Integration of Visual Contracts and Model Transformation for Enhanced MDE Development

Matt Selway, Georg Grossman, Markus Stumptner
University of South Australia
Adelaide, SA, Australia
<firstname>.<lastname>@unisa.edu.au

Kerryn R. Owen, Richard M. Dexter
Land Simulation, Experimentation, and Wargaming, Land Capability Analysis, Joint and Operations Analysis Division, Defence Science and Technology Group
Edinburgh, SA, Australia
<firstname>.<lastname>@dst.defence.gov.au

ABSTRACT

Model transformations are an important aspect of Model-Driven Engineering as models throughout the software development process are transformed and refined until, finally, application code is generated. However, model transformations are complex to build, maintain, and verify for correctness. We propose the combination of visual contracts, an implementation independent approach for specifying correctness requirements for verification purposes, with operator-based model transformation execution to integrate both the specification of transformation requirements and the transformations themselves within the same framework. The graphical operator-based notation is used to define both the constraints of a contract and the transformation definition. This allows reuse of operators between the two and maintains implementation independence as the operators can be directly executed or compiled to other model-transformation languages. To illustrate the concept, we report on a prototype integration of visual contracts with our our existing operator-based model transformation framework and applied it in the law-enforcement context to transform relational data sources into Elasticsearch for multisource analytics.

CCS CONCEPTS

• **Software and its engineering → Integration frameworks**; **Integrated and visual development environments**; *Visual languages*; *Specification languages*; • **Information systems → Extraction, transformation and loading**;

KEYWORDS

model-driven engineering, model transformation, transformation operators, visual contracts

ACM Reference Format:
Matt Selway, Georg Grossman, Markus Stumptner and Kerryn R. Owen, Richard M. Dexter. 2018. Integration of Visual Contracts and Model Transformation for Enhanced MDE Development. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239399

1 INTRODUCTION

A key aspect of Model-Driven Engineering is automated Model Transformation, where models are transformed and refined throughout the software engineering process until executable application code is produced. Equally important are model transformations for data integration in heterogeneous information environments, where, instead of code, the end result may be message exchanges comprising data serialised in XML, JSON, or other data formats.

One such scenario occurs in the Integrated Law Enforcement project within the Data to Decisions Cooperative Research Centre (D2D CRC)[1]. Law enforcement agencies are facing the problem of a large number of heterogeneous data sources within the agency and external sources that need to be integrated so they are able to extract data from multiple sources efficiently with a single query. For example, a police investigator working on a case needs to find all relevant information about an individual, which may require the extraction of information from the police person database, phone call or driving license records, the border security database, and others. Each data source is provided by a different department or agency and the data is likely to be in different structures and formats. One option to overcome this problem was to import various data sources into Elasticsearch[2] to establish a "data lake" so that queries can be performed on the integrated platform. Implementing the integration directly in a programming language like Java would be possible for one or two data sources, but in the case in question, more than 20 different sources can be available and often are not known up front. To accommodate these needs of the end-user agency required a model-driven approach that optimises reusability of integration components. Finally, the existing data sources, especially those outside the authority of the agency, may change at short notice, requiring near-instant upgrades to service interface versions. This required an approach that can verify whether the data sources have changed, and if so, to upgrade quickly.

Achieving these goals, and developing and using model transformations in general, is a challenging task as model transformations are complex to build and maintain; therefore, the ability to verify transformations for correctness is of the utmost importance. Most model transformation tools, however, do not provide capabilities to verify model transformations [13].

The Visual Contract framework PaMaMo [10] is designed as an implementation-independent method of specifying correctness requirements for Model-to-Model (M2M) transformations using a

[1]http://d2dcrc.com.au
[2]https://www.elastic.co/products/elasticsearch

design-by-contract [17] approach. The formal graphical nature of PaMaMo leads a to more concise and understandable representation of the preconditions, postconditions, and invariants than using only OCL [18] as well as lending itself to the analysis of the specified requirements for isses such as redundancies and conflicts.

On their own, however, Visual Contracts do not provide an executable transformation definition, they only specify the requirements such a definition should fulfil. Switching tools or environments to implement the transformation may slow development of the transformation definition. Moreover, the use of OCL to define constraints in the contracts is suited to specifying simple relationships between elements of the model, but not the complex data transformations that may be required between one model and another before its elements can be compared.

To resolve this, we propose the integration of Visual Contracts with a graphical pattern and operator-based model transformation architecture (the Metamodel-based Information Integration Architecture [4]) to provide a complete solution to specifying bidirectional transformations and their requirements. The two are a natural fit: both are pattern-based so the definition of the contracts provides the definition of the structural transformation, while the incorporation of reusable operators allows the same operators to be used for both complex data transformations and to describe the constraints of the contracts. The combination of the two within the same framework potentially leads to more rapid development and testing of model transformations. Moreover, the complete framework remains implementation independent as the operator-based constraints and transformation definitions can be viewed as abstract specifications that can be compiled to a target transformation language. This provides the capacity to perform development and testing within the integrated framework, followed by compilation to a high-performance model transformation engine to achieve large scale model transformations at industrial scale.

A prototype implementation of this integrated framework has been developed and applied in two domains: first, in the law enforcement context mentioned above and, second, for the transformation of behaviour models for execution by disparate simulation environments in the Defence context [22]. For simplicity and demonstration purposes, however, we illustrate the approach only on the law enforcement use-case. The remainder of this paper is as follows: Sections 2 and 3 introduce the Metamodel-based Information Integration Architecture and the Visual Contracts of PaMaMo, respectively, followed by a conceptual description of the combined architecture in Section 4; the prototype implementation is illustrated in Section 5; related work is discussed in Section 6; and the paper concludes with a discussion of future work in Section 7.

2 METAMODEL-BASED INFORMATION INTEGRATION (MMIIA)

The architecture proposed in [4] makes use of a joint/generic metamodel, a library of operators, and a library of mapping templates to perform (meta)model-based information integration. Figure 1 illustrates this architecture, comprising 3 levels of abstraction, in the context of the DoME[3] (meta-)modelling environment. Each level of abstraction (metamodel, model, and instance) is associated with particular tasks or roles in the integration process [4]. These are summarised in the following. Note that while we discuss source and target models for clarity, the patterns and transformations can be interpreted bidirectionally.

[3]http://dome.ggrossmann.com/

2.1 Metamodel Level: Domain Expert

At the topmost level, a *domain expert* uses their understanding of the application domain and the metamodels that are to be integrated (called *sub-metamodels*) to define a *joint* metamodel, i.e., a metamodel describing the similarities of the sub-metamodels. As such, the joint metamodel is both generic (wrt. capturing the common semantics of the sub-metamodels) and domain specific (wrt. the concepts and entities of relevance to users). This stage of integration typically requires that the domain engineer defines the sub-metamodels within the tool such that they are equivalent to the original metamodels on which they are based.

In the original description of the architecture (refer to [4]), the concrete operators used to transform instances were also defined by the domain expert in the joint metamodel. This approach has since been revised to externalise the operator library along the lines of [25] to improve reuse of operators across transformations and allow integration designers to create specific composite operators (see Section 2.2 below). In particular, operators for generating constants, accessing and setting attributes, and merging and splitting elements are predefined in the operator library.

2.2 Model Level: Integration Designer

The role of the *integration designer* is to populate the *template library* with the *mapping templates* that will be used to match fragments of the source models to be transformed into fragments of the target model, using the joint model as an intermediary. As such, the mapping templates instantiate aspects of the joint metamodel as well as the source and target metamodels [4].

The mapping fragments constitute the *structural* aspect of the mapping templates; however, the integration designer also specifies the *data transformation* aspect using the mapping operators. They can also create new composite mapping operators to achieve the required data transformations between the mapped fragments. These composite operators are stored in an *operator library* alongside the primitive operators and domain specific specialisations of the primitives. This supports re-usability of operators at different levels of granularity across many mapping templates. For example, a specific property, e.g., 'name', may be retrieved regularly, so a specialisation of the primitive 'read property value' operator can be stored in the library; this 'read name' operator can then be reused across mapping templates and within composite operators. Similarly, a common data transformation may be to convert the 'name' of an entity into `CamelCase`. This could be defined as an operator comprising: 'read name'→'convert to title case'→'remove spaces'→'store name'.

This approach allows mapping templates to include schema and instance information, thereby allowing data to be mapped across modelling levels [4]. Such flexibility is a necessity in many transformation scenarios where the metamodels being mapped represent information at different levels.

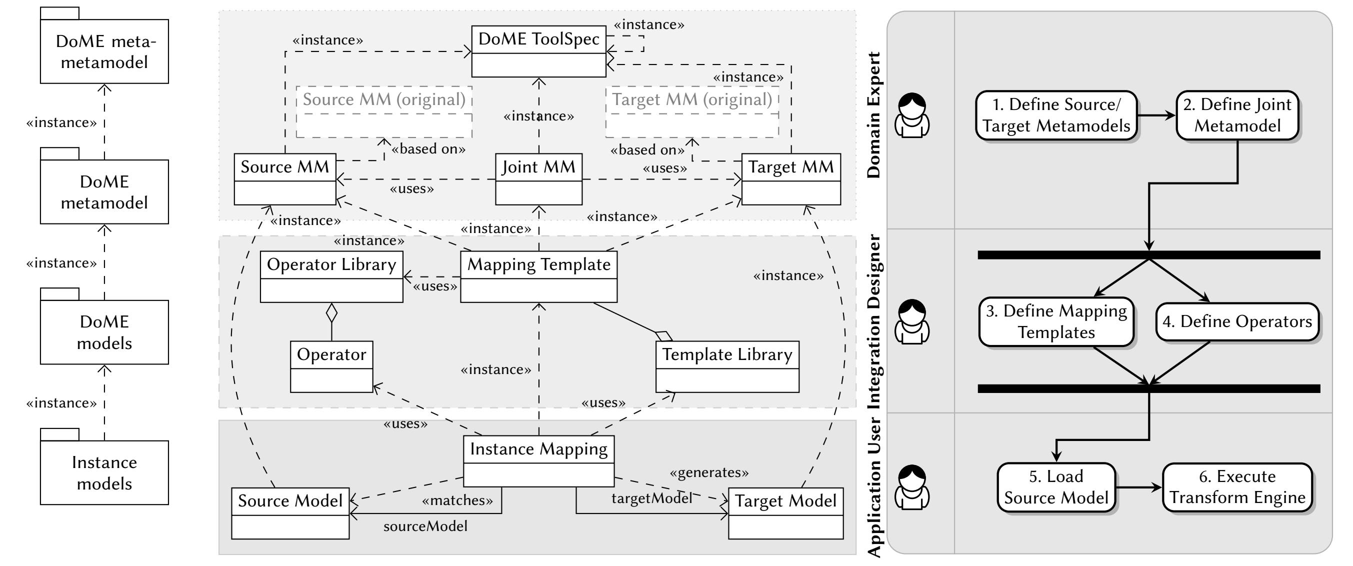

Figure 1: Metamodel-based Information Integration Architecture (MMIIA), adapted from [4]

2.3 Instance Level: Application User

At the instance model level, the *application user* loads a source model and initiates the transformation process [4]. The transform engine searches the template library for matching templates, instantiates the target elements, composes the model from multiple matches, and provides the result to the user for review and refinement. When ready, the user may save or export the model to the required format.

3 VISUAL CONTRACTS

While the MMIIA has been used in the context of industrial information integration and exchange (see [4]), building, maintaining, and verifying the transformations at a large scale remains a challenge. To help resolve this, we investigated the integration of visual contracts into the framework.

Visual Contracts [10] support the specification of the requirements of a transformation in an implementation independent manner using a visual notation (called PaMaMo) and OCL constraints. Such contracts enable the verification of the implemented model transformations (before and after transformation occurs), support their creation and maintenance, and provide documentation. In particular, contracts assist the testing of model transformations—an important issue that is lacking support in most model transformation tools [13]—by enabling the automated checking of an output model against the expected result defined by the contracts [10].

PaMaMo allows the definition of *preconditions* on a source model, *post-conditions* on the target model, and *invariants* that apply across both models. Moreover, the language provides additional constructs (*enabling/disabling conditions*) to control when a precondition, post-condition, or invariant should come into effect. The different constructs of PaMaMo have been given a formal semantics, which allows the contracts to be analysed for redundancies, contradictions, pattern satisfaction, and coverage [10]: all of which are important aspects of model transformations tools that are still missing in most cases [13].

By defining such a declarative, visual language, PaMaMo aims to be transformation language independent [10]. The intent is to allow the definition of model transformations in an arbitrary language, e.g., QVT Relations [19], and compile the contracts to the same language so they can be performed in a transformation chain along with the transformations implementing the contracts. Figure 2 illustrates the architecture of PaMaMo in an automated verification use case. As with MMIIA, the contracts of PaMaMo are bidirectional, but are described conceptually with a source and target.

The following summarises the elements of PaMaMo contract specifications, with reference to the notation in Figure 3[4]:

Rules (1, 2) are defined declaratively and comprise visual contracts. They can be either *positive* (1) or *negative* (2). Positive rules represent situations that are necessary for the model(s) to be considered valid, while negative patterns state situations that are forbidden to occur. There are three types of rules, depending on whether their source and target patterns are present/empty or not: preconditions, postconditions, and invariants. A contract is considered satisfied iff all rules in the contract are satisfied. *Preconditions & Postconditions* are rules that contain only a source or target pattern, respectively. Whether a rule is one or the other depends on the direction of execution. For example, the negative rule (2) would be a postcondition when executing left-to-right, but a precondition when executing in reverse. The semantics of preconditions/postconditions are defined as:

$$\begin{array}{ll} \mathit{Positive}: & \exists o \in \mathit{Occ}(P,m) : \mathit{Expr}(o) \\ \mathit{Negative}: & \nexists o \in \mathit{Occ}(P,m) : \mathit{Expr}(o) \end{array}$$

where P is the pattern (source or target) of the rule, m is a model, $Occ(P,m)$ is the function $Occ : \mathbb{P} \times M \mapsto \mathbb{PI}$, representing the

[4]For brevity we describe the notation implemented in our tool prototype, which differs slightly from the description in [10]

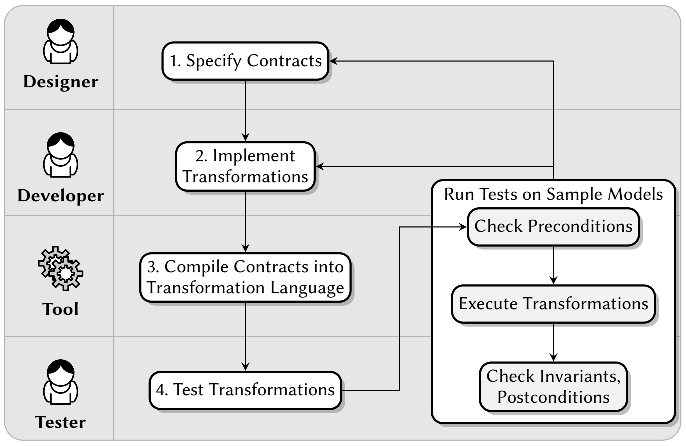
Designer
Developer
Tool
Tester
1. Specify Contracts
2. Implement Transformations
3. Compile Contracts into Transformation Language
4. Test Transformations
Run Tests on Sample Models
Check Preconditions
Execute Transformations
Check Invariants, Postconditions

Figure 2: PaMaMo workflow for automated verification (adapted from [10])

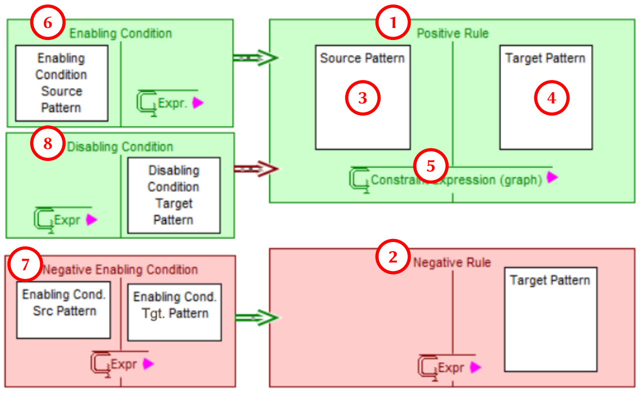
6
Enabling Condition
Enabling Condition Source Pattern
Expr.
8
Disabling Condition
Disabling Condition Target Pattern
Expr
1
Positive Rule
Source Pattern
3
Target Pattern
4
5
Constraint Expression (graph)
7
Negative Enabling Condition
Enabling Cond. Src Pattern
Enabling Cond. Tgt. Pattern
Expr
2
Negative Rule
Target Pattern
Expr

Figure 3: Visual Contract notation, based on PaMaMo

occurrences (or instances) of a pattern P in model m, given the set of patterns $\mathbb{P}$, the set of models M, and the set of pattern instances/occurrences $\mathbb{PI}$. $Expr(o)$ represents the application of the rule's constraint expression on a pattern occurrence; an empty expression evaluates as true.

Invariants are rules that contain both a source and target pattern. They are evaluated with respect to instances of both models, regardless of direction. However, an invariant only "matches" if the part of the expression relevant to the source pattern is satisfied. The semantics are defined as:

$$\begin{aligned}
\textit{Positive}: \quad & \forall o_{src} : o_{src} \in Occ(P_{src}, m_{src}) \wedge Expr|_{src}(o_{src}) \implies \\
& \exists o_{tgt} : o_{tgt} \in Occ(P_{tgt}, m_{tgt}) \wedge Expr(o_{src} \cup o_{tgt}) \\
\textit{Negative}: \quad & \forall o_{src} : o_{src} \in Occ(P_{src}, m_{src}) \wedge Expr|_{src}(o_{src}) \implies \\
& \nexists o_{tgt} : o_{tgt} \in Occ(P_{tgt}, m_{tgt}) \wedge Expr(o_{src} \cup o_{tgt})
\end{aligned}$$

where P_{src} is the source pattern of the rule, m_{src} is the source model, P_{tgt} and m_{tgt} are the target pattern and model, respectively, and $Expr|_{src}$ is the constraint expression reduced to include only source elements, properties and variables. A consequence of this definition is that the rule is satisfied if either no occurrences of the source pattern are found or if no occurrences satisfy the reduced constraint expression.

Source and Target Patterns (3,4) specify the entities, associations, and attributes to be matched against a model during execution. The patterns are defined as instances of the source or target metamodel but may contain variables as well as concrete values. Variables may be referenced in multiple locations to specify equality and/or be used in the rule's constraint expression.

Constraint Expressions (5) define additional constraints not captured by the patterns themselves. Moreover, constraints allow pattern elements to be related within and across models as well as with more complex relations than just variable substitution-based equality. In [10], constraint expressions are defined using OCL.

Enabling Conditions (6, 7) control when a rule should be applied, much like a guard condition in textual transformation languages such as QVT relations [19] and ATL [12]. Like rules, enabling conditions specify source and/or target patterns, an optional constraint expression, and may be positive or negative. As an extension to [10], we allow enabling/disabling conditions to be positive or negative. An enabling condition acts as an implication, restricting the situations in which its associated rule can be satisfied to only those that satisfy the enabling condition itself. The basic formulation for the semantics of an enabling condition on an invariant is:

$$\begin{gathered}
\forall en_{src}, en_{tgt}, o_{src} : en_{src} \in Occ(P^{\text{EN}}_{src}, m_{src}) \wedge \\
en_{tgt} \in Occ(P^{\text{EN}}_{tgt}, m_{tgt}) \wedge o_{src} \in Occ(P_{src}, m_{src}) \wedge \\
(Expr^{\text{EN}} \cup Expr|_{src})(en_{src} \cup en_{tgt} \cup o_{src}) \implies \\
\exists o_{tgt} : o_{tgt} \in Occ(P_{tgt}, m_{tgt}) \wedge Expr(o_{src} \cup o_{tgt})
\end{gathered}$$

where P^{EN}_{src} and P^{EN}_{tgt} are the source and target patterns of the enabling condition, respectively, and $(Expr^{\text{EN}} \cup Expr|_{src})(\dots)$ is the combination of the enabling condition's constraint expression and the source part of the rule's constraint expression applied to the combined occurrences of the enabling condition's source and target patterns plus the source pattern of the main rule. By evaluating the combined conditions against the combined occurrences in this way, the constraint expression of the main rule can relate elements to those of the enabling condition.

Note that there can only be one enabling condition per rule.

Disabling Conditions (8) contrast with enabling conditions by preventing a rule from being evaluated when the disabling condition is satisfied. This allows rules to be excluded in certain contexts. The combined schema for enabling and disabling conditions is:

if enabling and (not disabling$_1$) and (not disabling$_{\dots}$)
and (not disabling$_n$) then evaluate main rule

Variable/Pattern Sets (not shown for brevity) can be defined in source/target patterns, allowing the matching of arbitrary subgraphs within a pattern and, when matched, represent the set of all occurrences of the subgraph. Hence, rules can be defined based on the number of times the subgraph occurs within its context (possibly the entire model). Moreover, variable sets can be nested.

4 APPLYING VISUAL CONTRACTS TO MMIIA

Individually, the two architectures, MMIIA and PaMaMo, are complementary. MMIIA provides a flexible framework for defining model transformations using graphical mapping templates and a reusable operator library, but does not provide adequate verification and testing features. In contrast, Visual Contracts provide a means of specifying verification properties and performing automated testing, but do not provide the transformation capability. Therefore, we merge these two frameworks to provide a complete model transformation solution that can achieve both functionalities.

Additionally, we introduce reusable operators to the constraint expressions, allowing a graphical operator-based representation of the constraints. This enhances the implementation-independent aspect of the visual contracts by not requiring OCL for their definition. Moreover, it allows the constraints to more easily incorporate complex relations between the elements of the source and target models. Finally, we define the source and target patterns using the domain specific notation of the source and target metamodels, rather than a generic notation. This makes it easier for domain engineers to understand and validate the mappings.

In the context of the MMIIA, Visual Contracts specify the requirements of each mapping template. The semantics of the contracts, through enabling/disabling conditions, also enhance the mapping templates with additional controls for when a template should match an input model. Conversely, the transformation definition of the mapping templates provide transformation execution capability to the rules of the Visual Contracts. Furthermore, the integration of these two frameworks preserve the elements of the two ideas wrt. the user roles involved in the development of contracts and transformations as well as the implementation independent nature of the contracts: the operators can be compiled into a target language like the patterns themselves.

The integrated workflow is illustrated in Figure 4. At the metamodel level, the domain expert defines the joint metamodel along with the metamodels for the source and target. Furthermore, they define high-level contracts that the integration designer will use as a guide to develop the transformation definitions at the model level. Since each rule of a contract specifies the basis of a mapping template, the integration designer incorporates the transformation definition into the templates defined by the domain expert, while defining any operators required to complete the transformation. The integration designer may also define new mapping templates (contract rules) and their transformation definitions to enhance modularity and reusability, or refine the high-level contracts to take into account more detailed requirements. For example, the domain expert may define a rule that indicates the name of two mapped elements must be equal, while the integration designer identifies that a naming convention is required and refines the contract to ensure that it is the `CamelCase` variant of the name that is considered equal.

Following the Visual Contracts approach, the contracts and their transformation definitions can be compiled into a target transformation language. However, since the contracts and operator-based constraints and transformations are fully executable, this step may be skipped. By allowing direct testing of transformations, development time can be improved, while the possibility of compilation supports the deployment of the transformations to a high-performance transformation engine for large-scale applications.

Next, the transformations are tested to ensure their correctness by leveraging the capabilities of the Visual Contracts. Errors identified in the testing process may lead to corrections to any of the joint metamodel, the contracts and/or, the transformation definitions. Finally, the transformation(s) can be executed by an end-user by loading the desired source model(s), triggering the transformation engine, revising the output model if desired, and storing the result.

5 APPLYING THE COMBINED FRAMEWORK

The proposed integrated framework has been implemented as a prototype using the DoME metamodelling environment in VisualWorks Smalltalk. The implementation mainly involved the addition of the Visual Contracts framework to the existing operator-based transformation framework and the addition of comparison operators to support the definition of constraints. To illustrate the framework, we describe its application to a (simplified) use case from the Integrated Law Enforcement project within the D2D CRC.

In the use case, a relational database containing person information must be integrated with Elasticsearch. The source of the transformation comprises a simple relational database schema containing two tables: Person and Address. The Address table refers back to the Person table to associate (possibly many) addresses to individual people. Moreover, the address may be specified as **billing** or **mailing** addresses. The schema with some sample data is illustrated in Figure 5: all values are represented by strings.

The target is a JSON[5] fragment to be ingested by the Elasticsearch search engine. For Elasticsearch, each Person is a top-level document containing a set of nested Address objects. An example JSON fragment representing a person is displayed in Listing 1.

Both the source and target describe people and their (billing and/or mailing) addresses. There are several differences that require different forms of transformation to handle:

(1) structural differences, e.g., the DB separates Person and Address with a link between them, while the JSON nests the Addresses within each Person; single **name** attribute in the DB vs. **given_name**, **last_name** pair in JSON; single **street address** attribute in JSON vs. **number**, **street** pair in DB
(2) main types represented by tables vs. the **_type** attribute
(3) differences in attribute names: e.g, **id** vs. **le_id**, **date-of-birth** vs. **dob**
(4) differences in data types, namely: **date-of-birth** is a string in the format "YYYY-MM-DD", while **dob** is a long integer representing the date as the "number of milliseconds since the epoch".

5.1 Defining the Joint Metamodel

To help develop the transformation and support the integration of other source/target models, we first need to develop the joint metamodel and the source/target metamodels. For brevity, we focus on the joint metamodel as we have generic RDBMS and JSON metamodels that can be used for such simple schemas in the source/target.

[5]https://json.org/

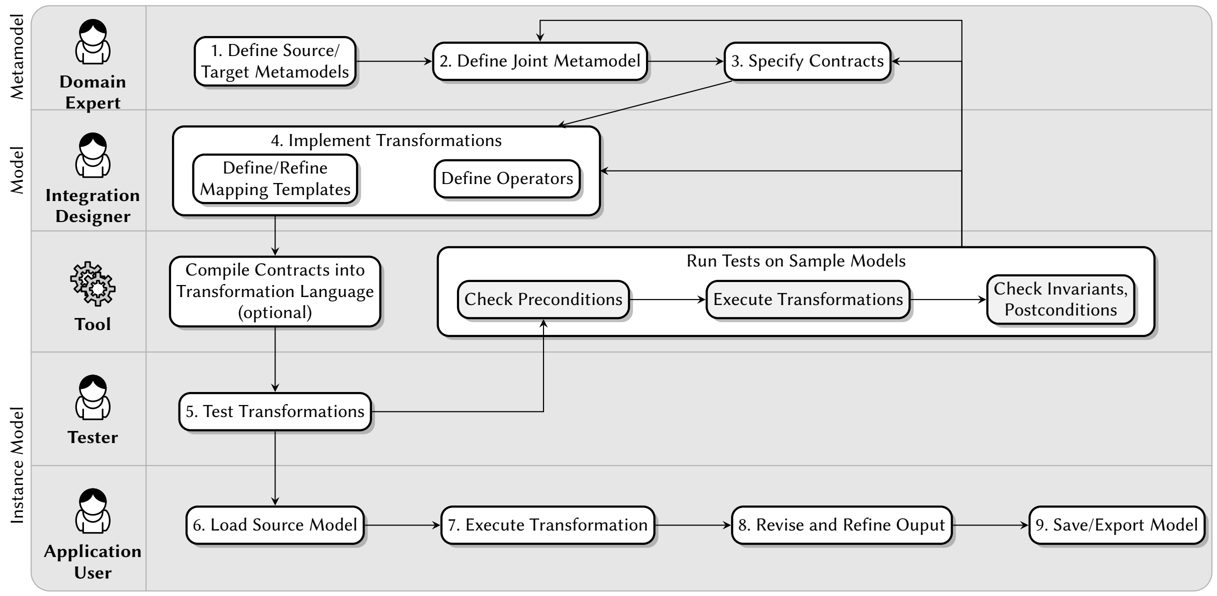
Metamodel
Model
Instance Model
Domain Expert
Integration Designer
Tool
Tester
Application User
1. Define Source/ Target Metamodels
2. Define Joint Metamodel
3. Specify Contracts
4. Implement Transformations
Define/Refine Mapping Templates
Define Operators
Compile Contracts into Transformation Language (optional)
Run Tests on Sample Models
Check Preconditions
Execute Transformations
Check Invariants, Postconditions
5. Test Transformations
6. Load Source Model
7. Execute Transformation
8. Revise and Refine Ouput
9. Save/Export Model

Figure 4: Combined workflow of MMIIA and Visual Contracts

Person

id	name	title	date-of-birth
1234567890	Selway, Matt	Dr	2018-01-01
2345678901	Grossmann, Georg	Dr	2018-01-01

Address

person id	kind	number	street	city	postcode	state	country
1234567890	billing	10	At Home Cr	Adelaide	5000	SA	Australia
1234567890	mailing	1-100	University Blvd	Mawson Lakes	5095	SA	Australia

Figure 5: Simple Person-Address DB Schema and sample data.

```
"_source": {
  "le_id": "1234567890",
  "profile": {
    "_type": "Person",
    "title": "Dr",
    "last_name": "Selway",
    "given_name" "Matt",
    "dob": 3692217600000,
    "addresses": [
      {
        "_type": "Billing_Address",
        "street address": "10 At Home Cr",
        "city": "Adelaide",
        "state": "SA",
        "postcode": "5000",
        "country": "Australia"
      },
      {
        "_type": "Mailing_Address",
        "street address": "1-100 University Blvd",
        "city": "Mawson Lakes",
        "state": "SA",
        "postcode": "5095",
        "country": "Australia"
      }
    ]
  }
}
```

Listing 1: Example JSON fragment represent and person and their addresses for the Elasticsearch target

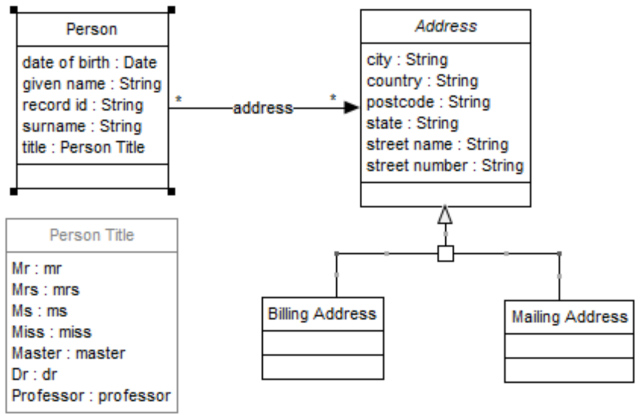

Figure 6: Joint "Person" Metamodel

As discussed, the joint metamodel is domain specific, therefore it describes people and addresses, while being generic wrt. the models being integrated. Figure 6 displays the metamodel defined using DoME's metamodelling capabilities. It describes four classes: **Person**, **Address**, **Billing Address**, and **Mailing Address**. The latter two are subclasses of **Address**, which is an *abstract* class. We add a many-to-many association, **address**, from **Person** to **Address**, allowing addresses of any type to be associated with a person(s).

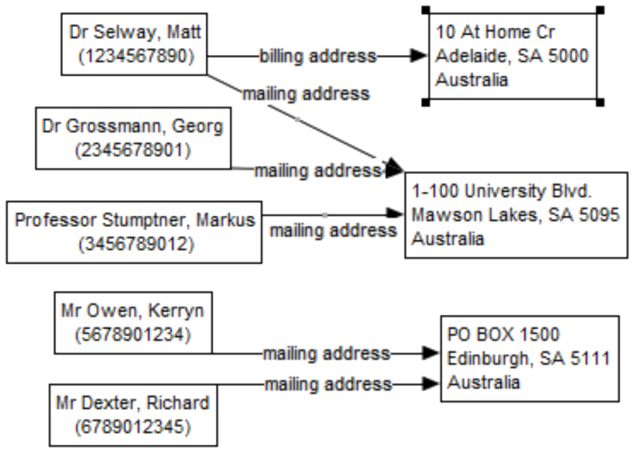

Figure 7: Example instance of the "Person" joint metamodel

We also define an appropriate set of attributes to model those required by all source/target metamodels. Like the original metamodel, many of these attributes are simply strings; however, the **date of birth** is defined to be of type **Date** (rather than a string or a long integer) and **title** is defined with an enumeration (**Person Title**) containing the allowable title values of each dataset. This demonstrates the importance of the joint metamodel as normalising the data of each source/target model. If a transformation were to be defined directly from the DB to the JSON, the date of birth, for example, would likely need to be passed through a date datatype on an ad-hoc basis to perform the transformation correctly.

An example instance model is shown in Figure 7. It uses a *domain specific* notation, rather than a generic object notation, that makes the information about a **Person** and an **Address** readily visible, with the link between a **Person** and an **Address** labelled with the type of address at the destination. This representation has the benefit of reducing the duplication of addresses, as both the source and target schemas duplicate the same address for different people.

5.2 Specifying Contracts

At this point the domain engineer who developed the joint metamodel can also specify the contracts to which the transformation implementation must adhere. Figure 8 shows an invariant for mapping PERSON in the database to **Person** in the joint metamodel (8a) including the source/target pattern definitions (8b, 8c), and a portion of the constraint definition (8d). The source and target patterns each have a sub-diagram containing the actual pattern definition. Our implementation of Visual Contracts uses ports (shown as small pink triangles) to configure the interface for the data that is extracted from a pattern and that is made available to the constraint and the transformation. In this case we are extracting data from each of the attributes of the respective models; however, in more complex use cases only a portion of the attributes are ever used, especially as we use the domain specific notation which, by necessity, includes all of the attributes defined for any particular object included in the pattern.

Elements of the pattern definitions are connected to the ports of the pattern through dataflows to boundary nodes in the subdiagram (the large triangles connected to circles, cf. 8b & 8c). The RDBMS pattern includes an element representing the table, with a subelement for each field of the table, while the target pattern includes an instance of **Person** from the joint metamodel with ports accessing its attributes. The *field* elements of the RDBMS pattern and the *ports* on the instance of **Person** are specialisations of the read/store operators embedded in the pattern notation. Moreover, they perform both roles depending on the execution direction.

The constraint for the rule also defines ports for the data used within its definition (subdiagram). Describing constraints requires the extension of the MMIIA operator library to include comparison operators, such as 'equivalence', 'less than', etc., as well as logical operators, 'and', 'or', etc. A portion of the definition is shown in 8d, which illustrates the constraint implemented with a series of equality tests (diamonds containing '=') joined by a logical conjunction operator (circle containing 'AND'). Most importantly, this constraint uses an operator from the *operator library* that splits the single **name** attribute of the PERSON table into 'surname' and 'givenName' to be compared to the attributes of the **Person** class with the same name. Figure 8e is a snapshot of the operator library (called the *Shelf* in DoME) containing the definition of the operator. Composite operators ('Complex Transformation' category) are also defined and the constraints themselves appear as reusable elements: a constraint is simply a type of operator that returns true/false.

Since the values of **name** in the database may not necessarily conform to the pattern '⟨surname⟩, ⟨given names⟩', a precondition could be defined over the PERSON table to ensure that it conforms and so prevent the failure of the transformation. Preconditions could also be defined for other aspects, e.g., the format of the **date-of-birth** string if such validation is not performed by the RDBMS or the client application. To complete the DB-to-joint metamodel contracts, an Address-to-Address rule is specified similar to Person-to-Person. By defining the invariant such that it ignores the PERSON to whom the ADDRESS belongs, we can ensure the joint model contains all of the addresses without duplication. Moreover, this invariant uses the 'generalisation class' property in the pattern for **Address** on the joint metamodel side. This is necessary as **Address** is defined as *abstract*, hence, the concrete notation cannot instantiate **Address** itself. Instead, one of the concrete subclasses is instantiated as a placeholder and its 'generalisation class' property is set to the desired superclass: **Address** in this case.

A third invariant is defined with an enabling condition that matches both **Person** and **Address** for context on each side. The invariant then uses that context to ensure that an association exists between the two in the target if it is present in the source.[6] Finally, a set of similar invariants are defined between the joint metamodel and the JSON. The constraints involve a different set of operators to perform the data transformations required between the joint metamodel and the JSON definition. For example, the date of birth will be transformed from long integer, representing milliseconds, to a proper Date datatype, and the subclasses of **Address** will

[6]An alternative would use a single invariant to enforce the entire person/address structure using a pattern set; however, in more complex mappings this quickly becomes unwieldy. Breaking it down this way simplifies the definitions and allows the reduction of duplication during the transformation to the joint model.

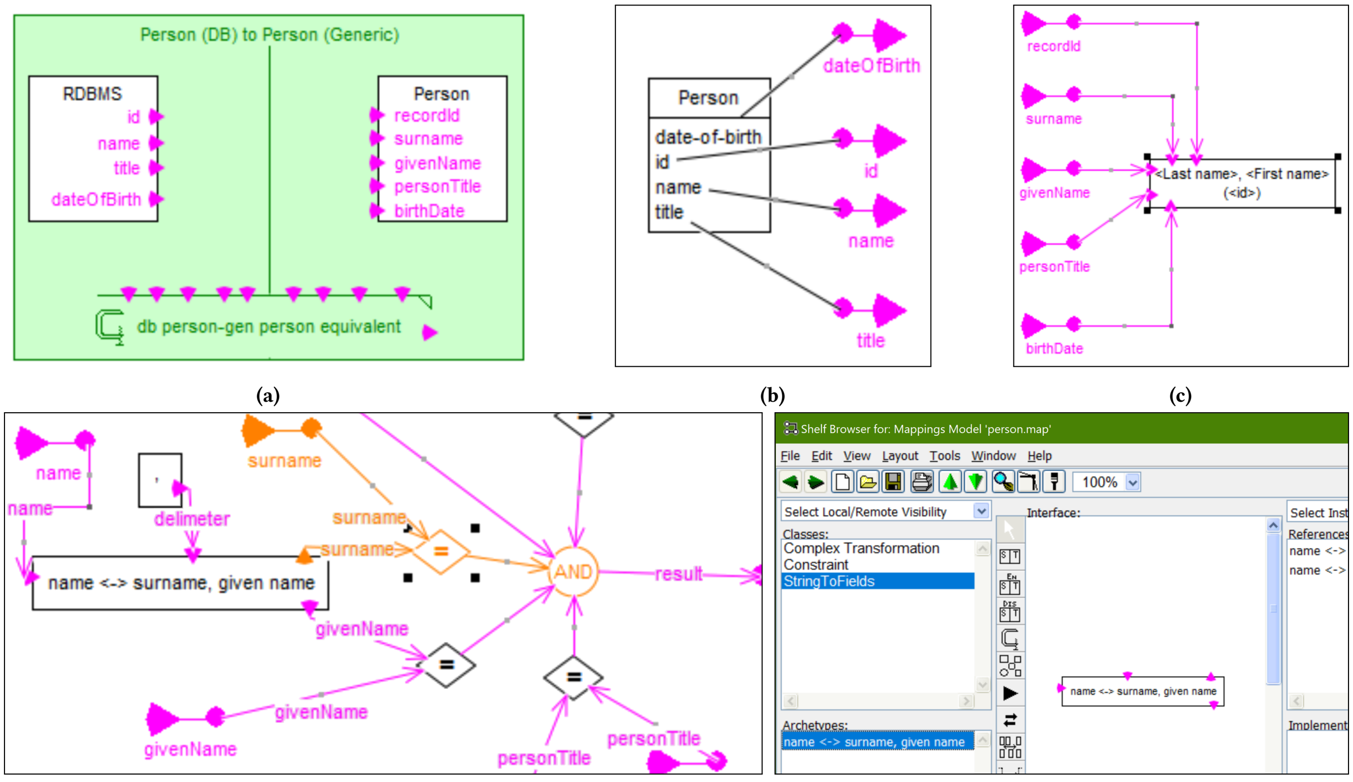

Figure 8: Person-to-Person Contract (a); source pattern, DB (b); target pattern, joint metamodel (c); constraint definition (d); operator library (e)

have their names appropriately modified to be matched against the names 'Billing_Address' and 'Mailing_Address', among others. All operators defined by the domain engineer at this stage will be available for the integration designer to use when specifying the model transformations themselves in the next step.

5.3 Implementing the Transformations

The integration designer adds the transformation definitions themselves to the contracts defined by the domain expert. This enables the precondition, postconditions, and invariants to be executed in the context of a transformation. The transformation definition for the 'Person (DB) to Person (Generic)' invariant is shown in Figure 9. It illustrates the straight copy of the **id** attribute, the use of the same operators from the library as used by the constraint, and the addition of the bidirectional operator 'string ↔ PersonTitle', which is a new operator defined to ensure the bidirectional transformation to/from the enumeration of titles.

The execution semantics in the context of a transformation are such that the result of the transformation must satisfy the precondition/postcondition/invariant on completion of the transformation (assuming the correct definition of the transformation). For example, a positive invariant transformation would instantiate the elements of the target pattern and assign values to their attributes according to the result of the data transformation. If the invariant is violated after the transformation completes, the transformation is incorrect.

More formally:

$$\forall o_{src} : o_{src} \in Occ(P_{src}, m_{src}) \wedge Expr|_{src}(o_{src}) \implies \\ \exists o_{tgt} : o_{tgt} = copy(P_{tgt}, m_{tgt}) \wedge Expr(o_{src} \cup o_{tgt})$$

where $copy(P_{tgt}, m_{tgt})$ represents a new instance of the pattern in the target model, as opposed to a matched occurrence already present. This means that, for a basic invariant, the entire target pattern is instantiated each time. The context of an enabling condition helps control the parts of a pattern that are instantiated. If the enabling condition contains a target pattern and the target pattern of the main rule incorporates some (or all) of the context, only the part *not* included in the context is instantiated. That is,

$$\forall en_{src}, en_{tgt}, o_{src} : en_{src} \in Occ(P^{EN}_{src}, m_{src}) \wedge \\ en_{tgt} \in Occ(P^{EN}_{tgt}, m_{tgt}) \wedge o_{src} \in Occ(P_{src}, m_{src}) \wedge \\ (Expr^{EN} \cup Expr|_{src})(en_{src} \cup en_{tgt} \cup o_{src}) \implies \\ \exists o_{tgt} : o_{tgt} = copy(P_{tgt} \setminus P^{EN}_{tgt}, m_{tgt}) \wedge Expr(o_{src} \cup o_{tgt})$$

The execution of negative rules as transformations are similarly applied; however, instead of instantiating the target pattern, the matching occurrence is deleted from the target model (excluding any context matched by an enabling condition, if present). For example, the transformation of a negative invariant is defined as:

$$\forall o_{src}, o_{tgt} : o_{src} \in Occ(P_{src}, m_{src}) \wedge o_{tgt} \in Occ(P_{tgt}, m_{tgt}) \wedge \\ Expr(o_{src} \cup o_{tgt}) \implies delete(o_{tgt}, m_{tgt})$$

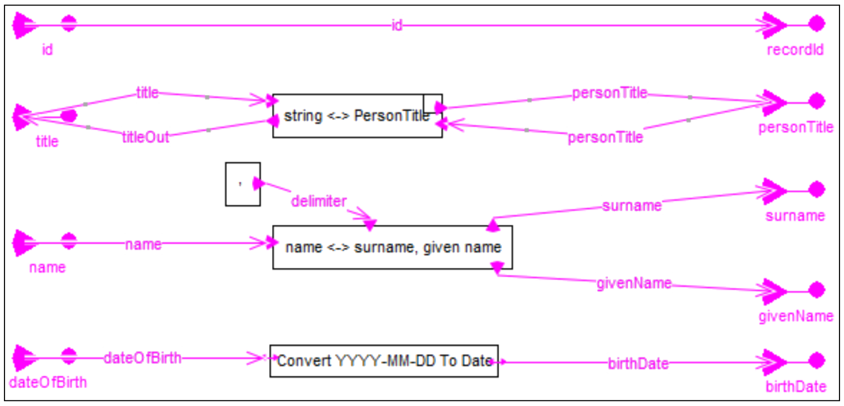

Figure 9: Transformation definition for the Person (DB) to Person (Generic) invariant

The result of which is a model state that would satisfy the negative invariant. In some contexts, these semantics may have an unexpected side-effect: depending on the direction of the transformation, elements will be deleted from the different models. This is not an issue for in-place/refactoring transformations, which are always executed left-to-right. In heterogeneous transformations, however, while the invariant will be satisfied by deleting from either model, the deletion of elements from the opposite model may not be the intended result. For example, in round-trip engineering it may be undesirable to lose elements from the original model. This can be controlled by combining an invariant with no transformation (i.e. the original invariant defined by the domain expert) with negative pre-/post-conditions that have enabling/disabling conditions. The invariant will report if it is ever violated, while the pre-/post-conditions provide the integration designer with finer control over which model is updated during transformation.

Unlike other approaches that define bidirectional transformations either by being completely declarative (with limitations in the types of data transformations that can be performed) or by specifying a pair of independent transformations, one for each direction, this framework allows bidirectional transformations to be defined at various levels of granularity. The declarative nature of the patterns, i.e. the structural transformation, are bidirectional by their nature and the primitive operators have opposites (read/store, split/merge, etc.). Not all complex data transformations are naturally bidirectional, though. For example, a merge between two strings in one direction may not be automatically detectable as to where the string should be split in the opposite direction. In this case, only the part of the transformation that is *not* automatically reversible need be defined in the reverse direction. This can be seen in Figure 9, where the composite operator 'string ↔ PersonTitle' internally defines a transformation in each direction—visible by the two sets of input/output ports—as the operators used to convert the string into a valid enumeration value are different to those for converting the enumeration value to a string. As a result, the entire operator becomes bidirectional and can be reused. Furthermore, this approach lends itself to "user-interaction" operators that can prompt the user for the appropriate transformation and remember the result for future executions.

5.4 Testing the transformations

Once the integration designer finishes implementing the transformations, the tester can ensure they adhere to the contracts defined by the domain expert. First of all, the contracts can be analysed according to the rules defined in [10] to report on redundancies, contradictions, pattern satisfaction, and coverage.

Next the transformations can be tested by executing them on sample models. The mappings can be executed in either 'check-only' mode, where the source and target models are simply checked for violations, or transformation mode, where the target model is updated according to the described semantics. If any preconditions, postconditions, and/or invariants are violated by the models (either as the result of the transformation or in 'check-only' mode), the locations of the violations are reported to the tester. Additionally, the traceability information generated by the transformations can be inspected by the tester to further identify what or where the transformation erred. The tool also makes it possible to step through the transformation interactively and inspect the data input/outputs of each operator as it executes, which is invaluable to the tester attempting to debug a particularly troublesome transformation.

6 RELATED WORK

6.1 Model Transformations

There are many approaches to model transformations: a recent survey by Kahani et al. [13] identified 60 active tools that support model transformation. The approaches based on graph transformations (i.e. algebraic graph transformation [2], or triple-graph grammars [21]), such as [5, 7, 9, 20], are closest to what we describe here in that they typically use graphical, pattern-based transformation definitions and support verification due to their formal foundations. In general, they define a left-hand side pattern, a right-hand side pattern, and (possibly) a negative application condition. This is similar to the source pattern, target pattern, and disabling conditions of the rules of Visual Contracts; however, patterns are specified using only the abstract syntax or a generic graph representation while our approach uses the concrete syntax and supports enabling conditions to further control the context in which a rule is applied. Moreover, graph transformation approaches typically require the user to manually define traceability and focus on structural transformations not complex data transformations, whereas our approach incorporates automated traceability link creation and a dataflow-/operator-based representation to handle data transformations. In addition, [20] and [7] support only unidirectional transformations, while [5] and [9] support bidirectional transformations.

VIATRA2 [24] is a model transformation tool for the Eclipse Modelling Framework (EMF)[7] that has a hybrid declarative/imperative model transformation language based on graph transformations and abstract state machines. As a result it also provides verification capabilities. However, it has only a textual syntax to specify transformations, and its transformation definitions are unidirectional.

The Visual Model Transformation Language (VMTL) [1] uses annotations on models to define the transformation. This approach leverages the modelling notation that is the subject of the transformations by utilising its comment/annotation capability. As a result, transformations can be defined using the original modelling notation. However, the annotations are textual and designed for in-place, i.e. refactoring, transformations within a single model; hence, VMTL does not extend to heterogeneous model transformations.

DSLTrans [3] is a visual transformation language for EMF that can use contracts for verification via the SyVOLT extension [16]. SyVOLT leverages restrictions in DSLTrans to provide exhaustive verification of model transformations. Contract pre-/post-conditions can be specified using a visual notation similar to that of [10] and allows propositional connectors between attributes, e.g., for equality. In contrast to the approach presented here, SyVOLT contracts are specified separately from the transformation itself uses its own concrete syntax rather than that of the models for which the transformation and contracts are being developed.

6.2 Visual Constraints

There has been some previous work in visual representations of constraints for object-oriented models. Kent's *constraint diagrams* [14] used a mainly graphical notation to depict constraints that was intended to complement UML-based diagrams or other Object-Oriented modelling notations. Based on Venn-diagrams and object models, it allowed the visualisation of constraints on (typed) sets of objects through arcs representing navigation across associations. Amongst its limitations was the inability to represent existential quantification nor constraints on primitive values such as integers. Extensions to constraint diagrams, such as [8, 11], resolved these limitations at the expense of making the diagrams more complex. While the result is more expressive, the ability to specify constraints involving complex data transformations is limited.

Other approaches to visual constraints, such as VisualOCL [15] and the Extensible Visual Constraint Language [6], wrap an otherwise textual notation in graphical blocks to make them easier to read. While such a breakdown into compartments helps separate what would otherwise be large chunks of text, and in the case of VisualOCL allows the inclusion of aspects of object diagrams, the notations are predominantly text-based.

The Visual Model Query Language (VMQL) [23] uses text-based comments or annotations within the notation of the models for which constraints are being added. This is much like the standard use of OCL in the comments of UML Class Diagrams. The approach differs in its use of variables and other features within the diagrams themselves, which then relate to content of the constraints. This allows the definition of pattern-like constraints; however, is less expressive than full OCL [23].

[7]http://www.eclipse.org/modeling/emf/

One feature common to all of these approaches to visual constraints is that they focus on constraints *within* a model, not *between* models of different types. They focus on equivalences, non-equivalences, and other relational constraints within the model, but do not factor in data transformations that may need to occur between compared elements. For example, the 'name' of something in one model may not be directly equivalent to the name in another model, but rather it will be after modification, such as whitespace removal. In contrast, our use of data flow and operator-based constraints is amenable to the definition of constraints between models of the same or different types while supporting the necessary data transformations to perform accurate comparisons.

7 CONCLUSION

In this paper we have proposed and demonstrated an integrated model transformation framework based on Visual Contracts that allows rapid development and testing of model transformations. The integrated framework maintains the benefits of the two original architectures: (1) pattern- and operator-based transformations with well-defined activities for the different user roles engaged in the development of model transformations, and (2) language-independent verification and testing capabilities for model transformations using a clear graphical notation. In addition, it supports the definition of complex data transformations, not only structural transformations, in both the transformations and the constraints of the contracts. The operator-based approach allows the entire transformation, not only the contracts, to be language independent, allowing for transformations to be deployed to high-performance transformation engines after testing within the integrated framework.

We have demonstrated the implementation on a simplified industrial case study from a D2D CRC project on integrated law enforcement. Moreover, the framework is currently in use in the Defence context, where behaviour models are transformed into executable models for different simulation environments.

Future work will investigate the analysis of the rules and the transformation definitions to automatically derive a direct transformation from one submodel to another, rather than having to perform a multi-stage process where one submodel is transformed to the joint model followed by another transformation to the target model. Combined with compilation to a high-performance transformation engine, this could provide the basis for improved large-scale industrial transformations. In addition, we will investigate the use of the framework for end-user model comparison and repair. In a metamodelling environment such as DoME, this would allow end users to tailor model comparisons to their metamodels using models, rather than needing to "hard-code" the comparisons in the implementation environment.

ACKNOWLEDGMENTS

This research was funded by Australian Defence Science & Technology Group Research Agreement no. 6748 and the D2D CRC.

REFERENCES

[1] Vlad Acreţoaie, Harald Störrle, and Daniel Strüber. 2016. VMTL: a language for end-user model transformation. *SoSyM* (09 Jul 2016). https://doi.org/10.1007/s10270-016-0546-9

[2] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. 1999. Graph transformation for specification and programming. *Science of Computer Programming* 34, 1 (1999), 1–54. https://doi.org/10.1016/S0167-6423(98)00023-9

[3] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. 2011. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. Software Language Engineering 2010*. Springer, 296–305. https://doi.org/10.1007/978-3-642-19440-5_19

[4] Stefan Berger, Georg Grossmann, Markus Stumptner, and Michael Schrefl. 2010. Metamodel-Based Information Integration at Industrial Scale. In *Proc. MODELS 2010*, Vol. LNCS 6395. Springer, 153–167. https://doi.org/10.1007/978-3-642-16129-2_12

[5] Peter Braun and Frank Marschall. 2003. Transforming Object Oriented Models with BOTL. *Electronic Notes in Theoretical Computer Science* 72, 3 (2003), 103–117. https://doi.org/10.1016/S1571-0661(04)80615-7

[6] Brian Broll and Ákos Lédeczi. 2015. Extensible Visual Constraint Language. In *Proc. Workshop on Domain-Specific Modeling (2015) (DSM 2015)*. ACM, 63–70. https://doi.org/10.1145/2846696.2846704

[7] C. Ermel, M. Rudolf, and G. Taentzer. 1999. The AGG Approach: Language And Environment. In *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 551–603. https://doi.org/10.1142/9789812815149_0014

[8] Andrew Fish, Jean Flower, and John Howse. 2005. The semantics of augmented constraint diagrams. *Journal of Visual Languages & Computing* 16, 6 (2005), 541–573. https://doi.org/10.1016/j.jvlc.2005.03.001

[9] Holger Giese, Stephan Hildebrandt, and Leen Lambers. 2014. Bridging the gap between formal semantics and implementation of triple graph grammars. *SoSyM* 13, 1 (01 Feb 2014), 273–299. https://doi.org/10.1007/s10270-012-0247-y

[10] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. 2013. Automated verification of model transformations based on visual contracts. *Automated Software Engineering* 20, 1 (2013), 5–46. https://doi.org/10.1007/s10515-012-0102-y

[11] John Howse and Steve Schuman. 2005. Precise visual modeling: A case-study. *SoSyM* 4, 3 (2005), 310–325. https://doi.org/10.1007/s10270-004-0074-x

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72, 1-2 (2008), 31–39.

[13] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. 2018. Survey and classification of model transformation tools. *SoSyM* (2018), 1–37. https://doi.org/10.1007/s10270-018-0665-6 online first.

[14] Stuart Kent. 1997. Constraint Diagrams: Visualizing Invariants in Object-oriented Models. In *Proc. OOPSLA'97*. ACM, 327–341. https://doi.org/10.1145/263698.263756

[15] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. 2002. *Visual OCL: A Visual Notation of the Object Constraint Language*. Technical Report 2002/23. Technische Universität Berlin. http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/

[16] Levi Lúcio, Bentley James Oakes, Cláudio Gomes, Gehan M. K. Selim, Juergen Dingel, James R. Cordy, and Hans Vangheluwe. 2015. SyVOLT: Full Model Transformation Verification Using Contracts. In *Proc. MoDELS 2015 Demo and Poster Session*. 24–27. http://ceur-ws.org/Vol-1554/PD_MoDELS_2015_paper_8.pdf

[17] B. Meyer. 1992. Applying "design by contract". *Computer* 25 (1992), 40–51.

[18] OMG. 2014. *Object Constraint Language (v2.4)*. standard formal/2014-02-03. http://www.omg.org/spec/OCL/2.4

[19] OMG. 2016. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (v1.3)*. Object Management Group. http://www.omg.org/spec/QVT/1.0/PDF/formal/16-06-03.

[20] Arend Rensink. 2004. The GROOVE Simulator: AÂăTool for State Space Generation. In *Proc. 2nd International Workshop AGTIVE 2003*. Springer, 479–485. https://doi.org/10.1007/978-3-540-25959-6_40

[21] Andy Schürr. 1995. Specification of Graph Translators with Triple Graph Grammars. In *Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94) (LNCS)*, Vol. 903. Springer, 151–163. https://doi.org/10.1007/3-540-59071-4_45

[22] Matt Selway, Kerryn R. Owen, Richard M. Dexter, Georg Grossmann, Wolfgang Mayer, and Markus Stumptner. 2018. Automated Techniques for Generating Behavioural Models for Constructive Combat Simulations. In *Data and Decision Sciences in Action: Proc. Australian Society for Operations Research Conference 2016 (LNMIE)*. Springer, 103–115. https://doi.org/10.1007/978-3-319-55914-8_8

[23] Harald Störrle. 2011. Expressing model constraints visually with VMQL. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 195–202. https://doi.org/10.1109/VLHCC.2011.6070399

[24] Dániel Varró and András Balogh. 2007. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68, 3 (2007), 214–234. https://doi.org/10.1016/j.scico.2007.05.004

[25] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schoenboeck, and Wieland Schwinger. 2010. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. ICMT 2010*. Springer, 260–275. https://doi.org/10.1007/978-3-642-13688-7_18