.62	25.45
Q1	135.71	81.56	76.25	43.78
Q2	90.42	86.21	65.25	21.94
Q17	44.00	24.07	94.62	38.87
Q18	21.85	8.53	47.25	28.96

Treatments Score Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.78	0.56	2.00	0.00
5	3.92	2.24	5.00	0.00
1	1.00	0.00	0.75	0.70
1	1.00	0.00	1.00	0.00
2	1.28	1.34	-0.37	1.62
1	1.00	0.00	1.00	0.00

Median Time Comparison		Median Score Comparison	
CT	**EC**	**CT**	**EC**
107.00	**46.00**	2.00	2.00
28.00	**20.50**	5.00	5.00
105.00	**63.50**	1.00	1.00
90.42	**65.00**	1.00	1.00
44.00	100.00	**2.00**	-1.50
19.00	46.00	1.00	1.00





Bold values indicate best median performance
* Statistical significant value

Table 6 Session B (PhyDSL)—results: determining metamodel coverage and expression location

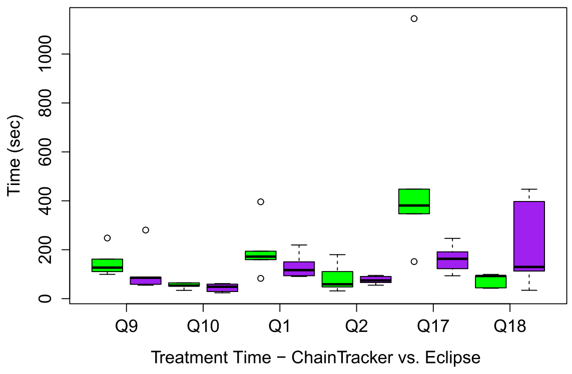
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q9	What files does the generateLayout.mtl template generate?	1.0000	0.1508
Q10	What files does the generateGraphics.mtl template generate?	1.0000	0.3095
Q1	What ATL script contains the 'Effect2Action' rule?	1.0000	0.4206
Q2	In Game2Layout.atl, what transformation rule contains the binding expression `isBall`...	1.0000	0.8413
Q17	Are there any unused elements in the PhyDSL metamodel?	0.1563	0.0555
Q18	Are there any unused elements in the Dynamics metamodel?	0.09296	0.1508

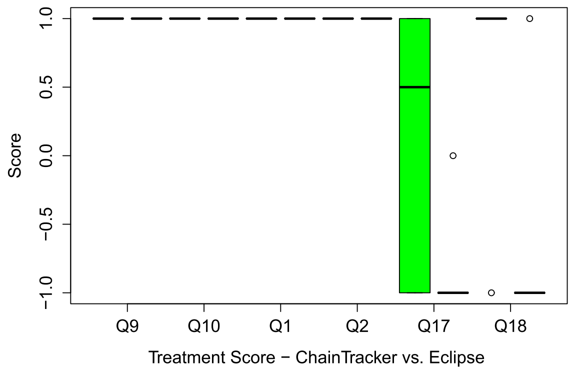
Treatments Time Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q9	148.60	59.78	113.20	0.00
Q10	53.00	12.70	44.00	17.29
Q1	200.00	116.74	133.60	53.43
Q2	85.00	60.31	75.60	17.06
Q17	493.80	379.79	162.80	59.69
Q18	74.00	28.43	223.80	185.28

Treatments Score Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	0.09	1.02	-0.80	0.44
1	0.60	0.89	-0.60	0.89

Median Time Comparison	
CT	**EC**
126.00	**85.00**
53.00	**48.00**
171.00	**116.00**
58.00	74.00
380.00	**162.00**
92.00	129.00

Median Score Comparison	
CT	**EC**
1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00
0.50	-1.00
1.00	-1.00





Bold values indicate best median performance

Table 7 Session A (ScreenFlow)—results: identifying metamodel dependencies in M2M transformations (element level)

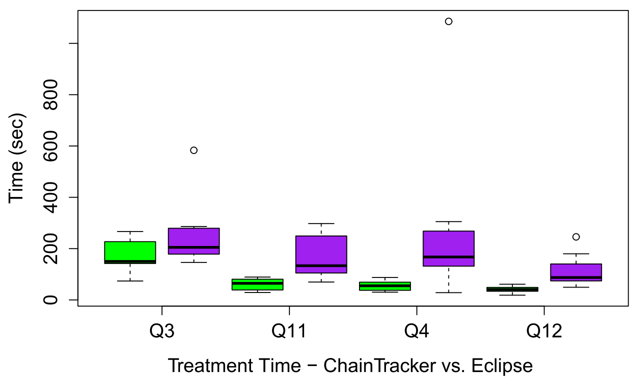
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	Score (p-value)	Time (p-value)
Q3	What metamodel elements are used in the creation of the GUI!Application element?	0.0675	0.1893
Q11	What metamodel elements are used in the creation of the GUI!Screen element?	**0.0310***	**0.0021***
Q4	What metamodel elements are created using the Mockup!TriggerSection element?	1.0000	**0.0139***
Q12	What metamodel elements are created using the Mockup!ScreenSection element?	**0.0006***	**0.0012***

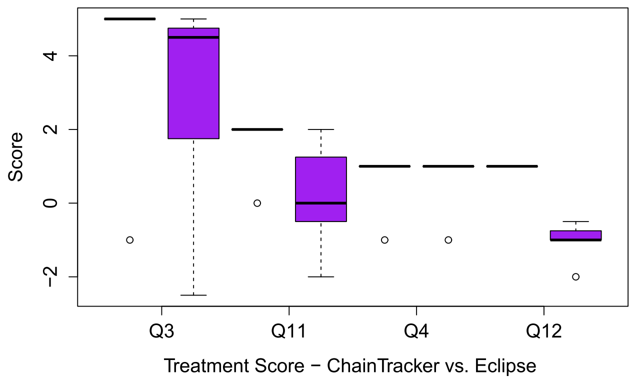
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q3	174.85	66.98	256.37	140.37
Q11	59.71	25.02	167.12	84.90
Q4	54.85	22.01	280.75	334.90
Q12	40.28	14.52	111.62	66.18

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
5	4.14	2.26	3.06	2.70
2	1.71	0.75	0.18	1.36
1	0.71	0.75	0.75	0.70
1	1.00	0.00	-1.00	0.46

Median Time Comparison	
CT	**EC**
149.00	204.00
64.00	132.50
55.00	167.00
40.00	86.50

Median Score Comparison	
CT	**EC**
5.00	4.50
2.00	0.00
1.00	1.00
1.00	-1.00





Bold values indicate best median performance
* Statistical significant value

Table 8 Session B (PhyDSL)—results: identifying metamodel dependencies in M2M transformations (element level)

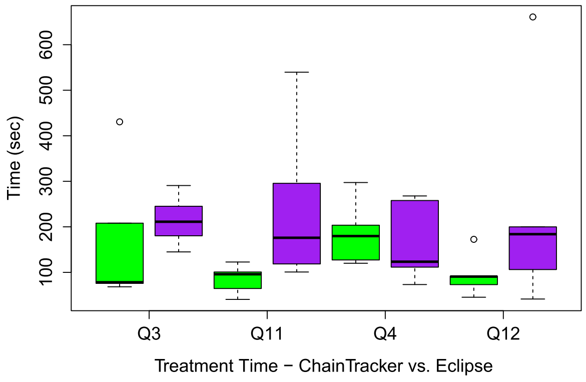
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	Score (p-value)	Time (p-value)
Q3	What metamodel elements are used in the creation of the Scoring!TouchRule element?	**0.0065***	0.3095
Q11	What metamodel elements are used in the creation of the Scoring!CollisionRule element?	**0.0310***	0.0555
Q4	What metamodel elements are created using the PhyDSL!Coordinate element?	**0.0072***	0.5476
Q12	What metamodel elements are created using the PhyDSL!Ends element?	**0.0065***	0.2222

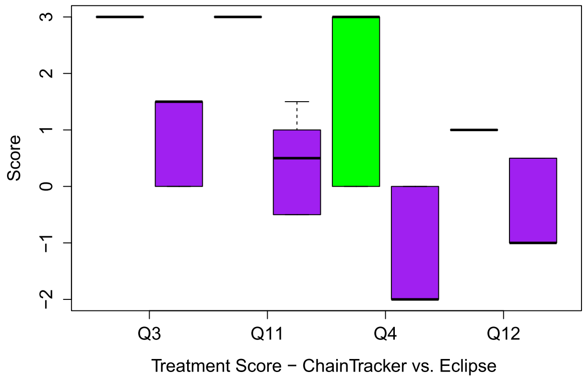
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q3	171.80	155.66	214.00	56.57
Q11	84.20	32.23	245.60	180.86
Q4	185.20	71.57	166.20	89.44
Q12	94.20	47.30	237.80	244.36

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
3	3.00	0.00	0.90	0.82
3	3.00	0.00	0.40	0.89
3	1.80	1.64	-1.20	1.09
1	1.00	0.00	-0.40	0.82

Median Time Comparison	
CT	**EC**
78.00	211.00
95.00	176.00
179.00	123.00
90.00	183.00

Median Score Comparison	
CT	**EC**
3.00	1.50
3.00	0.50
3.00	-2.00
1.00	-1.00





Bold values indicate best median performance
*Statistical significant value

Table 9 Session A (ScreenFlow)—results: identifying metamodel dependencies in single M2M transformations (property level)

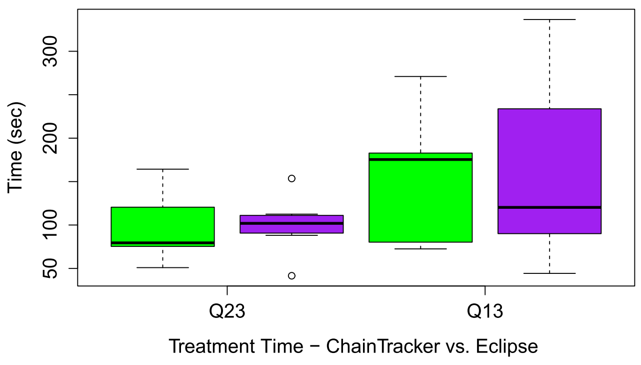
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q23	What metamodel elements are created using the property 'fromScreen' of the Mockup!Transition element?	**0.0046***	0.6943
Q13	What metamodel elements are created using the property 'name' of the Mockup!Screen element?	**0.0491***	0.9551

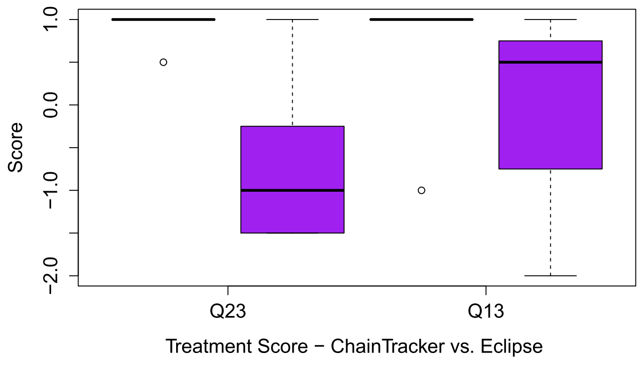
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q23	97.57	39.55	99.87	31.05
Q13	148.8	74.76	158.00	105.53

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	0.92	0.18	-0.75	0.96
1	0.71	0.75	0.00	1.13

Median Time Comparison	
CT	**EC**
79.00	101.5
175.0	119.5

Median Score Comparison	
CT	**EC**
1.00	-1.00
1.00	0.50





Bold values indicate best median performance
* Statistical significant value

Table 10 Session B (PhyDSL)—results: identifying metamodel dependencies in M2M transformations (property level)

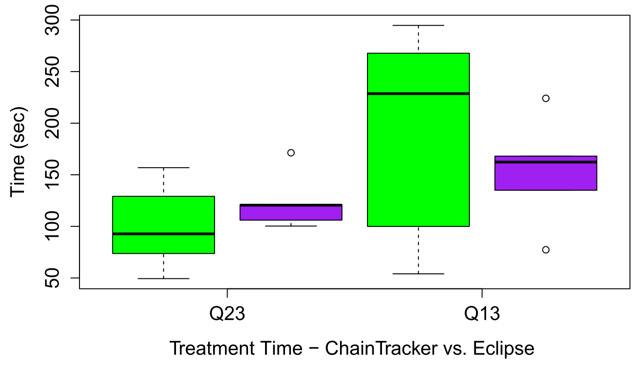
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q23	What metamodel elements are created using the property 'value' of the PhyDSL!BooleanType element?	**0.0412***	0.4206
Q13	What metamodel elements are created using the property 'name' of the PhyDSL!Actor element?	**0.0393***	0.5476

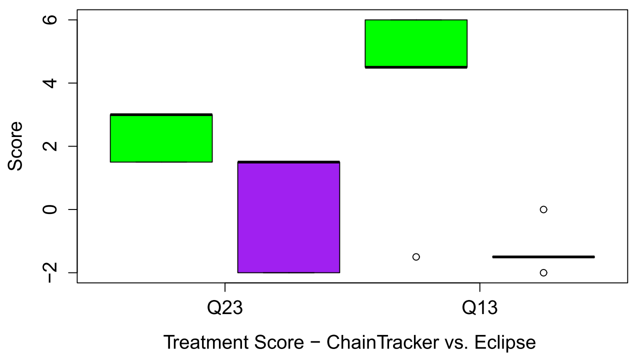
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q23	99.80	42.92	123.60	27.98
Q13	188.40	106.01	153.20	53.49

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
3	2.40	0.82	0.09	1.91
6	3.90	3.11	-1.30	0.75

Median Time Comparison	
CT	**EC**
92.00	120.00
228.00	**162.00**

Median Score Comparison	
CT	**EC**
3.00	1.50
4.50	-1.50





Bold values indicate best median performance
* Statistical significant value

Table 11 Session A (ScreenFlow)—results: identifying metamodel dependencies in M2T transformations

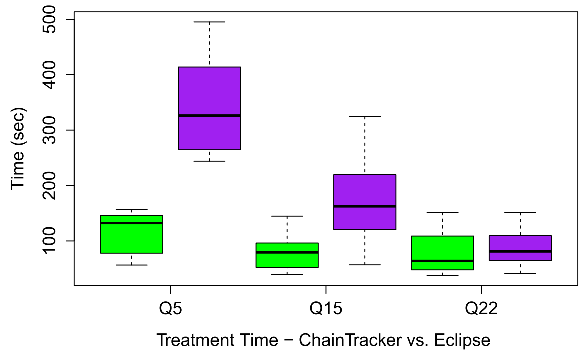
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q5	Considering the entire transformation chain, what metamodel elements does the template line 24 in generateControllers.mtl depend on?	**0.0119***	**0.0003***
Q15	Considering the entire transformation chain, what metamodel elements does the template line 19 in generateViews.mtl depend on?	0.2143	**0.0205***
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 41-44 in generateControllers.mtl depend on?	**0.0025***	0.6126

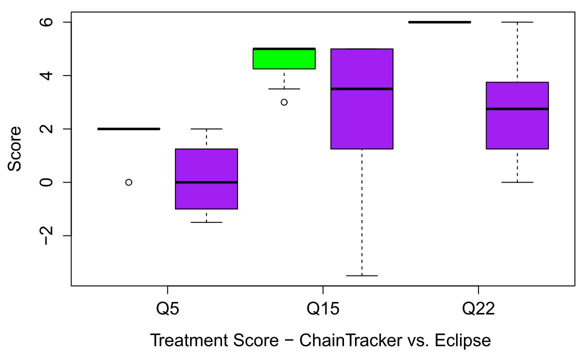
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q5	112.85	41.42	343.12	91.70
Q15	79.57	37.23	172.75	82.97
Q22	80.28	45.18	87.625	35.12

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.71	0.75	0.12	1.27
5	5.00	0.00	2.62	3.05
6	6.00	0.00	2.68	1.88

Median Time Comparison	
CT	**EC**
132.00	326.00
79.00	162.00
63.00	81.00

Median Score Comparison	
CT	**EC**
2.00	0.00
5.00	3.50
6.00	2.75





Bold values indicate best median performance
* Statistical significant value

Table 12 Session B (PhyDSL)—results: identifying metamodel dependencies in M2T transformations

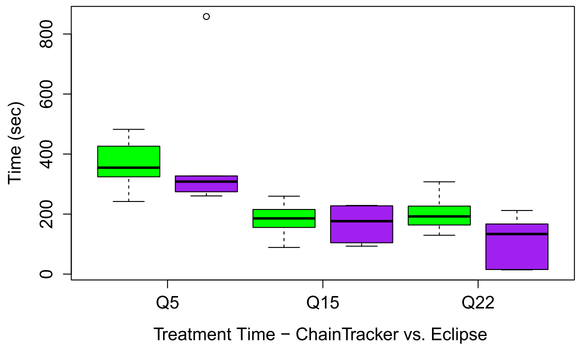
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q5	Considering the entire transformation chain, what metamodel elements does the template line 148 in generateDynamics.mtl depend on?	**0.0072***	0.6905
Q15	Considering the entire transformation chain, what metamodel elements does the template line 110 in generateScoring.mtl depend on?	**0.0097***	1.0000
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 104-108 in generateControls.mtl depend on?	**0.0117***	0.2222

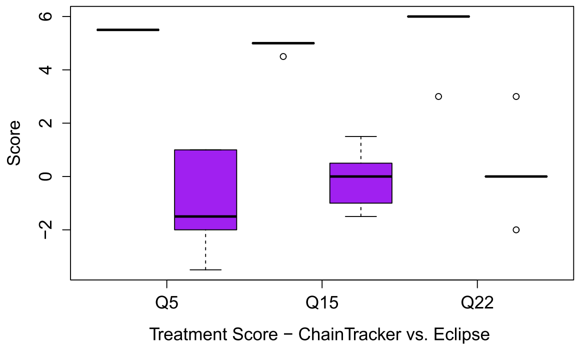
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q5	365.60	92.65	405.40	254.40
Q15	180.40	64.38	165.60	64.87
Q22	203.40	68.08	108.20	90.00

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
6	5.50	0.00	-1.00	1.96
5	4.90	0.22	-0.09	1.19
6	5.40	1.34	0.19	1.78

Median Time Comparison	
CT	**EC**
354.00	**308.00**
185.00	**176.00**
192.00	**133.00**

Median Score Comparison	
CT	**EC**
5.50	-1.50
5.00	0.00
6.00	0.00





Bold values indicate best median performance
* Statistical significant value

Table 13 Session A (ScreenFlow)—results: identifying generation dependencies in M2T transformations

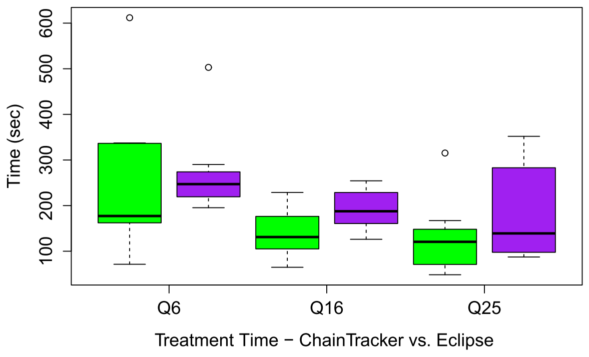
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q6	What template lines in generateControllers.mtl are used in the generation of line 21 in PlayerActivity.java?	0.5014	0.5358
Q16	What template lines in generateControllers.mtl are used in the generation of line 37 in LoginActivity.java?	0.0874	0.0938
Q25	What template lines in generateViews.mtl are used in the generation of line 27 in AndroidManifest.xml?	**0.0377***	0.3969

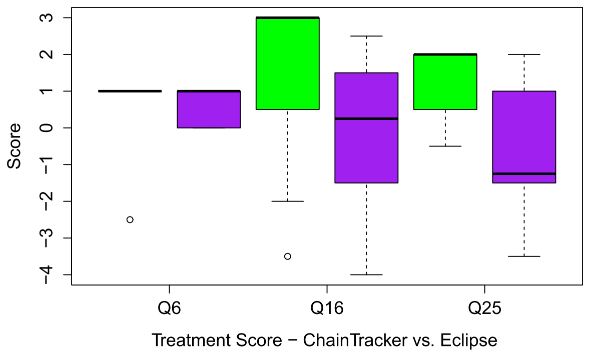
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q6	264.85	181.08	271.87	98.24
Q16	140.42	60.10	191.37	43.42
Q25	131.42	91.66	184.25	104.5

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	0.50	1.32	0.62	0.51
3	1.35	2.83	-0.12	2.19
2	1.21	1.03	-0.62	1.82

Median Time Comparison	
CT	**EC**
177.00	246.50
131.00	187.00
120.00	138.50

Median Score Comparison	
CT	**EC**
1.00	1.00
3.00	0.25
2.00	-1.25





Bold values indicate best median performance
* Statistical significant value

Table 14 Session B (PhyDSL)—results: identifying generation dependencies in M2T transformations

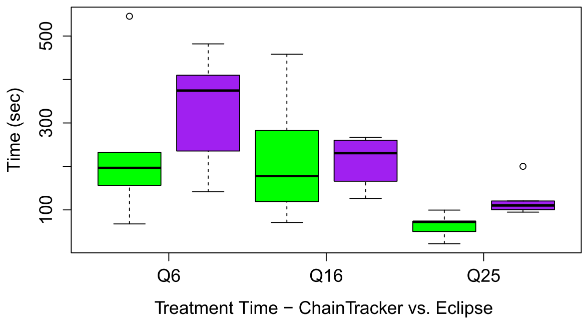
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q6	What template lines in generateScoring.mtl are used in the generation of line 102 in ScoringManager.java?	**0.0482***	0.4206
Q16	What template lines in generateLayout.mtl are used in the generation of line 264 in PhysicsView.java?	0.8288	1.0000
Q25	What template lines in generateControls.mtl are used in the generation of line 99 in ControlManager.java?	1.0000	**0.0158***

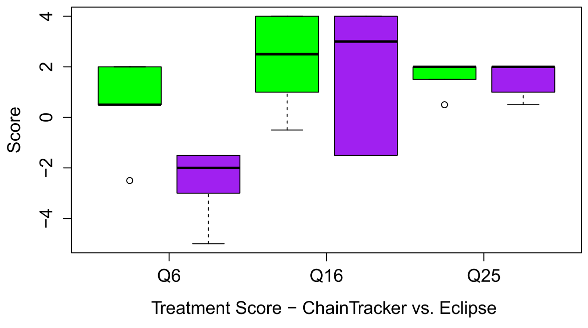
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q6	239.20	181.66	328.00	137.57
Q16	221.40	153.89	209.40	61.47
Q25	63.20	28.83	124.80	43.18

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	0.50	1.83	-2.60	1.47
4	2.20	1.95	1.59	2.85
2	1.60	0.65	1.50	0.70

Median Time Comparison	
CT	**EC**
196.00	374.00
177.00	230.00
72.00	110.00

Median Score Comparison	
CT	**EC**
0.50	-2.00
2.50	**3.00**
2.00	2.00





Bold values indicate best median performance
* Statistical significant value

Table 15 Session A (ScreenFlow)—results: identifying generation dependencies in MTCs (element level)

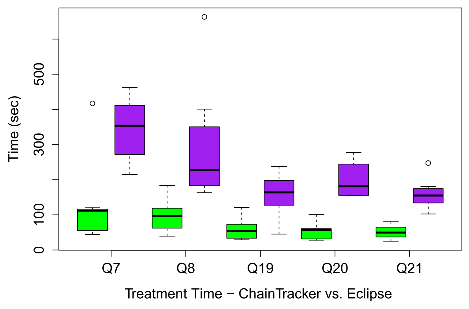
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 4 in login.xml depend on?	**0.0032***	**0.0093***
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 34 in AndroidManifest.xml depend on?	**0.0073***	**0.0012***
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in AndroidManifest.xml depend on?	**0.0443***	**0.0037***
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 38 in LoginActivity.java depend on?	**0.0007***	**0.0003***
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in login_activity.xml depend on?	**0.0007***	**0.0003***

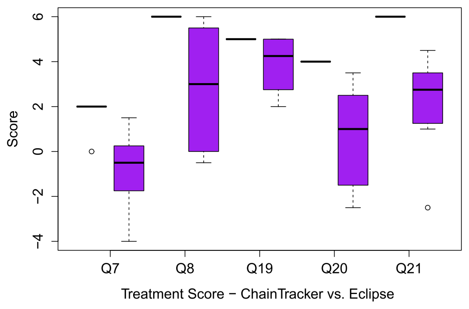
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q7	130.57	129.99	343.25	85.39
Q8	96.85	50.61	293.12	168.04
Q19	59.00	3.23	157.12	60.58
Q20	52.28	25.75	198.75	51.61
Q21	50.85	22.11	159.12	43.22

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.71	0.75	-0.81	1.71
6	6.00	0.00	2.81	2.84
5	5.00	0.00	3.87	1.32
4	4.00	0.00	0.62	2.24
6	6.00	0.00	2.12	2.19

Median Time Comparison	
CT	**EC**
112.00	353.00
96.00	227.50
53.00	163.50
57.00	180.50
49.00	154.50

Median Score Comparison	
CT	**EC**
2.00	-0.50
6.00	3.00
5.00	4.25
4.00	1.00
6.00	2.75





Bold values indicate best median performance
* Statistical significant value

Table 16 Session B (PhyDSL)—results: identifying generation dependencies in MTCs (element level)

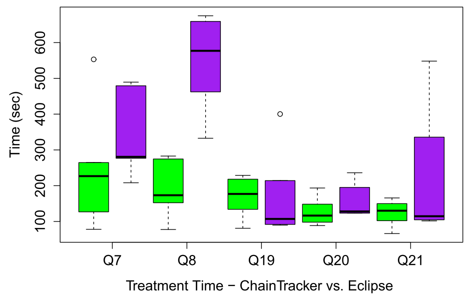
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 100 in ScoringManager.java depend on?	**0.0094***	0.3095
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 220 in DrawingHelper.java depend on?	**0.0072***	**0.0079***
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 141 in MainActivity.java depend on?	**0.0066***	0.8413
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 29 in ControlManager.java depend on?	**0.0055***	0.2222
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 281 in PhysicsView.java depend on?	**0.0055***	0.6905

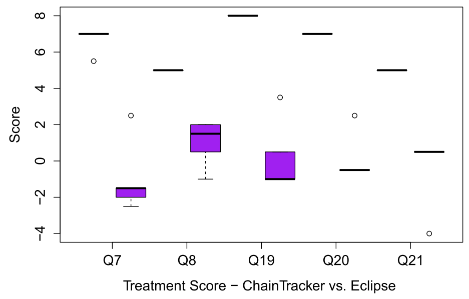
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q7	249.4	185.43	346.40	128.87
Q8	191.6	86.61	540.80	143.90
Q19	167.0	61.49	180.20	133.31
Q20	128.4	42.80	161.00	51.86
Q21	122.4	39.29	240.40	198.48

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
7	6.70	0.67	-1.00	2.00
5	5.00	0.00	1.00	1.27
8	8.00	0.00	0.20	1.95
7	7.00	0.00	0.10	1.34
5	5.00	0.00	-0.40	2.01

Median Time Comparison	
CT	**EC**
226.00	280.00
173.00	577.00
176.00	107.00
116.00	128.00
130.00	114.00

Median Score Comparison	
CT	**EC**
7.00	-1.50
5.00	1.50
8.00	-1.00
7.00	-0.50
5.00	0.50





Bold values indicate best median performance
* Statistical significant value

Table 17 Session A (ScreenFlow)—results: identifying generation dependencies in MTCs (property level)

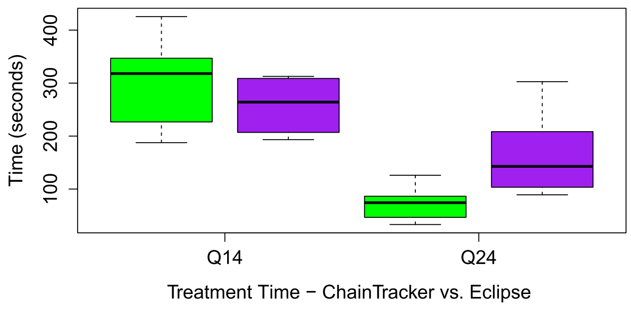
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 17 in login_activity.xml depend on?	**0.0090***	0.3969
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 8 in login_activity.xml depend on?	**0.04871***	**0.0059***

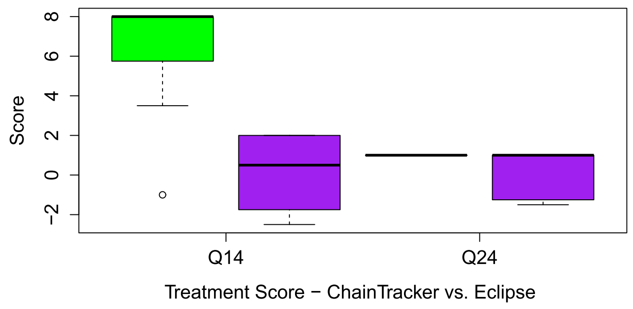
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q14	296.42	88.17	257.75	50.20
Q24	71.14	32.69	162.25	74.83

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
8	6.07	3.54	0.125	1.92
1	1.00	0.00	0.125	1.21

Median Time Comparison	
CT	**EC**
317.00	**263.50**
74.00	142.50

Median Score Comparison	
CT	**EC**
8.00	0.50
1.00	1.00





Bold values indicate best median performance
* Statistical significant value

Table 18 Session B (PhyDSL)—results: identifying generation dependencies in MTCs (property level)

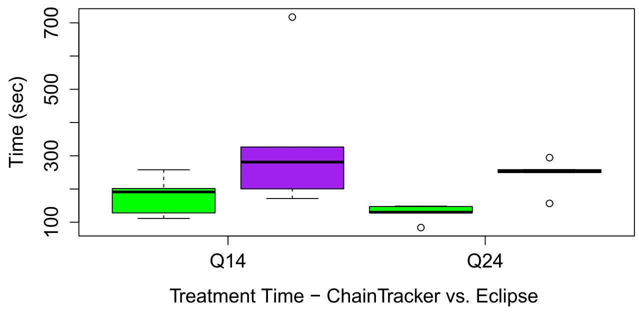
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 76 in ScoringManager.java depend on?	**0.0066***	0.1508
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 18 in ControlManager.java depend on?	**0.0248***	**0.0079***

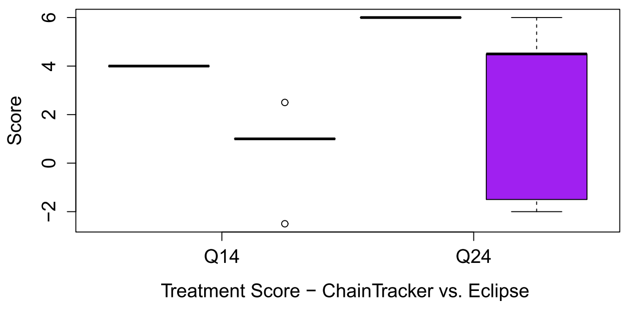
Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q14	177.60	59.00	339.00	220.20
Q24	127.40	25.97	242.00	51.29

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
4	4.00	0.00	0.60	1.85
6	6.00	0.00	2.30	3.75

Median Time Comparison	
CT	**EC**
191.00	281.00
130.00	253.00

Median Score Comparison	
CT	**EC**
4.00	1.00
6.00	4.50





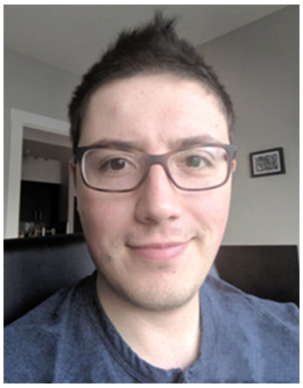
Bold values indicate best median performance
* Statistical significant value

References

1. Czarnecki, K.: Generative programming: methods, techniques, and applications tutorial abstract. In: Software Reuse: Methods, Techniques, and Tools, pp. 477–503 (2002)
2. Hemel, Z., Kats, L.C., Visser, E.: Code generation by model transformation, pp. 183–198 (2008)
3. Vara, J.M., Vela, B., Cavero, J.M., Marcos, E.: Model transformation for object-relational database development. In: Proceedings of the 2007 ACM Symposium on Applied Computing. ACM, pp. 1012–1019 (2007)
4. Vara, J.M., Marcos, E.: A framework for model-driven development of information systems: technical decisions and lessons learned. J. Syst. Softw. **85**(10), 2368–2384 (2012)
5. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA-visual automated transformations for formal verification and validation of UML models. In: Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on. IEEE, pp. 267–270 (2002)
6. Cariou, E., Ballagny, C., Feugas, A., Barbier, F.: Contracts for model execution verification. In: European Conference on Modelling Foundations and Applications, pp. 3–18. Springer, Berlin (2011)
7. Selim, G.M., Wang, S., Cordy, J.R., Dingel, J.: Model transformations for migrating legacy deployment models in the automotive industry. Softw. Syst. Model. **14**(1), 365–381 (2015)
8. Kavimandan, A., Gokhale, A.: Applying model transformations to optimizing real-time QOS configurations in DRE systems. In: International Conference on the Quality of Software Architectures, pp. 18–35. Springer, Berlin (2009)
9. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: 2007 Future of Software Engineering. IEEE Computer Society, pp. 37–54 (2007)
10. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 471–480 (2011)
11. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In: International Conference on Graph Transformation, pp. 20–37. Springer, Berlin (2012)
12. Walderhaug, S., Stav, E., Johansen, U., Olsen, G.K.: Traceability in model-driven software development. In: Designing Software-Intensive Systems: Methods and Principle, pp. 133–159 (2008)
13. Von Knethen, A., Grund, M.: Quatrace: a tool environment for (semi-) automatic impact analysis based on traces. In: Software Maintenance, 2003. ICSM 2003. Proceedings of International Conference on. IEEE, pp. 246–255 (2003)
14. Winkler, S., Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. **9**(4), 529–565 (2010)
15. Amar, B., Leblanc, H., Coulette, B.: A traceability engine dedicated to model transformation for software engineering. In: ECMDA Traceability Workshop (ECMDA-TW), pp. 7–16 (2008)
16. Correa, A., Werner, C.: Applying refactoring techniques to UML/OCL models. In: << UML>> 2004-The Unified Modeling Language. Modelling Languages and Applications. Springer, Berlin, pp. 173–187 (2004)
17. Ermel, C., Ehrig, H., Ehrig, K.: Refactoring of model transformations. In: Electronic Communications of the EASST, vol. 18 (2009)
18. Wang, J., Kim, S.-K., Carrington, D.: Verifying metamodel coverage of model transformations. In: Software Engineering Conference, 2006, Australian. IEEE, p. 10 (2006)
19. Wang, J., Kim, S., Carrington, D.: Automatic generation of test models for model transformations. In: 19th Australian Conference on Software Engineering (ASWEC 2008). IEEE, pp. 432–440 (2008)
20. Finot, O., Mottu, J.-M., Sunyé, G., Degueule, T.: Using meta-model coverage to qualify test oracles. In: Analysis of Model Transformations, pp. 1613–0073 (2013)
21. Burgueno, L.: Testing M2M/M2T/T2M transformations. In: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on (Student Competition) (2015)
22. Van Amstel, M.F., Van Den Brand, M.G.: Model transformation analysis: staying ahead of the maintenance nightmare, pp. 108–122 (2011)
23. Kleppe, A.: First european workshop on composition of model transformations-cmt 2006. In: Technical Report TR-CTIT-06-34 (2006)
24. Kemerer, C.F.: Now the learning curve affects case tool adoption. IEEE Softw. **9**(3), 23–28 (1992)
25. Hardgrave, B.C., Davis, F.D., Riemenschneider, C.K.: Investigating determinants of software developers' intentions to follow methodologies. J. Manag. Inf. Syst. **20**(1), 123–151 (2003)
26. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: are the tools really the problem? In: Model-Driven Engineering Languages and Systems, pp. 1–17. Springer, Berlin (2013)
27. Guana, V., Gaboriau, K., Stroulia, E.: Chaintracker: towards a comprehensive tool for building code-generation environments. In: Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME). IEEE Press (2014)
28. Guana, V., Stroulia, E.: Chaintracker, a model-transformation trace analysis tool for code-generation environments. In: Theory and Practice of Model Transformations, pp. 146–153. Springer, Berlin (2014)
29. Guana, V., Stroulia, E.: Reflecting on model-based code generators using traceability information. In: ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2015)
30. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, Reading (2003)
31. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, pp. 128–138. Springer, Berlin
32. Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., Allilaire, F.: Acceleo User Guide (2006)
33. Bragança, A., Machado, R.J.: Transformation patterns for multi-staged model driven software development. In: Software Product Line Conference, 2008. SPLC'08, 12th International. IEEE, pp. 329–338 (2008)
34. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, vol. 45, no. 3. Citeseer, pp. 1–17 (2003)
35. Van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: a research agenda. In: Proceedings 1st International Workshop on Model-Driven Software Evolution, pp. 41–49 (2007)
36. Bennett, K., Rajlich, V.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. ACM, pp. 73–87 (2000)
37. Di Ruscio, D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and atl transformations. In: International Conference on Theory and Practice of Model Transformations, pp. 60–75. Springer, Berlin (2013)
38. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: Alfonso Pierantonio (co-chair) Universita degli

Studi dellâĂŹAquila (Italy) Bernhard Schätz (co-chair) fortiss GmbH (Germany), p. 22 (2014)
39. Group, O.M.: A proposal for an MDA foundation model. Needham ormsc/05-04-01 ed (2005)
40. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. **45**(3), 515–526 (2006)
41. Galvao, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. IEEE, pp. 313–313 (2007)
42. Santiago, I., Jiménez, A., Vara, J.M., De Castro, V., Bollati, V.A., Marcos, E.: Model-driven engineering as a new landscape for traceability management: a systematic literature review. Inf. Softw. Technol. **54**(12), 1340–1356 (2012)
43. Falleri, J., Huchard, M., Nebut, C. et al.: Towards a traceability framework for model transformations in kermeta (2006)
44. Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) Workshop on Traceability, Nuremberg, Germany, vol. 91. Citeseer (2005)
45. von Pilgrim, J., Vanhooff, B., Schulz-Gerlach, I., Berbers, Y.: Constructing and visualizing transformation chains. In: Model Driven Architecture–Foundations and Applications. Springer, Berlin (2008)
46. Matragkas, N.D., Kolovos, D.S., Paige, R.F., Zolotas, A.: A traceability-driven approach to model transformation testing. In: AMT@MoDELS (2013)
47. Santiago, I., Vara, J.M., de Castro, M.V., Marcos, E.: Towards the effective use of traceability in model-driven engineering projects. In: Conceptual Modeling, pp. 429–437. Springer, Berlin (2013)
48. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Traceability visualization in metamodel change impact detection. In: Proceedings of the Second Workshop on Graphical Modeling Language Development. ACM, pp. 51–62 (2013)
49. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the coupled evolution of metamodels and textual concrete syntax specifications. In: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on. IEEE, pp. 114–121 (2013)
50. van Amstel, M., Serebrenik, A., van den Brand, M.: Visualizing traceability in model transformation compositions. In: Pre-Proceedings of the First Workshop on Composition and Evolution of Model Transformations (2011)
51. Oldevik, J., Neple, T.: Traceability in model to text transformations. In: 2nd ECMDA Traceability Workshop (ECMDA-TW). Citeseer, pp. 17–26 (2006)
52. García, J., Azanza, M., Irastorza, A., Díaz, O.: Testing MOFscript transformations with HandyMOF. In: Theory and Practice of Model Transformations, pp. 42–56. Springer, Berlin (2014)
53. Olsen, G.K., Oldevik, J.: Scenarios of traceability in model to text transformations. In: European Conference on Model Driven Architecture-Foundations and Applications, pp. 144–156. Springer, Berlin (2007)
54. Santiago, I., Vara, J.M., de Castro, V., Marcos, E.: Reducing the level of complexity of working with model transformations. In: International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 1–17. Springer, Berlin (2014)
55. Wieringa, R.: An introduction to requirements traceability (1995)
56. Almeida, J., Van Eck, P., Iacob, M.: Requirements traceability and transformation conformance in model-driven development. In: Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International. IEEE, pp. 355–366 (2006)
57. Pinheiro, F.A.: Requirements traceability. In: Perspectives on Software Requirements. Springer, Berlin, pp. 91–113 (2004)
58. Duan, C., Cleland-Huang, J.: Visualization and analysis in automated trace retrieval. In: 2006 First International Workshop on Requirements Engineering Visualization (REV'06-RE'06 Workshop). IEEE, p. 5 (2006)
59. Card, D.N.: Designing software for producibility. J. Syst. Softw. **17**(3), 219–225 (1992)
60. Cleland-Huang, J.: Toward improved traceability of non-functional requirements. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering. ACM, pp. 14–19 (2005)
61. Acceleo traceability: Eclipse plug-in, http://goo.gl/eenOE3. Accessed 14 Sept 2016
62. Atlas transformation language (atl) user guide, http://goo.gl/KzPaze. Accessed 14 Sept 2016
63. Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the on Future of Software Engineering. ACM, pp. 55–69 (2014)
64. Cuadrado, J., Molina, J., Tortosa, M.: Rubytl: a practical, extensible transformation language. In: Model Driven Architecture–Foundations and Applications, pp. 158–172. Springer, Berlin (2006)
65. Kolovos, D., Paige, R., Polack, F.: The epsilon transformation language. In: *Theory and Practice of Model Transformations*, pp. 46–60 (2008)
66. T. project (IRISA), *The Metamodeling Language Kermeta*. http://www.kermeta.org (2006)
67. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: GraphvizâĂŤopen source graph drawing tools. In: Graph Drawing, pp. 483–484. Springer, Berlin (2002)
68. van Amstel, M.F., van den Brand, M.G., Serebrenik, A.: Traceability visualization in model transformations with tracevis. In: International Conference on Theory and Practice of Model Transformations, pp. 152–159. Springer, Berlin (2012)
69. Von Pilgrim, J., Duske, K., McIntosh, P.: Eclipse GEF3D: bringing 3D to existing 2D editors. Information Visualization **8**(2), 107–119 (2009)
70. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M. et al.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. ACM, p. 2 (2013)
71. Wettel, R., Lanza, M.: Program comprehension through software habitability. In: 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE, pp. 231–240 (2007)
72. Störrle, H.: On the impact of size to the understanding of UML diagrams. Softw. Syst. Model., pp. 1–20 (2016)
73. van Ravensteijn, W.: Visual traceability across dynamic ordered hierarchies (2011)
74. Holten, D.: Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. IEEE Trans. Vis. Comput. Gr. **12**(5), 741–748 (2006)
75. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
76. Soni, D., Nord, R. L., Hofmeister, C.: Software architecture in industrial applications. In: Software Engineering, 1995. ICSE 1995. 17th International Conference on. IEEE, pp. 196–196 (1995)
77. Kleppe, A.: First european workshop on composition of model transformations-cmt 2006 (2006)
78. Wegman, E.J.: Hyperdimensional data analysis using parallel coordinates. J. Am. Stat. Assoc. **85**(411), 664–675 (1990)
79. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In: OCL and Model Driven Engineering, UML 2004 Conference Workshop, vol. 12, pp. 69–83 (2004)
80. Inselberg, A., Dimsdale, B.: Parallel coordinates: a tool for visualizing multi-dimensional geometry. In: Proceedings of the 1st

conference on Visualization'90. IEEE Computer Society Press, pp. 361–378 (1990)
81. Myers, B.A., Ko, A.J., LaToza, T.D., Yoon, Y.: Programmers are users too: Human-centered methods for improving programming tools. Computer **49**(7), 44–52 (2016)
82. Guana, V., Stroulia, E.: Phydsl: a code-generation environment for 2d physics-based games. In: 2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM) (2014)
83. Guana, V., Stroulia, E., Nguyen, V.: Building a game engine: a tale of modern model-driven engineering
84. Tong, T., Guana, V., Jovanovic, A., Tran, F., Mozafari, G., Chignell, M., Stroulia, E.: Rapid deployment and evaluation of mobile serious games: a cognitive assessment case study. Procedia Comput. Sci. **69**, 96–103 (2015)
85. Ricca, F., Di Penta, M., Torchiano, M., Tonella, P., Ceccato, M., Visaggio, C.A.: Are fit tables really talking? In: 2008 ACM/IEEE 30th International Conference on Software Engineering. IEEE, pp. 361–370 (2008)
86. Ricca, F., Leotta, M., Reggio, G., Tiso, A., Guerrini, G., Torchiano, M.: Using unimod for maintenance tasks: an experimental assessment in the context of model driven development. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. IEEE Press, pp. 77–83 (2012)
87. Guana, V., Stroulia, E.: How do developers solve software-engineering tasks on model-based code generators? an empirical study design. In: First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS, pp. 33–38 (2015)
88. Burkhardt, J.-M., Détienne, F., Wiedenbeck, S.: Object-oriented program comprehension: effect of expertise, task and phase. Empir. Softw. Eng. **7**(2), 115–156 (2002)
89. Hermans, F., Aivaloglou, E.: Do code smells hamper novice programming? a controlled experiment on scratch programs. In: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on. IEEE, pp. 1–10 (2016)
90. Gravino, C., Risi, M., Scanniello, G., Tortora, G.: Do professional developers benefit from design pattern documentation? A replication in the context of source code comprehension. In: International Conference on Model Driven Engineering Languages and Systems, pp. 185–201. Springer, Berlin (2012)
91. OMG: MOF model to text transformation language (mofm2t), 1.0 (2008)
92. OMG: Meta object facility (mof) 2.0 query/view/transformation (qvt) (2015)
93. Di Ruscio, D., Kolovos, D., Matragkas, N.: Scalability in model driven engineering: Bigmde'13 workshop summary. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, ACM, p. 1 (2013)



Victor Guana is a Ph.D. Candidate in the Department of Computing Science at the University of Alberta. His main areas of research are model-driven engineering, human aspects of software engineering, and code analysis and verification. He received his B.Sc. and M.Sc. degrees from the University of Los Andes, Colombia. In 2013, he was a Visiting Scholar at the National Institute of Aerospace (NIA) and the NASA Langley Research Center. In 2015, he became a Killam Laureate.

Eleni Stroulia is a Professor with the Department of Computing Science at the University of Alberta. She holds M.Sc. and Ph.D. degrees from Georgia Institute of Technology. In 2009, she was awarded the NSERC/AITF Industrial Research Chair on Service Systems Management (w. support from IBM). Her research addresses industrially relevant software-engineering problems with automated methods. She is an internationally recognized expert in software design and analysis, web-based system development and service-oriented systems. She is a member of ACM, and IEEE.