International Journal on Software Tools for Technology Transfer (2021) 23:655–677
https://doi.org/10.1007/s10009-021-00630-2

FOUNDATION FOR MASTERING CHANGE

Regular

Towards language-to-language transformation

Dawid Kopetzki[1] · Michael Lybecait[1] · Stefan Naujokat[1] · Bernhard Steffen[1]

Accepted: 18 May 2021 / Published online: 18 June 2021
© The Author(s) 2021

Abstract

This paper proposes a simplicity-oriented approach and framework for language-to-language transformation of, in particular, graphical languages. Key to simplicity is the decomposition of the transformation specification into sub-rule systems that separately specify purpose-specific aspects. We illustrate this approach by employing a variation of Plotkin's Structural Operational Semantics (SOS) for pattern-based transformations of typed graphs in order to address the aspect 'computation' in a graph rewriting fashion. Key to our approach are two generalizations of Plotkin's structural rules: the use of graph patterns as the matching concept in the rules, and the introduction of node and edge types. Types do not only allow one to easily distinguish between different kinds of dependencies, like control, data, and priority, but may also be used to define a hierarchical layering structure. The resulting *Type-based Structural Operational Semantics* (TSOS) supports a well-structured and intuitive specification and realization of semantically involved language-to-language transformations adequate for the generation of purpose-specific views or input formats for certain tools, like, e.g., model checkers. A comparison with the general-purpose transformation frameworks ATL and Groove, illustrates along the educational setting of our graphical WebStory language that TSOS provides quite a flexible format for the definition of a family of purpose-specific transformation languages that are easy to use and come with clear guarantees.

Keywords Multi-level transformations · Model-to-model transformation · Graph rewriting · (Typed) structural operational semantics · Abstraction · Structural aggregation · Rule systems · Meta language · Model checking · Graph pattern

1 Introduction

Today, computational thinking [31,51] is not only important for computer scientists. Rather, it enters almost everybody's life, when, e.g., dealing with modern mobile phones, configuring your TV, or even simply buying a ticket at today's vending machines. Domain-specific languages are an ideal means to support this trend and to ultimately help transferring a significant part of the application development to the application expert. Characteristic for this development are two approaches:

– Language-Oriented Programming (LOP), which aims at moving program development closer to the application domain by providing developers with dedicated domain functionality [13,20,49], and
– Language-Driven Engineering (LDE) which directly involves the application experts in the application development by providing them with dedicated purpose-specific (graphical) languages that typically enhance the graphical notation used in the domain already, and that are meant to evolve during the application lifecycle to capture arising new needs [44].

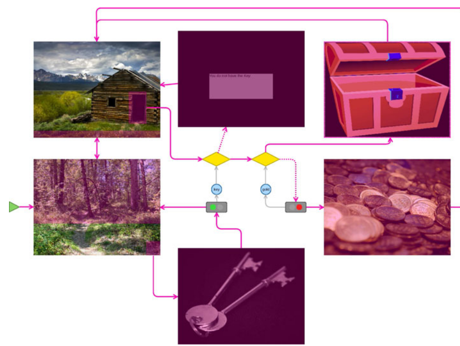
Precondition for these approaches are powerful frameworks, often called language workbenches [21], for the realization and evolution of the required domain-specific development environments. This often results in evolution steps that are not downwards compatible in the following sense: programs developed in the original version of the language are no longer contained in the evolved language. Thus,

✉ Stefan Naujokat
stefan.naujokat@tu-dortmund.de

Dawid Kopetzki
dawid.kopetzki@tu-dortmund.de

[1] Chair for Programming Systems, Department of Computer Science, TU Dortmund University, 44227 Dortmund, Germany

WebStory Language (WSL)



Transformation with TSOS rules

Model Checking

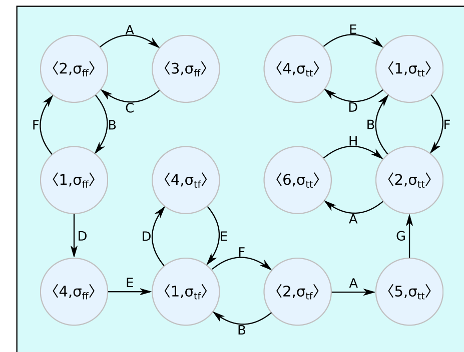


Fig. 1 Transforming WebStory Language models for model checking

program transformations are required to support the migration of existing programs to the new language.

In this paper, we propose an approach and framework for language-to-language transformation of, in particular, graphical languages[1] that is designed for transforming model representations into input formats for, e.g., optimization, verification and visualization purposes, or even for supporting the migration of applications after language evolution.

Simplicity [33], in particular compared to the common graph transformation-based approaches [1,3,17,18,39], was the guiding principle in the design of the underlying specification formalism which can be regarded as a generalization of Plotkin's structural operational semantics (SOS) [38] for pattern-based transformations of typed graphs:

- It clearly separates source and target graphs, and, similar to a typical compiler, leaves the source graph intact during the construction of the target graph.[2]
- It is additive in the sense that the target graphs are successively built up.
- It allows for a modular specification of abstraction using auxiliary transition relations [29].

Technically, this requires two generalizations of Plotkin's SOS rules: the use of graph patterns as the matching concept in the rules, and the introduction of node and edge types, both for the source and the target graphs. The point is that types do not only allow one to easily distinguish between different kinds of dependencies, like control, data, and priority—but may also be used to define a hierarchical layering structure where, e.g., micro and macro steps are distinguished.

The resulting *Type-based Structural Operational Semantics* (TSOS) supports a well-structured and intuitive specification and realization of complex model-to-model transformations adequate for the generation of purpose-specific views or input formats for certain tools, like, e.g., model checkers, or even for the migration of models after the underlying modeling language has evolved [6].

We consider TSOS, even though quite restrictive from a graph transformation perspective, as a format for the definition of a family of purpose-specific transformation languages that are easy to use, have clear guarantees, and which we envisage to semi-automatically derive from the source and target metamodels of an intended model-to-model transformation.

This paper, which extends [29], illustrates the purpose-specific transformation along the setting displayed in Fig. 1. On the left, it shows the WebStory language (WSL), which is a graphical domain-specific language we often use for education [30]. The model on the right represents the input language required by our model checker GEAR [2]. The green arrow marks the transformation between these two languages, whose specification in terms of TSOS-based rules will be discussed in detail in this paper.

In order to conveniently make the TSOS format applicable in practice, we use our language workbench CINCO [7,37] to generate domain-specific development environments for language-to-language transformations that are already tailored to the considered source and target languages. Of course, the ideas presented in this paper are applicable to any sophisticated language workbench supporting graphical languages. We just used CINCO as an example platform for our implementation, because we naturally were already very familiar with its meta-metamodel structures.

In the following, Sect. 2 first introduces the WebStory language—i.e., which kind of modeling elements are available and what they are used for—by means of a simple 'find

[1] Our main application context is CINCO [7,37], a language workbench for graphical languages. This is the reason for focusing our presentation of graphical languages. The ideas apply straightforwardly to textual languages also.

[2] This separation distinguishes us from typical graph transformation-based solutions which, similar to term rewriting systems, successively transform the source graph (inline) until the target graph is reached [1,3,17,18,39].

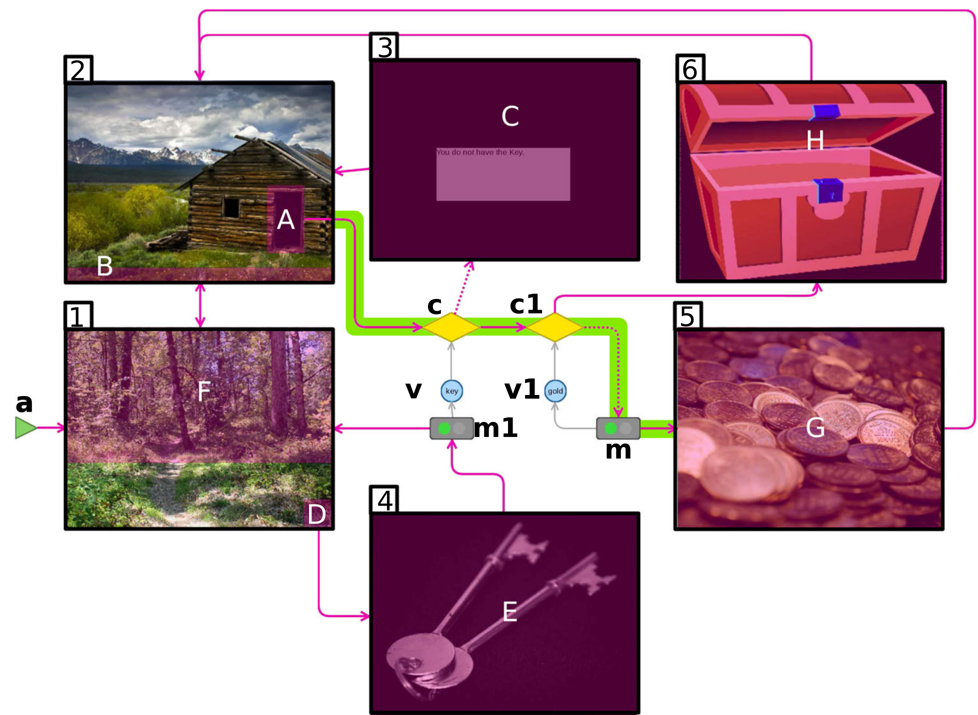


Fig. 2 An exemplary WebStory model reprinted from [29] (Images by:[11,25,42,50])

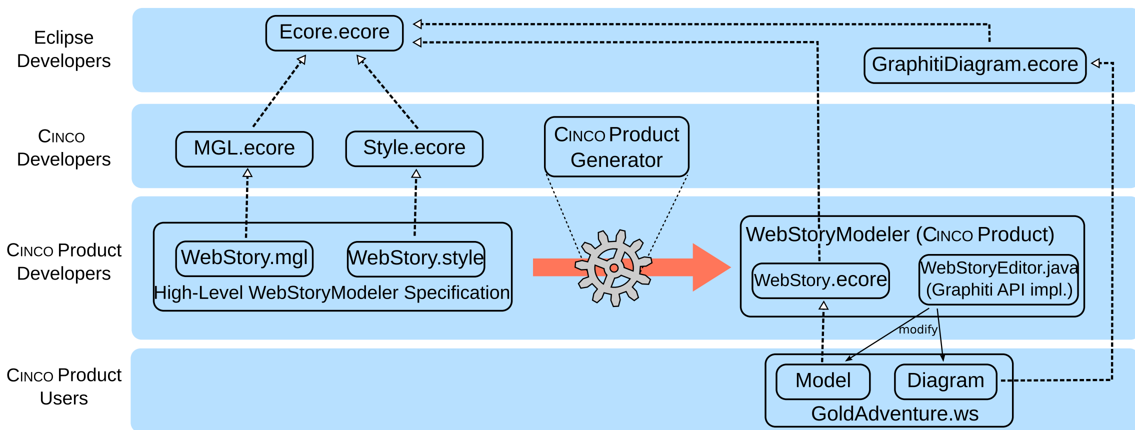


Fig. 3 Overview of the Cinco Meta Tooling Suite adapted from [37]

the treasure' example story, before Cinco is presented alongside the metamodel definition of the WebStory language. Afterwards, Sect. 3 first discusses the common practice of algebraic approaches to graph transformations, before we present our design decisions and formalisms for transformation systems based on TSOS. Subsequently, in Sect. 4, we apply those concepts to the WebStory language to enable it for model checking. To illustrate the simplicity of our DSL-based approach, Sect. 5 sketches the very same transformation using two well-known general-purpose approaches: (1) the Atlas Transformation Language (ATL), and (2) the algebraic, graph transformation framework *Groove*. We discuss the meta pattern for the presented transformation language in Sect. 6 and conclude the paper with a summary and indications for future work in Sect. 7.

2 Preliminaries

This section discusses the basics required for understanding the main contributions presented in Sects. 3 and 4. We first introduce in Sect. 2.1 the WebStory Language (WSL) for graphical modeling of small point&click adventure games which is used throughout the paper as running example. Subsequently, we present in Sect. 2.2 our language workbench Cinco, which is the technological basis for all graphical languages developed in the context of this paper.

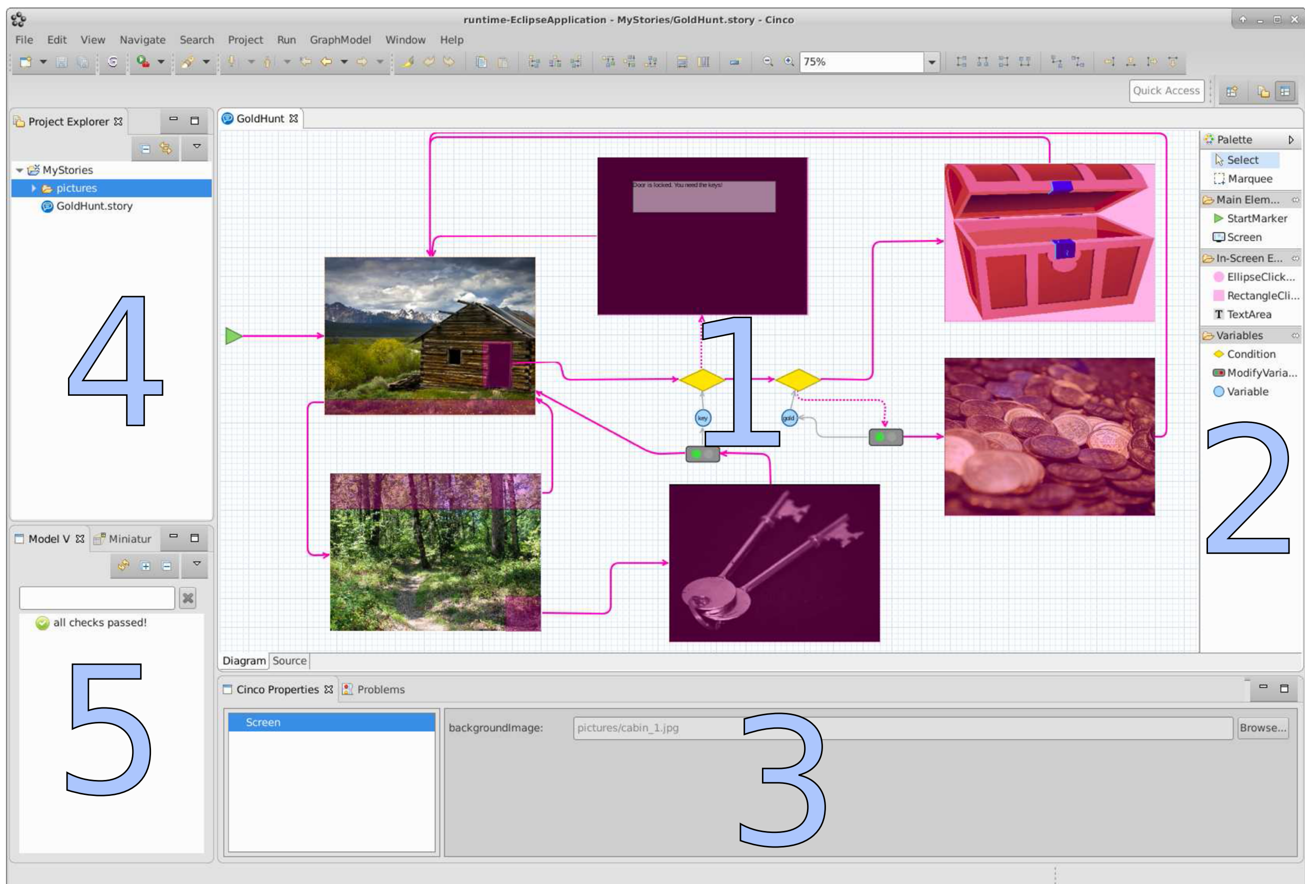


Fig. 4 Generated modeling tool (CINCO Product) from a CINCO language specification showing the canvas (1), the element palette (2), the property view (3), the project explorer (4), and the validation view (5)

2.1 The WebStory language

The *WebStory Language* (WSL) has been designed as a simple example for CINCO to be used for hands-on experience in teaching and workshops [30]. Created with simplicity in mind, WSL's aim is on the one hand to provide an intuitive 'game' modeling language that can even be used by non-programmers, and, on the other hand, to make it easy for the workshop/lecture participants (i.e., usually students) to learn metamodeling and DSL engineering concepts by expanding a simple language in terms of functionality.

A WebStory is modeled in a graphical editor and describes the flow from one screen to the other with conditional paths depending on values of Boolean variables. Modeled games can be generated to fully functional websites that can be executed in any web browser. The resulting story is played by clicking on certain areas within the screens with the aim to reach a predefined goal.

Figure 2 shows a model of a story where the player is challenged to get hold of a treasure hidden in a hut by finding the required key. Although being a simple story with little challenge, this model uses all types defined in WSL:

– Six *Screen* nodes (1-6), which correspond to the 'visitable places' in the story. The background image (shown in full-screen in the resulting website) is defined by the 'backgroundImage' attribute of the *Screen* type.
– Eight *Click Area* nodes (A-H), which define the interaction areas for the player. *Click Areas* are contained in *Screens* and define the next element in the story's control flow using pink *Transition* edges.
– Two *Variable* nodes (v and v1) named 'key' and 'gold', which represent a Boolean value. *Variables* are connected through gray *Data Flow* edges to *Condition* nodes and *ModifyVariable* nodes.
– Two *Condition* nodes (c, c1), which determine the successor in the story's control flow via the *True Transition* (represented as solid pink edge) or the *False Transition* (represented as dashed pink edge) based on the connected *Variable* node's value .
– Two *ModifyVariable* nodes (m and m1) which set a constant Boolean value of the connected *Variables* v1 and v, respectively. The constant value is represented by the 'value' attribute defined in the *ModifyVariable* type. The

successor of a *ModifyVariable* node is connected by a *Transition* edge.
– One *Start Marker* node (a) to define the initial *Screen* of the story.

The green background on the path $A \to c \to c1 \to m \to 5$ highlights the elements that are subject to our transformation examples in Sect. 4.

2.2 CINCO meta tooling suite

We use the CINCO *Meta Tooling Suite* [37] to generate the graphical development environments for all languages presented in this paper: the WebStory language, the state model used for model checking, as well as the dedicated DSL for modeling the transformations with TSOS-based rules.

CINCO is a simplicity-driven language workbench providing the generation of domain-specific graphical modeling tools. It is built upon the Eclipse Modeling Framework (EMF) [46] and the RCP [34] ecosystem and specializes on the (meta-level) domain of graph-based tools (consisting of nodes and edges). Those tools, which are fully generated from high-level specifications, support the inclusion of other Eclipse-based DSLs (like the ones developed with Xtext) in a service-oriented fashion [36]. Furthermore, Language-Driven Engineering (LDE) [44] can be facilitated with CINCO, and with the *Pyro* extension [53,54], CINCO-generated graphical modeling tools become web-based collaborative modeling environments.

Figure 3 shows an overview of the CINCO landscape for the development of the WebStory language presented in Sect. 2.1. The core of CINCO is the Meta Graph Language (MGL), which describes the abstract syntax of a graph-based Domain-Specific Visual Language (DSVL). The Meta Style Language (MSL) is used to define the concrete syntax of the DSVL. Thus, the high-level language specifications in CINCO (cf. Fig. 3, *WebStory.mgl* and *WebStory.style*) conform to the MGL and Style metamodels (cf. *MGL.ecore* and *Style.ecore* in Fig. 3). MGL and Style are in turn languages developed using the Eclipse Modeling Framework (EMF) [46] and Xtext [19] and thereby conform to the *Ecore.ecore* meta-metamodel.

MGL allows for the definition of *node* types, *container* types, and *edge* types. These types may comprise attributes. Furthermore, node and container types specify structural graph constraints: node types define *incoming* and *outgoing* edge constraints whereas container types additionally specify constraints for *containable* elements. Minimum and maximum cardinalities may be added for elements or groups of elements referenced in such constraints. When no cardinalities are present, arbitrary numbers of elements are allowed. Given a high-level specification consisting of instances of the MGL and Style languages, the CINCO Product Generator generates a CINCO Product consisting of a specific metamodel representing the abstract syntax of the DSL (*WebStory.ecore*) and a Graphiti-based [5] graphical modeling tool implemented in Java (simplified in the figure as *WebStoryEditor.java*).

Listing 1 shows an excerpt of the WebStory.mgl defining the available modeling elements for the language as introduced in Sect. 2.1. Given this high-level specification of the WebStory language, the CINCO Product Generator [28] generates the graphical modeling environment shown in Fig. 4. The resulting tool provides functionalities commonly known from (model-based) IDEs: a modeling canvas (1), a palette from which available elements can be drag&dropped to the canvas (2), a property view (3) and a project explorer (4). The generated tool is easily extensible, e.g., by custom validation rules which are shown in the 'Model Validation' view (5).

```
1  graphModel WebStory {
2  [...]
3    containableElements (Screen, Condition,
         ModifyVariable, StartMarker[1,1], Variable)
4
5    @style(startMarker)
6    node StartMarker {
7      outgoingEdges(InitialTransition[1,1])
8    }
9
10   @style(screen)
11   container Screen  {
12     @file("jpg", "JPG", "png")
13     attr EString as backgroundImage
14     containableElements (ClickableArea)
15     incomingEdges (InitialTransition[0,1],Transition,
           TrueTransition, FalseTransition)
16   }
17
18   abstract node ClickableArea {
19     outgoingEdges (Transition[1,1])
20   }
21
22   @style(condition)
23   node Condition {
24     incomingEdges (DataFlow[1,1], Transition,
          TrueTransition,FalseTransition)
25     outgoingEdges (TrueTransition[1,1],
          FalseTransition[1,1])
26   }
27
28   @style(modifyVariable)
29   node ModifyVariable {
30     incomingEdges(Transition, TrueTransition,
          FalseTransition)
31     outgoingEdges(Transition[1,1], DataFlow[1,1])
32     attr EBoolean as value := "false"
33   }
34
35   @style(variable, "${name}")
36   node Variable {
37     attr EString as name
38     outgoingEdges (DataFlow)
39     incomingEdges (DataFlow)
40   }
41 }
```

Listing 1 Excerpt of the WebStory language specification

3 Language-to-language transformations

In this section, we first give a brief introduction to the algebraic graph transformation approach and show how CINCO models can be represented as algebraic graph structures.

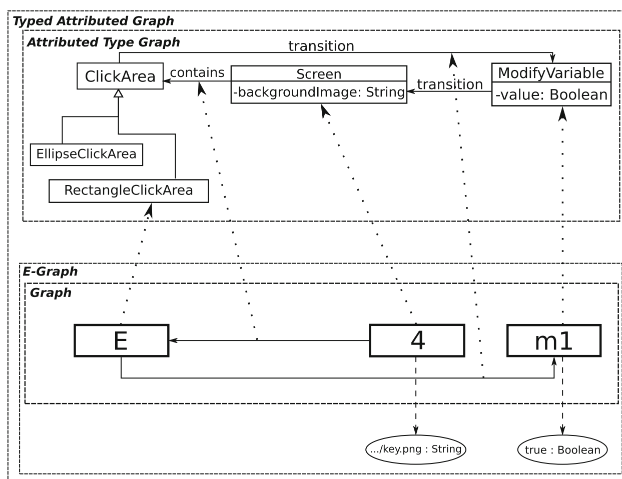


Fig. 5 Excerpt of the WebStory shown in Fig. 2 represented as typed attributed graph. Dotted arrows represent the *type* morphism, dashed arrows represent *node attribute edges*

Afterwards, we present our ideas towards a transformation language which aims at providing an intuitive way to specify transformations involving the aspect of computation.

3.1 Algebraic graph transformations

In the algebraic approach [15,16], graph transformations are applied to a *Typed Attributed Graph* (G, t). $G = (V, E)$ is a graph and $t : G \rightarrow ATG$ a graph morphism between G and an *Attributed Type Graph* $ATG = (TG, Z)$. Node and edge types are given by the distinguished *Type Graph* TG, attribute types are modeled by a *data signature* Z.[3] Attribute values are realized by an *E-Graph* which, roughly said, is an extension of the graph by *data nodes* (representing a value) and dedicated *attribute edges*, connecting graph elements with *data nodes*.

Models of DSLs developed with CINCO can generally be represented as typed attributed graphs. Figure 5 depicts an excerpt for the WebStory shown in Fig. 2: *Screen* '4', *Click Area* 'E', *Modify Variable* 'm1', and the transition from the *Click Area* to the *Modify Variable* node. Since a graph does not comprise the concept of container nodes and containment, we represent this relation in the type graph using a *contains* edge. The type morphism is given by dotted arrows and the attribute values are defined by dashed edges to the corresponding data nodes at the bottom of the figure. For instance, the dashed edge from node '4' to the data node '.../key.png : String' assigns the path to the background image for the *Screen* node '4'.

Ehrig et. al. provide in [16] a mapping between the metamodeling notions and the corresponding graph terminology, indicating the strong correspondence between model transformations and graph transformations.

The algebraic approach is used to formalize in-place graph transformations, i.e., transformations in which elements of the source language, target language, and possibly further elements (neither belonging to the source nor target language, e.g., auxiliary elements helping in bookkeeping during the transformation) are handled in one model. A transformation rule $p = (L, R)$, also called *production*, consists of a left-hand side graph L and a right-hand side graph R. It is applied by finding an occurrence of L in the input model (graph) G (a match of L in G) and replacing L by R in G, where elements of $L \setminus R$ are deleted in G. The graphs L and R are also called *patterns*.

In [24], the authors identified 60 model transformation tools, out of which 15 tools were purely graph-based, i.e., the entire transformation is specified using productions. Most transformation systems, however, use programming languages or provide imperative constructs to treat computational aspects. In Sect. 5.1, we present a transformation realized using ATL, a framework which supports the definition of imperative code.

In a purely graph-based approach, modeling the computation in the transformation rules requires the addition of a computational domain which complicates the definition of transformation rules. In the case of an in-place transformation, the intermediate transformation model and the rules consist of at least three parts (cf. Fig. 6). Elements of the source and target model are related by *trace links* to bookkeep the origins of created target elements. The computational part consists typically of a structure to reference the current element of the computation (the 'Pointer' node and outgoing dashed edge) a memory structure ('Memory' node) handling information of the computation and possibly further elements to indicate the current state of the computation ('Phase' node), e.g., for rule scheduling. Thus, the developer of a transformation has to model the initialization of a computation, the stepping through the model, the data representation and manipulation, and the termination control. Essentially this means that she has to develop a rule-based specification of an interpreter for the source model.

In Sect. 5.2, we present a transformation modeled using *Groove* [39], one of the most prominent purely graph-based graph transformation tools. We explicitly model the computation in the transformation rules, as sketched in the previous paragraph, and highlight the intricacies resulting from using a general-purpose graph transformation tool.

[3] For a more detailed description on the algebraic approach to graph transformations, data signatures and Σ-algebras, we recommend [15, 16].

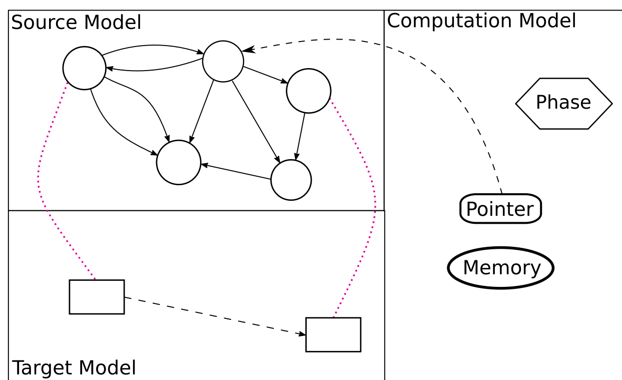


Fig. 6 Abstract representation of an intermediate in-place language-to-language transformation incorporating computational components

3.2 Towards language-to-language transformations inCINCO

In the context of CINCO, a language-to-language transformation is a transformation between DSVLs. As one of CINCO's fundamental motivations is simplicity [37], we want to introduce a transformation language which follows the CINCO paradigms as follows:

- modular, graph-based representation,
- domain focus,
- ease of specification.

This leads to the following design decisions, which are reminiscent of the structure of compilers and code generators, but unusual for graph rewriting-based model transformation systems where the considered graphs typically closely interlink source language, target language, and auxiliary aspects—structures which they often read, write, and delete in a single step in a so-called 'in-place' fashion (cf. Sects. 3.1 and 5.2).
Separation of Source Language and Target Language Source model graphs and target model graphs are clearly separated. This automatically imposes also a separation of the patterns used in rules which thereby clearly separate source language, target language, and auxiliary aspects like computation.
Additive Transformation System Source models are only read and target models only successively built up. No model elements will be deleted, which, e.g., eases cases where a multi-pass approach is adequate. The application of our transformations is "target-driven" in the sense that rules are applied as long as they lead to a change of the target model. This conforms to the fixpoint iteration in the *Rule Iteration* category described in [10].
Aspect-Specific Auxiliary Rules Specific aspects, in our example application the 'computation-based aggregation', are treated with a separate rule system which can be understood independently of the other rules. The rule system presented in Sect. 4 for treating computation resembles the structure of Plotkin's Structural Operation Semantics (SOS) [38]. This is why we call the rule format presented in this paper TSOS (typed structural operational semantics). It should be noted, however, that this computation-oriented rule system is only one possible realization of an auxiliary rule system. We are currently investigating other purposes, such as migration, and their corresponding aspect-specific auxiliary rule systems.
This clear structuring of the graph transformation approach follows our Archimedean point principle [45], which aims at localizing required changes and maximizing the parts which remain invariant, the Archimedean points.

3.3 A DSL for language-to-language transformation

Motivated by the preceding descriptions, we introduce a two-level transformation language. First-level rules define a global transformation modeling the relation between structures of the source and target language. This transformation might depend on internal information that has to be computed before the transformation can be executed. Therefore, a first-level rule may contain an optional part indicating the application of an auxiliary rule system.

Schematically, a rule is written as follows:

$$\frac{\boxed{\textit{Source Langauge}\ \text{Pattern}}\,,\ \boxed{\text{Auxiliary}}}{\boxed{\textit{Target Language}\ \text{Pattern}}}\ \text{Condition} \qquad (1)$$

Source and target language patterns are constructed using extended versions of the respective languages. The extension is typically small, like, e.g. the addition of a *wildcard* type to allow for matching of arbitrary types. The auxiliary rule system is realized by second-level rules:

$$\frac{\boxed{\textit{Source Language}\ \text{Pattern}}}{\langle \textit{Configuration} \rangle \dashrightarrow \langle \textit{Configuration} \rangle}\ \text{Condition} \qquad (2)$$

The structure resembles SOS rules [38] which consist of a premise, a conclusion and a condition. In our case, the premise specifies a graph pattern composed of elements of the source language, which has to be matched to execute a computation step. The definition of that step is again similar to SOS, i.e., it is defined by a transition ($\dashrightarrow$) between two *configurations*, where a configuration consists of the current position in an input (model) and a memory state σ (also called *store*) of the computation.

We say that this purpose-specific rule system specifies the typed structural operational semantics (TSOS) of a target graph model to emphasize its resemblance of Plotkin's SOS rule pattern. Moreover, as our two-level structure reminds of the semantics of so-called synchronous systems which

distinguish between macro steps (here the relation between source language and target language elements) and micro steps (here the computational steps), we call first-level rules *macro rules* and second-level rules *micro rules*.

In the following, Sect. 4 presents the transformation of graphical WebStory models into model structures that can directly be handed over to the GEAR model checker [2].

4 Enabling WebStories for model checking

To verify properties of a WebStory model like the one displayed in Fig. 2, the model has to be translated into a format suitable for a model checker.

Kripke Transitions Systems (KTSs), a generalization of both Kripke structures and labeled transition systems [35], had turned out to provide an adequate semantic model structure for graphical program models, as, e.g., supported by the jABC development environment [32]. Given a set of Atomic Propositions (AP), Kripke Transition Systems are quadruples

$$KTS = (S, Act, \rightarrow, I)$$

with:

- S, a set of states,
- Act, a set of actions,
- $\rightarrow \subseteq S \times Act \times S$, a transition relation (written as $s \xrightarrow{a} s'$), and
- $I : S \rightarrow 2^{AP}$, a *valuation* function.

A WebStory relates to a KTS in an intuitive way: Interacting with the story by clicking *Click Area*s on a *Screen* will cause a sequence of small state changes to happen until the next *Screen* is reached. Thus, each *Screen* combined with the 'initial *Screen*' information and a valuation of *Variable*s used in a WebStory defines a state in the KTS. The transition relation of a KTS is computed from control flow paths between *Click Area*s and *Screen*s in a WebStory.

Formally, a WebStory language model is a ten tuple

$$WS = (Sc, Ca, V, M, C, \dashrightarrow, \rightarrow, \xrightarrow{tt}, \xrightarrow{ff}, s_0)$$

with:

- Sc, a Set of *Screens*,
- Ca, a Set of *Click Areas*,
- V, a Set of *Variables*,
- M, a Set of *Modify Variable* elements,
- C, a Set of *Conditions*,
- $\dashrightarrow \subseteq (M \times V) \cup (V \times C)$, *DataFlow* relation,
- $\rightarrow \subseteq ((Ca \cup M) \times (Sc \cup M \cup C))$, a *Transition* relation,
- $\xrightarrow{tt} \subseteq (C \times (Sc \cup M \cup C))$, *TrueTransition* relation,
- $\xrightarrow{ff} \subseteq (C \times (Sc \cup M \cup C))$, *FalseTransition* relation,
- $s_0 \in Sc$, a screen marked with start marker, the initial screen of the WebStory.

Given a WebStory WS, we define for the set of atomic propositions AP $=_{df}$ $V \cup \{initial\}$ a KTS_{WS} with:

- $S \subseteq Sc \times \Sigma$, a set of states,
- $Act =_{df} Ca$,
- $\rightarrow \subseteq S \times Ca \times S$, and
- $I : S \rightarrow 2^{AP}$,

where Σ is the set of all possible *Variable* valuations in a WebStory:[4]

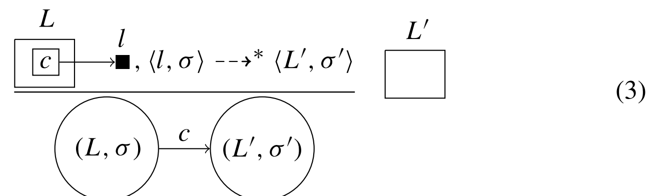
$$\Sigma =_{df} \{\sigma \mid \sigma : V \rightarrow \texttt{Bool}\}$$

In the remainder of this section, we model the transformation from WebStory to KTS using the DSVL for language-to-language transformations presented in Sect. 3.3. To illustrate the simplicity of our DSL-based approach, Sect. 5 subsequently sketches the very same transformation (as well as arising problems) using two well-known general-purpose approaches: (1) the Atlas Transformation Language (ATL) [23], and (2) the graph-transformation transformation framework *Groove* [39].

We will see that modeling the rules for the actual transformation—i.e., the rules mapping WebStory language elements to KTS elements—is manageable also in the generic approaches, but that the modeling of the execution semantics of an interpreter using graph re-writing techniques introduces an enormous overhead that can be drastically reduced by using our transformation language, in which computation is a purpose-specific aspect.

4.1 A DSL for computational transformations

In the following, we present the transformation rules for the ideas explained above. We show one rule in a schematic notation (cf. Rule 3) and as an instance of the CINCO generated transformation language (cf. Fig. 7).



(3)

[4] We use $[\![v]\!](\sigma)$ to evaluate the value of a variable $v \in V$ in the store σ.

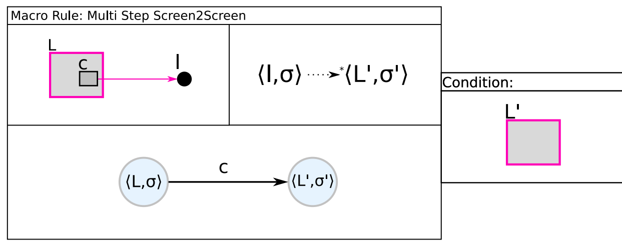


Fig. 7 Rule (3) represented in concrete syntax of our pattern-based transformation language

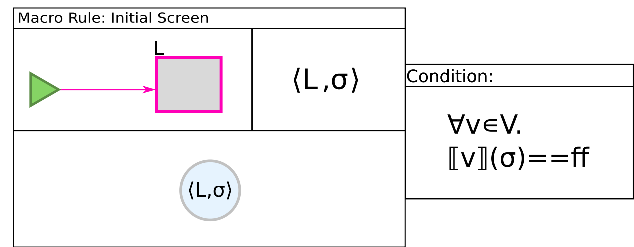


Fig. 8 Macro rule used to transform the initial state information

The (schematic) macro rule depicted in Rule (3) consist of four parts (cf. rule scheme (1)):

1. The upper left-hand side of the rule defines a pattern using elements of the WSL, where $L \in Sc$, $c \in Ca$ and l is a wildcard placeholder for an arbitrary node type of the WSL. The *Click Area* c and the placeholder l are connected through an edge typed as *Transition*.
2. The lower part describes the KTS-structure that should be created after application of this rule: Two states connected by a labeled transition. The state labels L, σ and L', σ' in the schematic representation as well as in the CINCO generated rule language representation are used as short hand notation meaning that a state is identified by the *Screen* L (L') and the *Variable* valuation in the store σ (σ').
3. The upper right-hand part of the rule represents an aggregated path of the WebStory: It starts with a store σ in the node l which is the node defined in the source pattern of the rule (i.e. the matched successor node of the *Click Area* node 'c'). It terminates in node L' with store σ' and may consist of an arbitrary number of computational steps (indicated by $\dashrightarrow^*$).
4. The condition of Rule (3) ensures that the aggregation process terminates when a *Screen* node is reached (requiring L' of configuration $\langle L', \sigma' \rangle$ to be a *Screen* node).

This rule models part of the transformation from a WebStory to a KTS: The information represented by *Variable*, *Condition*, and *Modify Variable* nodes is aggregated in the store. The global structure of a WebStory is described by the transition system which models the aggregated information explicitly in its states. Furthermore, the transformation has to specify how the valuation function I of the KTS states is derived from the store σ and σ'. In this case, the interpretation for state (L, σ) is defined by

$$I((L, \sigma)) = \{v \in V \mid [\![v]\!](\sigma) == \texttt{true}\}. \quad (4)$$

Transforming the information of the initial *Screen* to a KTS state is realized by the *macro* rule depicted in Fig. 8. The rule sets the valuation of the corresponding KTS state:

$$I((L, \sigma)) = \{initial\} \quad (5)$$

Please note that in the initial state of a WebStory model all variables are set to `false`.

The aggregation process of the transformation is defined by the *micro* rules depicted in Fig. 9. The lower part of a *micro* rule describes one computation step. For instance, the `Modify Variable` rule describes the step from node m to node l, updating the store by substituting the value of the Boolean variable v by the value of the *Modify Variable* node m.[5] Note that the identifiers used in the *configurations* refer to elements defined in the pattern in the upper part of the rule.

The `ConditionTT` and `ConditionFF` rules (cf. Fig. 9 bottom) determine the successor of a *Condition* node, depending on the value of the connected *Variable* node v.

Figure 10 shows the transformation process of the highlighted path in Fig. 2, with the variable configuration:

- $[\![key]\!](\sigma) = \texttt{true}$
- $[\![gold]\!](\sigma) = \texttt{false}$

The value of *key* and *gold* is represented by the index of $\sigma_{\texttt{t},\texttt{f}}$. Please note, that the figure does not show the rules, but their instantiation, i.e., the elements of the input model matched by pattern elements of the rules.

The MACRO rule 'Multi Step Screen2Screen' creates two KTS states corresponding to *Screen* '2' and *Screen* '5', respectively, and one transition connecting both states. The aggregation process starts in the configuration $\langle c, \sigma_{\texttt{t},\texttt{f}} \rangle$, i.e. the *Condition* node 'c' connected to the *Click Area* 'A' and the variable valuation described above. It terminates resulting in the configuration $\langle 5, \sigma_{\texttt{t},\texttt{t}} \rangle$, i.e. reaching *Screen* '5' and finding the treasure. The aggregation consists of three individual steps.

1. In the first step *Condition* node 'c', which evaluates the 'key' variable, is the current element. The variable's

[5] We use $[\![m]\!]$ to evaluate the value of the *Modify Variable* node in the WebStory model which is represented by the type's Boolean attribute *value*.

Fig. 9 Micro rules defining the aggregation in the WSL to KTS transformation

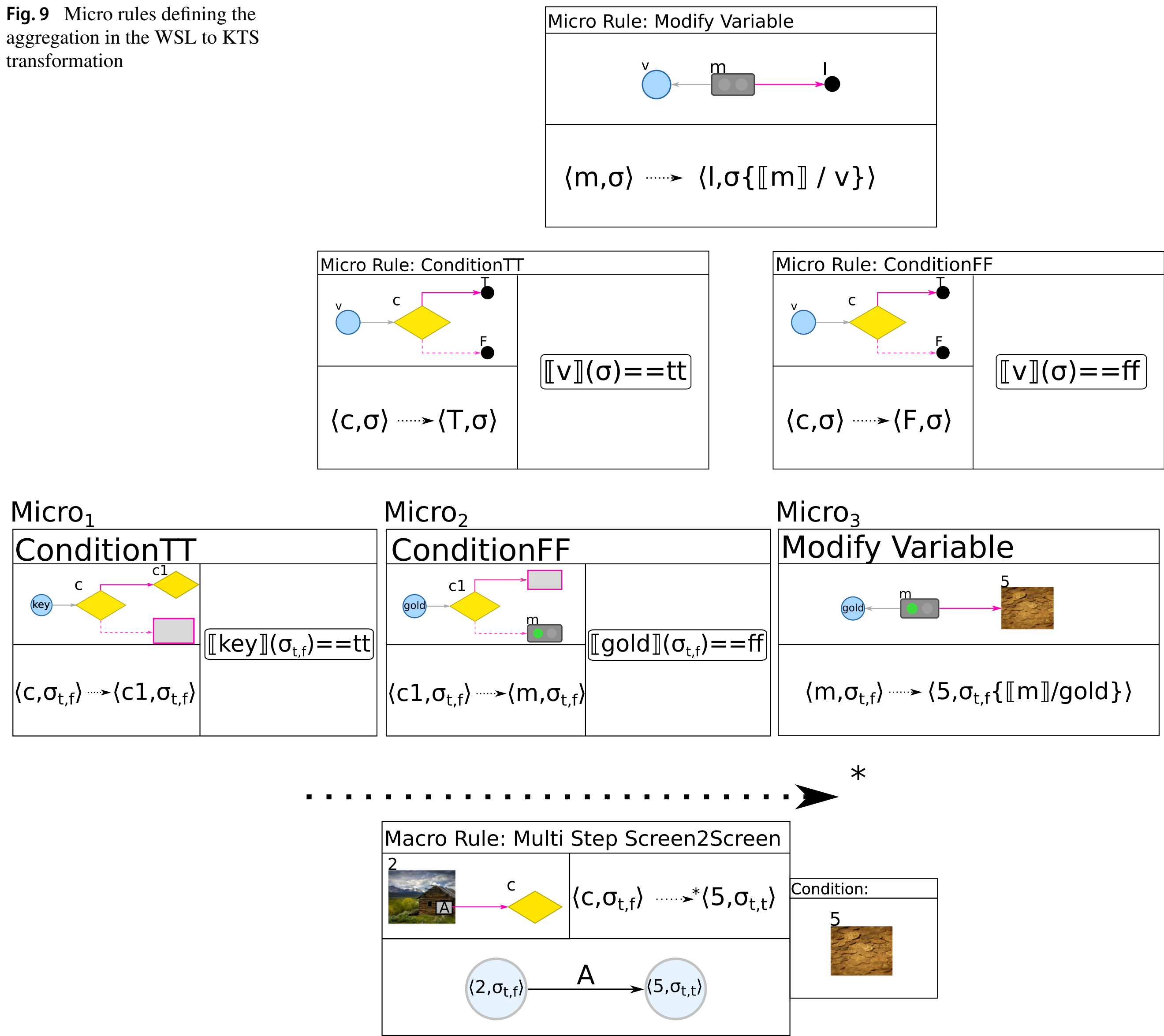


Fig. 10 An exemplary execution of the MACRO rule (3). The application of this rule on the *Screen* (2) and the *Click Area* (*A*) leads to the rule instantiations depicted below the dotted arrow. MICRO rules *ConditionTT*, *ConditionFF* and *Modify Variable* (above the dotted arrow) represent the aggregation steps resulting from the execution of the macro rule

value equals `true` and consequently, rule *ConditionTT* is applied. The rule determines *Condition* node ‘c1’ as the next element in the aggregation process without modifying the store (cf. $Micro_1$, $\langle c, \sigma_{\mathtt{t},\mathtt{f}}\rangle \dashrightarrow \langle c1, \sigma_{\mathtt{t},\mathtt{f}}\rangle$).

2. Variable ‘gold’ is evaluated by *Condition* ‘c1’ and since its value equals `false`, the *ConditionFF* rule is applied determining the *Modify Variable* node ‘m’ as next element (cf. $Micro_2$, $\langle c1, \sigma_{\mathtt{t},\mathtt{f}}\rangle \dashrightarrow \langle m, \sigma_{\mathtt{t},\mathtt{f}}\rangle$).
3. Now, the *Modify Variable* rule is applied, which continues the aggregation process to *Screen* ‘5’ and updates the store by substituting the value of the ‘gold’ *Variable* by the value of the *ModifyVariable* node ‘m’ (cf. $Micro_3$, $\langle m, \sigma_{\mathtt{t},\mathtt{f}}\rangle \dashrightarrow \langle 5, \sigma_{\mathtt{t},\mathtt{f}}\{[\![m]\!]/gold\}\rangle$).

Applying the entire transformation system to the WebStory shown in Fig. 2 results in the KTS shown in Fig. 11.

4.2 Application: model checking

Model checking allows for formal verification of finite state systems. Properties to verify are usually given in Linear Time Logic (LTL) or Computation Tree Logic (CTL) [9]. Given a model M of the system and a property φ, a model checker

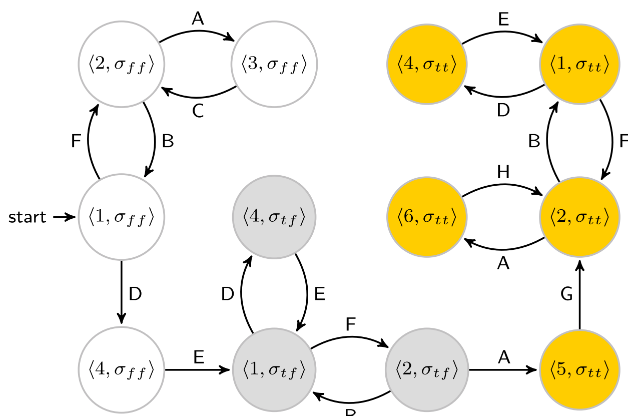


Fig. 11 KTS for the WebStory shown in Fig. 2. State labels indicate the valuation function I as described in (4). The colors highlight the valuation: σ_{ff} (white), σ_{tf} (gray) , and gold σ_{tt} (yellow). An incoming *start* edge for a state $s \in S$ indicates $initial \in I(s)$

checks if the model satisfies the property, written as $M \models \varphi$. As an example, one may want to verify for the initial state of the WebStory in Fig. 2 that:

1. there is always a way to get hold of the key and
2. the gold is only accessible with the key.

In CTL [8] these properties can be formulated as follows:

1. $AGEF(key)$
2. $A(\neg gold\ U\ key)$

In Fig. 11, gray states satisfy *key* (but not *gold*) and yellow states satisfy $key \wedge gold$. In this case, the resulting KTS satisfies the formulated properties.

The transformation of WebStories to KTS is an interesting combination of the partial evaluation [22] concerning the assignments and conditions with the property-oriented expansion (POE) [43] according to the store. There is no abstraction, and the POE prohibits information loss due to the introduction of non-determinism. Thus, the subsequent model checking is correct and complete for WebStories.

5 Computational transformation using a general purpose transformation language

The discussion in this section aims at illustrating the impact of special-purpose transformation languages. On the one hand, using a DSL instead of a powerful general-purpose tool, not all kinds of transformations can be modeled conveniently. On the other hand, DSLs supporting specific aspects of a transformation require the transformation developer to only understand a set of specialized language constructs, and, even more importantly, free the transformation developer from technical concerns like dealing with rule scheduling, bookkeeping, and termination conditions.

In Sect. 4, we used our SOS-based special-purpose language for realizing the transformation from WebStorys to KTS. In order to illustrate the impact of our DSL-based approach in comparison to general-purpose approaches, Sect. 5.1 shows specification excerpts of the same transformation in ATL [23], while Sect. 5.2 discusses how to realize the transformation within the graph transformation framework *Groove* [39].

5.1 WebStory to KTS using ATL

The most prominent Eclipse-based model transformation languages, like the ATLAS Transformation Language (ATL) [23], the Epsilon Transformation Language (ETL) [26], Viatra [48], and Xtend, support a text-based specification of transformations. Xtend, which is described as ‘Java with Spice’ [52], is a general-purpose programming language featuring special methods supporting tracing in model transformation.

ATL, ETL, and Viatra are hybrid languages comprising declarative and imperative language aspects. The declarative parts are used to traverse and define constraints for elements of the source language and their mapping to elements of the target language. The imperative parts are typically used to define transformations where ‘significant processing and complex mappings are involved’ [26]. In our example, the mapping between *Screen*s, computed variable valuations, and KTS states is such a complex mapping requiring significant processing to compute the variable valuations.

Our aim is to highlight the easy and elegant way to define these kinds of transformations. Viatra and ETL do not focus on this aspect. Viatra focuses on scalable reactive model transformations and provides a text-based language to define source language patterns. The actual transformation is then defined in code referring those patterns. ETL is part of Epsilon, a family consisting of ten languages used to solve problems in MDE. To define a transformation using Epsilon, one has to learn these languages. Since those aspects contradict our simplicity-oriented approach, we focus on ATL which provides an easy and direct access to model transformations and, similar to our approach, does not burden the transformation developer with technical details of the execution process.

In ATL, transformations are organized in modules. A module consists of a set of transformation rules and helper methods. Rules are preferably expressed in a declarative way and define mappings between elements of the source language and target language in form of patterns. Furthermore, they define how to traverse elements to retrieve information required to create target model elements.

ATL executes a transformation by applying *matched* rules whose source language pattern match structures in the source model, creating the corresponding target model structure. Similar to our approach, source model and target model are separated, and the transformation modifies only the target model. On rule application, a *trace model* is constructed managing bookkeeping information between the matched elements in the source model and created elements of the target model. In addition to the automatic rule execution, rules can be invoked from different parts of a transformation rule giving the developer more control over the rule execution.

ATL provides three types of rules to define transformations which are structured into different sections:

- A *from* section facilitating the definition of source language patterns,
- a *to* section facilitating the definition of target language patterns,
- a *using* section to define local variables, and
- a *do* section used to define imperative code.

The rule types are structurally similar, but differ in the way they are invoked:

- *Matched rules* provide a *from* section, a *to* section, and an optional *using* and *do* section. Matched rules are automatically executed, if the source model contains structures conforming to the pattern defined in the *from* section and create an instance of the structure defined in the *to* section.
- *Lazy rules* are syntactically similar to *matched rules*, but have to by explicitly called by other rules. Whenever a lazy rule is executed it creates an instance of the structure defined in the *to* section. Thus calling a lazy rule for the same match in the source model several times results in multiple occurrences of similar instances in the target model. This can be avoided using *unique lazy rules* (adding a `unique` flag in a *lazy rule*'s definition), which query the constructed trace model before creating a new element.
- *Called rules* consist of a *to* section, a *using* section, and a *do*-section. All sections are optional. A called rule does not provide a *from* section but is parameterized to define the input for the transformation rule. In contrast to *lazy* and *matched* rules, they can be called from imperative code sections.

In addition to rules, ATL provides *helpers* and *attribute helpers* to define factorized ATL code, which can be called from different points of a transformation. Helpers and attribute helpers correspond to methods and global variables of programming languages. We will use them in our transformation to realize the computation process and manage crucial bookkeeping information.

ATL's strength lies in the definition of transformations between models with a similar structure, where the mapping between source language elements and target language elements is straightforward. The preferred way to define a transformation in ATL is to use mainly matched rules and lazy rules with a unique flag to define the mappings between source language and target language elements, and then let ATL execute the transformation by applying the matched rules for each match of their source pattern in the source model.

In our example, the mapping between *Screen*s and KTS states relies on the variable valuation (σ), which, as we will see, poses the biggest challenge when defining a transformation in ATL.

In the following, we sketch a possible transformation which creates a KTS transition with its corresponding source and target states for each possible variable valuation. Given an initial variable valuation σ our algorithm executes the following four steps for each *Click Area* (***ca***) in the WebStory model:

1. Create KTS transition ***trans*** from ***ca***
2. Create KTS state ***source*** from (***parent***, σ) where ***parent*** is ***ca***'s parent *Screen* and set it as the ***trans***' source state.
3. Create KTS state ***target*** from (***next***, σ'), with
   - ***next*** as the reachable *Screen* from ***ca*** given σ, and
   - σ' as the variable valuation resulting from the reachable *Screen* computation.

   Set ***target*** as ***trans***' target state.
4. Repeat the transformation for all discovered variable valuations.

The integration of computed information concerns the operations 'Create KTS state ***source*** from (***parent***, σ)' and 'Create KTS state ***target*** from (***next***, σ')' (Steps 2 and 3, respectively) and the recursive call for discovered variable valuations (Step 4). More precisely, in these steps the following problems occur:

- Step 2: Since a *Screen* can contain multiple *Click Area*s, this step is executed several times with the same combination of *Screen* and σ requiring a bookkeeping mechanism, to either create or retrieve the corresponding KTS state.
- Step 3: A *Screen* can be reachable via two different *Click Area*s with the same variable valuation, requiring a bookkeeping mechanism, to create or retrieve the corresponding KTS state.
- Step 4: Variable valuations, which can be discovered during the computation of the reachable *Screen* in Step 3,

have to be provided as input for the transformation to capture all possible states of a WebStory model.

Realizing Step 4 using a matched rule would require to provide a pattern for variable valuations in a rule's *from* section and presume that all possible variable valuations are represented in the source model facilitating ATL to match them and thereby execute the transformation.

Implementing Steps 2 and 3 using matched rules or unique lazy rules demand for a correct bookkeeping between a KTS state and the tuple $t \in Sc \times \Sigma$ from which that state is created. Since ATL's *trace model* only manages relations between elements contained in the source model and target model, this also would require to represent the variable valuation explicitly in the source model.

Consequently, we cannot provide a correct implementation of these steps using matched rules or unique lazy rules, since the variable valuation is not persisted in the source model but a transient information.

In the following, we focus on the problems described for Steps 2 and 3, and exemplify the effect of using

- ATL's bookkeeping mechanism,
- no bookkeeping mechanism, and
- a bookkeeping mechanism integrating the variable valuation

to manage the relation between *Screen*s and KTS states for the transformation sketched above. All transformations are executed for the WebStory model represented in Fig. 12 with the initial variable valuation $v = \texttt{false}$. For this examples, we assume that the *Click Area*s are iterated in an alphabetical order.

5.1.1 ATL's bookkeeping

The transformation implemented using ATL's bookkeeping mechanism, i.e. realized utilizing unique lazy rules, creates the KTS structure

$$(1, \sigma_f) \xrightarrow{A} (2, \sigma_f)$$

for *Click Area* 'A', its parent *Screen* '1', and reachable *Screen* '2', along with the bookkeeping information that relates

- *Screen* '1' with $(1, \sigma_f)$ and
- *Screen* '2' with $(2, \sigma_f)$.[6]

For *Click Area* 'B', only the new transition '$\xrightarrow{B}$' is created. The transition's source state and target state are retrieved

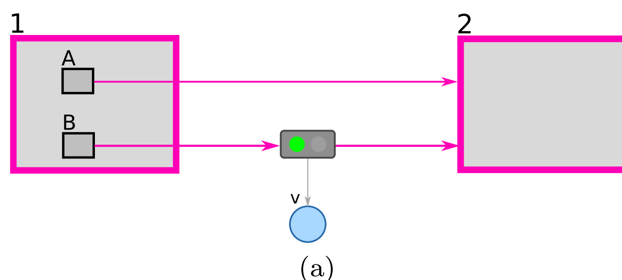


Fig. 12 WebStory model used to exemplify the usage of different bookkeeping mechanism (cf. Sects. 5.1.1, 5.1.2, 5.1.3) in the WebStory to KTS transformation

from the trace model, since the KTS states for the *Click Area*'s parent *Screen* '1' and reachable *Screen* '2' were created in the previous iteration. The subsequent iterations for the discovered variable valuation ($v = \texttt{true}$) and *Click Area*s 'A' and 'B' create duplicate KTS transitions, and, since ATL does not consider the variable valuation in its trace model, retrieves the existing KTS states, which were created for *Screens* '1' and '2'. This transformation results in a KTS representing the reachability between *Screen*s in a WebStory neglecting the computed information as illustrated in Fig. 13(a).

5.1.2 No bookkeeping

To realize the transformation in ATL without using its bookkeeping mechanism, we have to use called rules and lazy rules. Please recall that those types of rules create an instance of the structure defined in their *to* section each time they are executed. Without considering the trace model, the execution of Steps 2 and 3 result in the creation of a disconnected KTS containing duplicate states, since in each iteration and for each variable valuation a new instance of the structure consisting of two states, connected by a transition, is created. For the WebStory shown in Fig. 12 a resulting KTS is represented in Fig. 13b.

5.1.3 Manual bookkeeping with variable valuation

A correct transformation requires elaborate bookkeeping in order to avoid the problems of the previous realizations. Implementing a bookkeeping mechanism that integrates σ results in the bookkeeping information, relating

- (*Screen* '1', σ_f) with $(1, \sigma_f)$,
- (*Screen* '2', σ_f) with $(2, \sigma_f)$,
- (*Screen* '1' , σ_t) with $(1, \sigma_t)$, and
- (*Screen* '2' , σ_t) with $(2, \sigma_t)$.

In contrast to the execution using ATL's bookkeeping mechanism, the states $(1, \sigma_t)$ and $(2, \sigma_t)$ are created in the iteration for the variable valuation $v = \texttt{true}$. Furthermore, the KTS

[6] Please recall that we use σ in the representation of a KTS state as short hand notation for the *valuation* function of this state. (cf. Sect. 4)

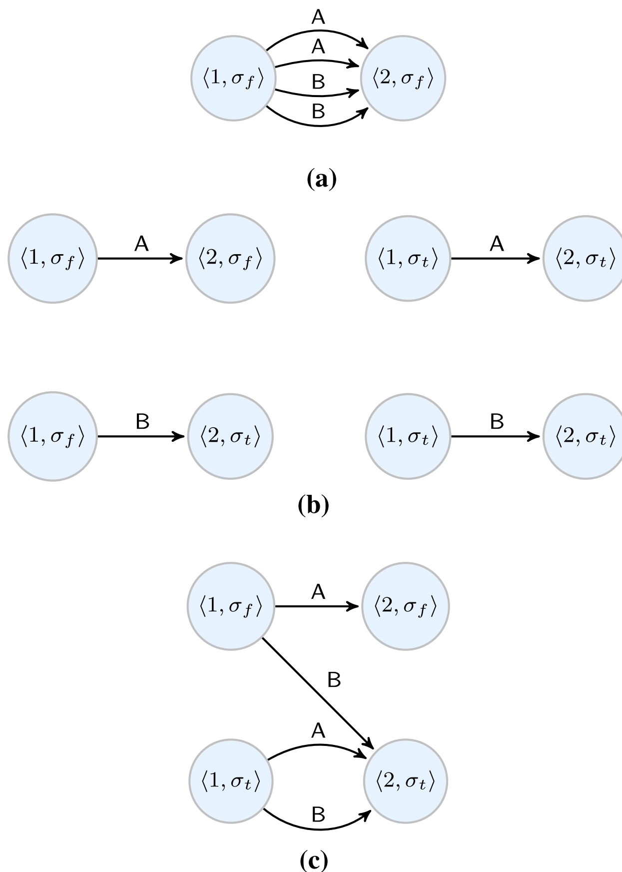


Fig. 13 Transformation results for the WebStory model represented in Fig. 12 omitting the variable valuations in the bookkeeping between *Screens* and KTS states (**a**), without any kind of bookkeeping (**b**), and considering the variable valuation in the bookkeeping (**c**)

is connected, since the bookkeeping information can be used to find a transition's source and target states, if they where created in preceding transformation steps, resulting in the KTS represented in Fig. 13c.

To implement a correct transformation in ATL, i.e. integrating σ into the bookkeeping mechanism, we augmented the approach using only called rules and lazy rules (i.e., the no bookkeeping mechanism approach) by our own bookkeeping implementation. Therefore, we add the following helper attributes:

- A map called `transformedScreens` mapping tuples of *Screens* and variable valuations to KTS states, managing bookkeeping information to avoid the creation of duplicate KTS states, thereby helping to correctly implement Steps 2 and 3.
- Two sets called `oldSigmas`, `newSigmas` monitoring the progress of the transformation, helping to implement Step 4 correctly.

Their realization is shown in Listing 2. In Line 2, the set `newSigmas` is initialized by the invocation of the `initSigmas` helper method, which creates a variable for each *Variable* node of the input WebStory and sets it to `false`.[7] These helpers are used in the following transformation rules:

```
1 helper def : oldSigmas :
      Set(Sequence(TupleType(name: String, value:
      Boolean))) = Set{};
2 helper def : newSigmas :
      Set(Sequence(TupleType(name: String, value:
      Boolean))) = Set{thisModule.initSigma};
3
4 helper def : transformedScreens :
5   Map (TupleType( screen: WS!Screen, sigma :
        Sequence(TupleType(name: String, value:
        Boolean))), KTS!State ) = Map{};
```

Listing 2 Helpers used to manage the correct execution of the transformation.

- A called rule `Transform`, which manages the execution of the transformation using the sets `oldSigmas` and `newSigmas`. This rule is the implementation of Step 4 in the sketched transformation.
- A lazy rule `Ca2Transition`, which creates the target model structures, i.e., a KTS transition for a given *Click Area* along with the associated source and target states. Therefore, it is the implementation of Steps 2 and 3 of the sketched transformation.
- A lazy rule `Screen2State`, which creates a KTS state for a given *Screen* and σ.

The `Transform` rule is shown in Listing 3. It is flagged by `entrypoint`, which leads to its invocation at the beginning of ATL's transformation process. It consists of a *do* section with two nested loops, iterating over all *Click Areas* of the input WebStory (Line 6) and all discovered variable valuations (Line 7), calling the `Ca2Transition` rule in each iteration.

```
1  entrypoint rule Transform() {
2  do {
3    if (thisModule.oldSigmas.size() <
         thisModule.newSigmas.size()) {
4      thisModule.discoveredSigmas <-
           thisModule.newSigmas - thisModule.oldSigmas;
5      thisModule.oldSigmas <- thisModule.newSigmas;
6      for (ca in WS!ClickArea->allInstances()) {
7        for (sig in thisModule.discoveredSigmas) {
8          thisModule.Ca2Transition(ca, sig);
9        }
10     }
11     thisModule.Transform();
12   }
13 }
14 }
```

Listing 3 WebStory to KTS transformation entry rule scheduling the transformation.

[7] Please note that `thisModule` has to be prepended to call helpers or rules defined in the current ATL module.

```
1  lazy rule Ca2Transition {
2    from ca: WS!ClickArea, sig:
         Sequence(TupleType(name: String, value:
         Boolean))
3    using {
4      reachableScreen :
5        TupleType(screen: WS!Screen,
6          sigma: Sequence(TupleType(name: String,
               value: Boolean)))
7      = thisModule.reachableScreen(ca, sig);
8      sourceState : KTS!State =
           thisModule.transformedScreens.get(
           Tuple{screen=ca.screen, sigma=sig});
9      targetState : KTS!State =
           thisModule.transformedScreens.get(
           Tuple{screen=reachableScreen.screen,
           sigma=reachableScreen.sigma});
10
11   }
12   to trans : KTS!Transition(
13     action <- ca.label,
14     source <-
15     if (sourceState <> OclUndefined )
16       then sourceState
17       else thisModule.Screen2State(ca.screen, sig)
18     endif,
19
20     target <-
21     if ( targetState <> OclUndefined )
22       then targetState
23       else thisModule.Screen2State(
24       reachableScreen.screen,
25       reachableScreen.sigma)
26     endif
27   )
28   do {
29     thisModule.newSigmas <- thisModule.newSigmas
30       .including(reachableScreen.sigma);
31     thisModule.transformedScreens <-
           thisModule.transformedScreens
32       .including(Tuple{screen=ca.screen, sigma=sig},
             trans.source);
33     thisModule.transformedScreens <-
           thisModule.transformedScreens
34       .including(Tuple{screen=reachableScreen.screen,
             sigma=reachableScreen.sigma},
             trans.target);
35   }
36 }
```

Listing 4 ATL rule creating a KTS transition from a *Click Area* and a variable valuation.

```
1 lazy rule Screen2State{
2   from sc: WS!Screen, sig: Sequence(TupleType(name:
        String, value: Boolean))
3   to state : KTS!State(
4     label <- sc.label,
5     ap <- thisModule.variablesToString(sig)
6   )
7 }
```

Listing 5 Rule creating a KTS state given a *Screen* and σ.

Listing 4 shows the realization of `Ca2Transition`. The KTS transition for the given *Click Area*, its source state and target state are created in Lines 12-26. To prevent the creation of duplicate KTS states, Lines 15 and 21 check if a corresponding state already exists using the local variables `sourceState` and `targetState` defined in the *using* section of the rule (cf. Lines 8 and 9). If they do not exist, they are created by calling the `Screen2State` rule (cf. Listing 5). The code defined in the *do* section of rule `Ca2Transition` updates the set `newSigmas` and the `transformedScreens` map.

```
1  helper def : reachableScreen(
2    current: WS!Activity,
3    sig: Sequence(
4      TupleType(name: String, value: Boolean)))
5  : WS!Screen =
6    if (current.oclIsTypeOf(WS!Screen))
7    then current
8    else
9      if (current.oclIsTypeOf(WS!ClickArea))
10     then thisModule.reachableScreen(
11       thisModule.traversClickArea(current), sig)
12     else if (current.oclIsTypeOf(WS!Condition))
13       then thisModule.reachableScreen(
14         thisModule.traverseCondition(current, sig),
             sig)
15       else if
               (current.oclIsTypeOf(WS!ModifyVariable))
16         then thisModule.reachableScreen(
17           thisModule.traverseModifyVariable(current),
18           thisModule.updateSigma(current, sig))
19         else OclUndefined
20         endif
21       endif
22     endif
23   endif
24 ;
25
26 helper def : traversClickArea(ca: WS!ClickArea) :
       WS!Activity =
27   ca.transition
28 ;
29
30 helper def : traverseCondition(
31   cond: WS!Condition,
32   sig: Sequence(
33     TupleType(name: String, value: Boolean))) :
34 WS!Activity =
35   if (sig->select(tup |
36     tup.name = cond.dataFlow.name).first().value)
37   then
38     cond.trueTransition
39   else
40     cond.falseTransition
41   endif
42 ;
43
44 helper def : traverseModifyVariable(
45   mv: WS!ModifyVariable) :
46 WS!Activity =
47   mv.transition
48 ;
```

Listing 6 Rules used to compute a reachable *Screen* given a variable valuation σ.

To create a transition's target KTS state, we have to compute the *Screen* which is reachable from the corresponding *Click Area* given σ. This is realized using the helpers shown in Listing 6. Given a node of the input model and a variable valuation, the `reachableScreen` helper retrieves the successor of the node by calling

- `traverseClickArea`,
- `traverseCondition`, and
- `traverseModifyVariable`

until a *Screen* is reached. Traversing a *Modify Variable* can change the variable valuation (cf. `updateSigma` in Line 18). Consequently, `reachableScreen` returns a tuple consisting of the reached *Screen* and the resulting variable valuation.
Summarizing, ATL's rule format, which can be regarded as a de-facto standard for transformations in the model-driven community, is not adequate to express transformations that comprise computational aspects like the one discussed here

for the WebStory. Rather, correctness can only be guaranteed via involved rule scheduling that has to be enforced using auxiliary information (via helpers) about variable valuations and other forms of bookkeeping, e.g., for managing the required target structures and guaranteeing termination.

In our setting, enforcing the correct rule scheduling is particularly challenging, as can be seen when looking at the small example transformation above: it requires auxiliary computations, e.g. reading bookkeeping information (cf. Listing 4, Lines 8-9, 15, and 21) and updating auxiliary structures (cf. Lines 28-35), which clutters the transformation rules. In fact, the resulting ATL transformation essentially resembles an imperative program as only a few parts can be defined in ATL's preferred declarative way.

5.2 WebStory to KTS using groove

Rules specified in *Groove* use a compact syntax comprising colour coding:

- Solid black lines define *Reader*. They have to occur in the input graph. Applying a rule does not manipulate these elements.
- Dashed blue lines define *Erasers*. They have to occur in the input graph to apply the rule and are deleted after rule application.
- Solid green lines define *Creators*. They are created after rule application.
- Dashed red lines define *Embargoes*. They are forbidden in the input graph. Thus, if an embargo occurs in the input graph, the rule is not applicable.

To model the transformation in *Groove*, we define an *attributed type graph* comprising WebStory types and KTS types. Furthermore, modeling the computational part with *Groove* requires to manage every aspect that is implicitly given when using our domain-specific approach, explicitly in the transformation rules. Therefore, we introduce additional types managing the computation:

- A **Phase** type which distinguishes between the stages of the transformation, e.g., target model creation or computation.
- A **Memory** type, used to temporarily save the name of a *Click Area*, until the corresponding *LabeledTransition* is created.
- A **Graph Pointer** type (GP)
- A **Current** edge type to connect GP typed node with the currently considered node in the input model.
- A **Bool** type containing a Boolean attribute **value**. For each *Variable* node in the input WebStory, a **Bool** node is created to store the *Variable* node's value.

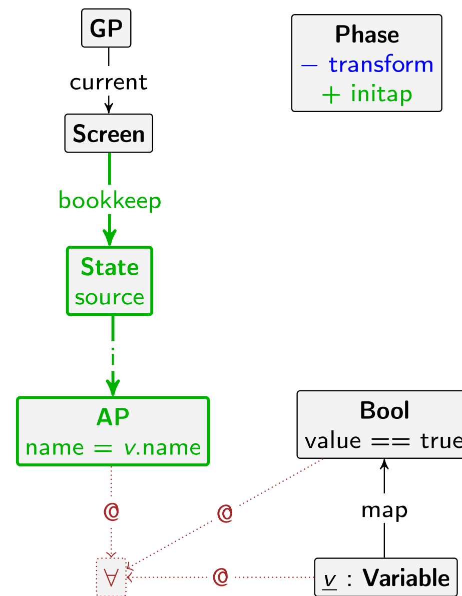


Fig. 14 Creation of the KTS state corresponding to a *Screen* in the WebStory

- Attributes **source** and **target** for the KTS *state* type, helping to connect created *state* nodes correctly.

We model the *transition* type of KTS as a node type *LabeledTransition* containing a string attribute *label*, since *Groove* does not support attributed edge types.

Given an input graph conforming to the attributed type graph, our transformation starts with an initialization, in which

1. we set the Boolean values for *Variable* nodes,
2. create a **Phase** node,
3. identify the *Start Screen* of the WebStory model, and create a **GP** node, referencing that *Screen* using a **Current** edge.

After the initialization, the actual transformation starts. We create two KTS states connected through a labeled transition by executing the following steps.

1. Create the source KTS state for the *Screen* node referenced by the **GP** (cf. Fig. 14).[8] For each *Variable* node whose Boolean values equals `true` an atomic proposition node (**AP**) named after the *Variable* node's name, is created.
2. Choose a *Click Area* node contained in the current *Screen* node.

[8] The '∀' node allows for matching of sub-graphs. Refer to [39,40] for a detailed explanation.

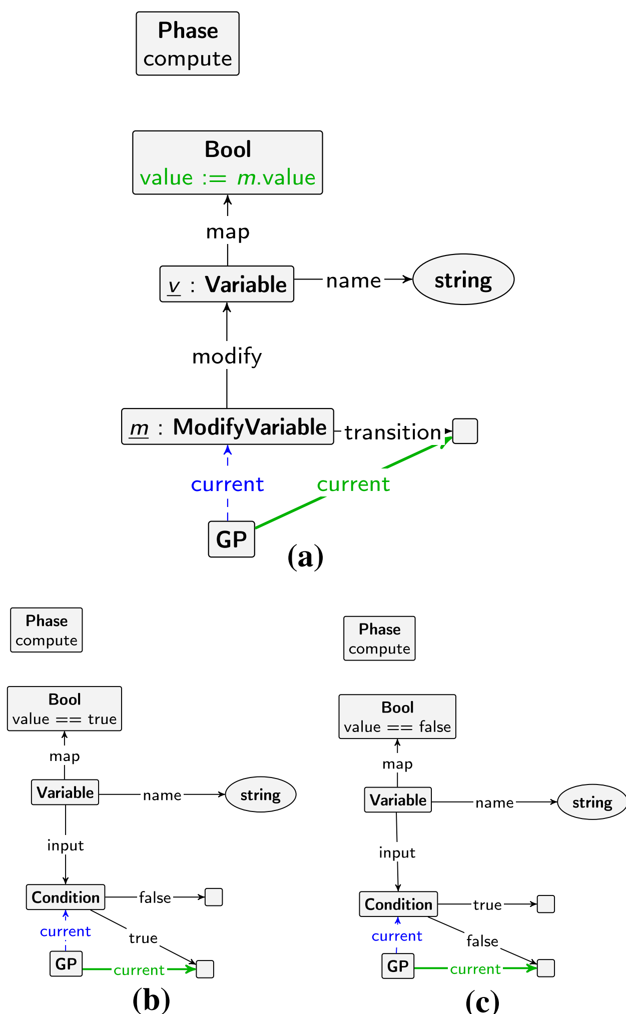


Fig. 15 Computational rules from the transformation shown in Fig. 9 modeled in Groove: `Modify Variable` (**a**), `ConditionTT` (**b**), and `ConditionFF` (**c**). In the rules green elements will be created after rule application, blue elements are matched and deleted after rule application

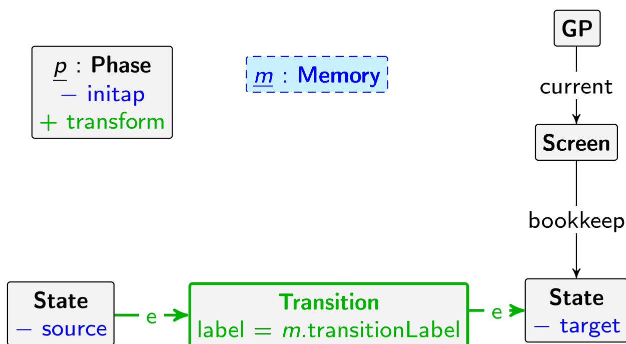


Fig. 16 Connecting the created KTS states after one computation

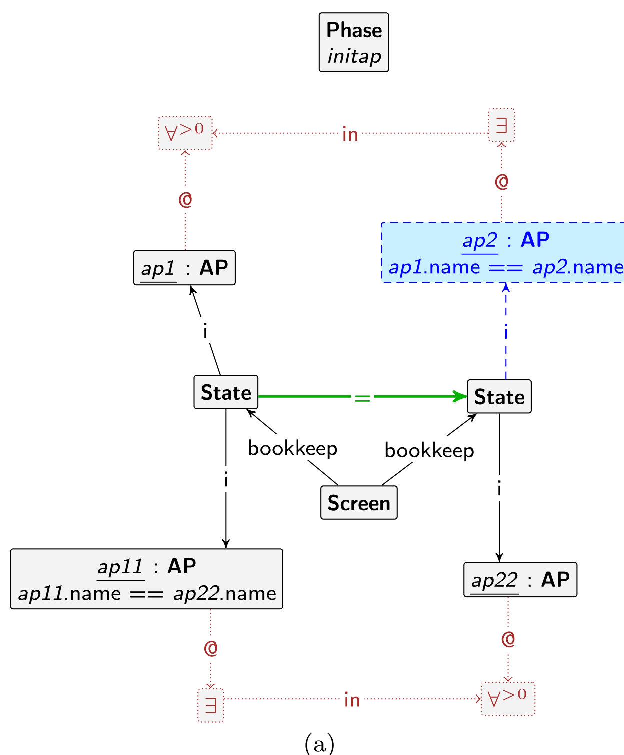


Fig. 17 Rule for the identification and merging of equivalent KTS states

3. Aggregate the path between that *Click Area* node and a *Screen* node using the rules shown in Fig. 15.
4. Create the target KTS state for the reached *Screen* node.
5. Connect the source KTS state and target KTS state by a labeled transition (cf. Fig. 16).

In addition to these steps, we have to take care of the following global transformation aspects:

– Identify and merge equivalent states in the target KTS (cf. Fig. 17).
– Fix-point recognition for detecting when a transformation can terminate. In essence, it requires to carefully control the order of transformation rule applications, in particular using the merge rule, in order to identify when a fixpoint has been reached. This is a quite tedious task, which, when the rules are modeled accordingly, is supported by Groove.
– Erasure of all the elements that are not part of the target model. This concerns elements of the source model (WebStory), as well as all the introduced auxiliary elements, e.g., for computation and bookkeeping. The corresponding transformation rules are typically quite straightforward, if one took care of adequately typing of all the involved artefacts (Fig. 18).

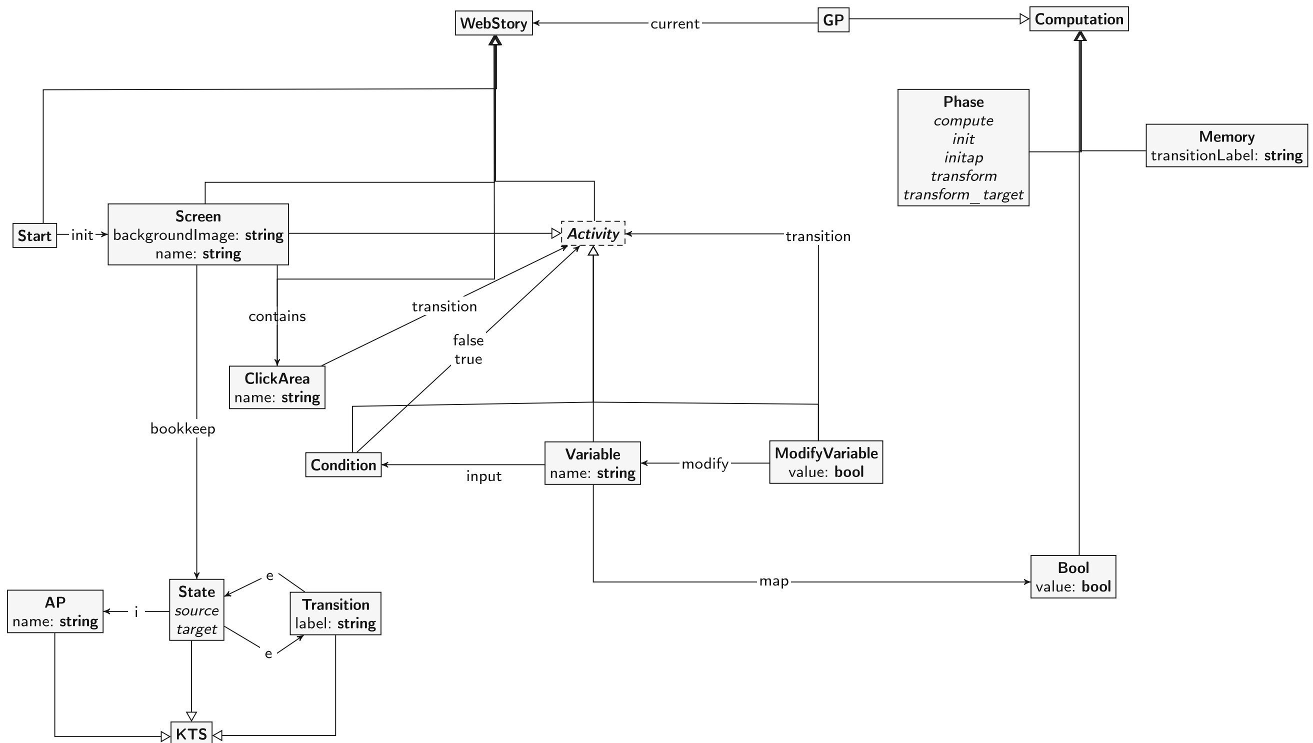


Fig. 18 The *Groove* type graph for the transformation from WebStory to KTS

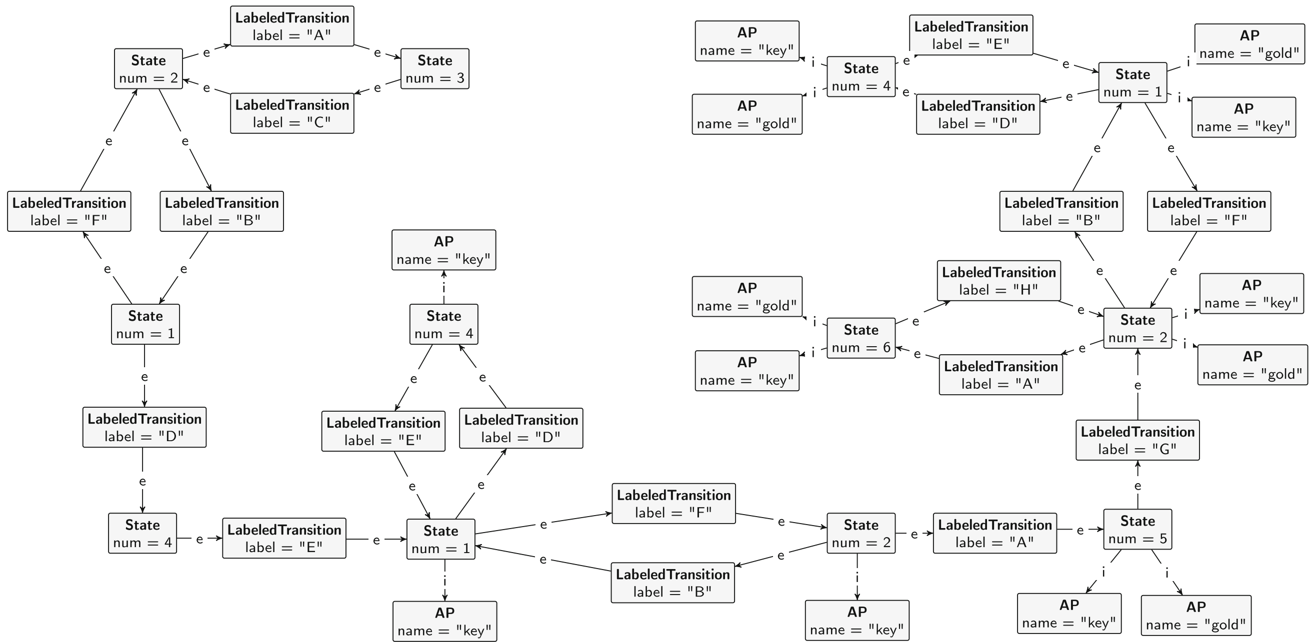


Fig. 19 Resulting KTS from the transformation using *Groove*

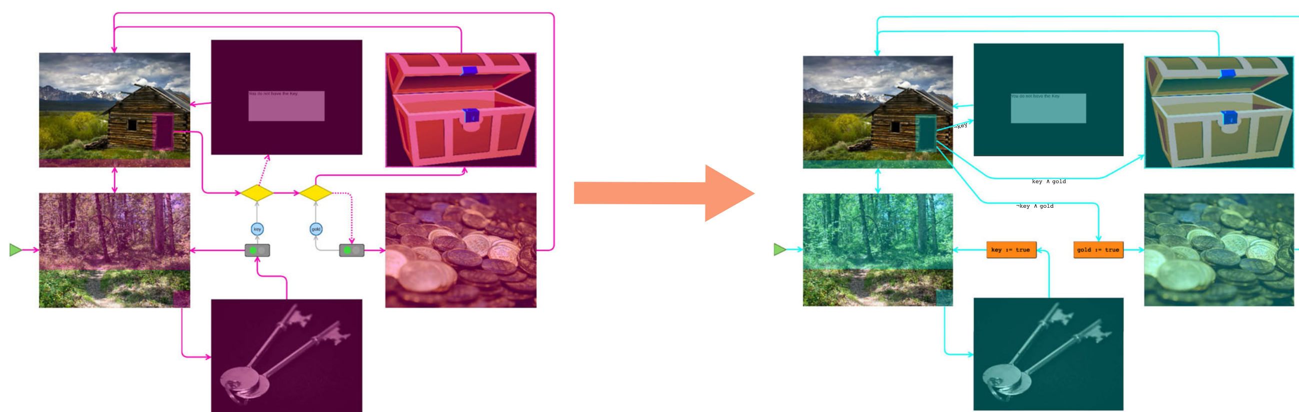


Fig. 20 Migration of the WSL after a metamodel change: removed *Condition* type, *Modify Variable* type, and *Variable* type, added an *Assignment* type and *Guarded Transition* type

Applying the transformation on the same path considered in Sect. 4.1 (the highlighted path in Fig. 2) with the same variable configuration ($[\![key]\!](\sigma) = true$, $[\![gold]\!](\sigma) = false$) results in a series of in-place graph transformations. An excerpt of the corresponding transformation sequence, which highlights the actual transformation steps while omitting the 'passive' context, is shown in Fig. 21.

The first part, Fig. 21a, shows the situation after the creation of the source KTS state. Fig. 21b represents two computation steps: the result after the application of the `ConditionTT` rule ('movement' of the **Current** edge from position *current* to *current*′) and `ConditionFF` rule ('movement' of the **Current** edge from position *current*′ to *current*″). The subsequent application of the `Modify Variable` rule results in the situation shown in Fig. 21c, where the value of the *Variable* 'gold' is set to `true` and the *current* edge refers to the *Screen* named 'Gold'.

Figure 21d shows the situation after the target KTS state was created and the source state and the target state are connected by a transition labeled with the name of the *Click Area* node 'A'.

The entire KTS resulting from the Webstory example is depicted in Fig. 19.

Using Groove, the main challenge was to model the transformation's execution which resulted in a big overhead. For instance, assuring a correct transformation using graph rewriting rules, confluence is an important requirement, i.e., if several rules are applicable in one situation the order in which these rules are applied must not have an effect on the final result. We solved this problem by adding additional structures to the input model and rules that prohibit transformation steps that might violate the confluence requirement. Interestingly, the rules responsible for the creation of KTS structures and the rules defining the atomic computational steps are straightforward (cf. Figs. 14, 15).

5.3 Comparison

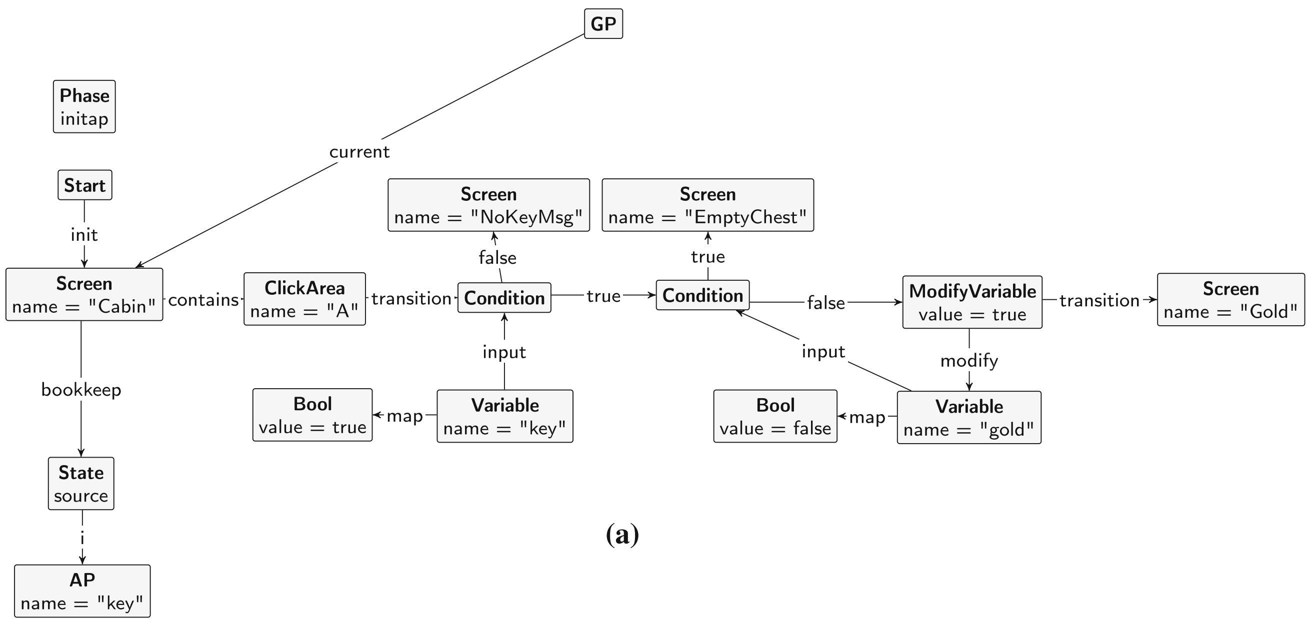
In order to discuss the state-of-the art of model-to-model transformation languages when it comes to transformations that comprise computational aspects like the one discussed here for the WebStory, we have investigated two prominent solutions:

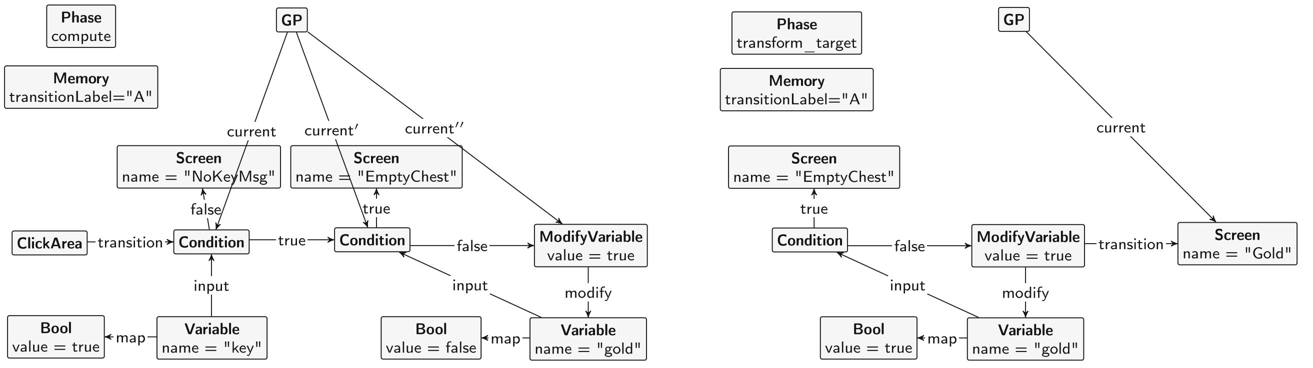
- ATL, a de-facto standard of the model-driven design community, and
- Groove, one of the leading tools for graph transformation.

Both solutions are known to elegantly capture static transformations like data model conversions (e.g. Families to Persons [47], Tree to Lists [47], Java to UML [4], or UML to RDB schema [10]). However, even transformations that appear to be more semantic (e.g. UML+OCL to Java [41] or Petri Nets from finite state automata [27] and state charts [14]) usually capture purely syntactic translations, which often operate on structurally quite similar models for which the mapping between elements is straightforward. However, treating computational structures is more intricate, as their correctness depends, e.g., on the concrete rule scheduling, an aspect transformation languages are meant to hide from the user:

- Using ATL, helper functionality is required which turn the ATL rule specification essentially into an imperative program (cf. Sect. 5.1), while
- Groove requires to model the rules scheduling via auxiliary structures in its graph format, which is not particularly well-suited for this purpose (cf. Sect. 5.2).

In contrast, the rule format of our SOS-based DSL for program transformation is designed for dealing with com-





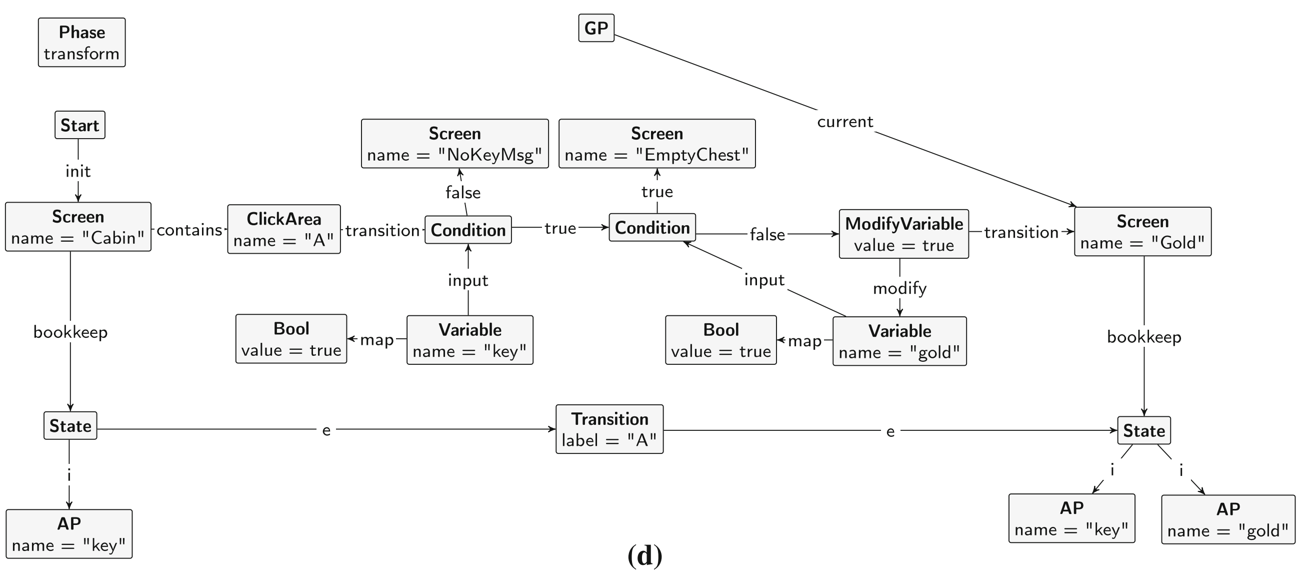


Fig. 21 Aggregation of the highlighted WebStory path depicted in Fig. 2

putational structures and does not require any complicated encodings.

6 Meta pattern of our approach

In this paper, we have introduced (and extensively discussed a single instantiation of) a meta pattern for transformation languages, which advocates a decomposition of graph transformations into aspect-specific transformation sub-DSLs. This meta pattern can be flexibly instantiated:

- The graphical source language: E.g., writing the rules for extracting the labeled transition system for a (graphical) Petri Net is an easy exercise.
- The not necessarily graphical target language: As indicated in Fig. 20, the graphical syntax that specifies dynamic links between the Web pages in the center of the left picture of Fig. 20 can also be nicely aggregated into a textual form that serves as edge labels in the right picture of Fig. 20.
- The purpose: E.g., the language-to-language transformation sketched in Fig. 20 is almost as easy as extracting a labeled transition systems from a Web story as discussed in this paper in detail. Other aspects, like, e.g., security, will require to define other dedicated sub-rule systems. The definition of such sub-rule systems and their modular use is topic of our current research.

In our experience, it is surprising how much can be specified with the very basic setting sketched here, and a lot more is possible in enriched settings, characterized, e.g., by allowing negative premises in SOS rules. In fact, there are workshop series that discuss the according potential (cf., e.g., the latest proceedings of EPTCS [12]). However, it is neither the goal of this paper to propose a dedicated, particularly expressive language, nor to investigate the possible range. Rather, we intend to illustrate the potential that comes with the purpose-oriented decomposition of rule-based specifications when using aspect-specific sub-rule systems.

With the TSOS format, we exemplified an instance of the meta pattern for dealing with the aspect of computation: The essence of the WSL-to-KTS transformation is the SOS-based partial evaluation of the structure that specifies the dynamic links between the Web pages in order to arrive at the right graph of Fig. 1. It was our goal to illustrate that achieving this effect with our computation-oriented sub-rule systems is straightforward, but very cumbersome in generic graph transformation systems like, e.g., ATL [47] and Groove [39].

7 Conclusions and perspectives

We have sketched a simplicity-oriented framework for graph-based language-to-language transformation which is characterized by its modular, graph-based representation, domain focus, and ease of specification. Technically, these properties are achieved by a clear separation of source language and target language patterns in the rules, purely additive transformation steps, and a separate rule system for treating dedicated aspects, like, e.g., computation. These restrictions lead to a localization of the changes imposed by individual steps and thereby maximize the parts of the models which remain invariant following our Archimedean point principle, which proposes to control change by understanding what remains invariant [45]. A detailed comparative discussion of ATL, a de-facto standard of the model-driven design community, and Groove, one of the leading tools for graph transformation, illustrates the impact of our simplicity-oriented approach.

It should be noted, however, that TSOS is just an instance of our framework which is meant to be instantiated with other purpose-specific formats and aspects at need. A corresponding study currently in progress for the purpose of model migration after a change of the underlying meta model shows very promising results. In that study, we investigate the migration of WSL models into models of a refined modeling language WSL' which allows one to label edges with guarded assignments as shown in the right picture of Fig. 20. Although the purpose of this transformation is quite different from the transformation presented in this paper it turns out that the migration transformation from WSL to WSL' can also elegantly be specified in the basic version TSOS. A dedicated article that covers migration as transformation purpose for our meta pattern is currently in preparation [6]. Of course, the migration step that introduces guarded assignments also builds on a computation-based aggregation. Other purposes may require (computation-based) expansion: E.g., a transformation for securing the communication of a Web application has to expand the model with security means. We are currently investigation how far our basic version of TSOS carries here, and where it needs to be extended (Fig. 21).

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copy-

right holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems, pp. 121–135. Springer, Berlin (2010)
2. Bakera, M., Margaria, T., Renner, C., Steffen, B.: Tool-supported enhancement of diagnosis in model-driven verification. Innov. Syst. Softw. Eng. **5**, 211–228 (2009). https://doi.org/10.1007/s11334-009-0091-6
3. Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The graph rewriting and transformation language: GReAT. Electron. Commun. EASST **1**, 1–8 (2007)
4. Bergmayr, A., Troya, J., Wimmer, M.: From out-place transformation evolution to in-place model patching. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 647–652. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642946
5. Brand, C., Gorning, M., Kaiser, T., Pasch, J., Wenz, M.: Graphiti - Development of high-quality graphical model editors. Eclipse Magazine (2011)
6. Busch, D., Kopetzki, D., Lybecait, M., Naujokat, S., Steffen, B.: Co-Evolution as Language-to-Language Transformation (2021). In preparation
7. Cinco SCCE Meta Tooling Suite (2018). http://cinco.scce.info. Accessed 31 May 2021
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
10. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**, 621–645 (2006)
11. Dachis, A.: Dsc00318 gold coins. https://www.flickr.com/photos/dachis/14569056769/ (2014). Available under Attribution 2.0 Generic (CC BY 2.0) https://creativecommons.org/licenses/by/2.0/
12. Dardha, O., Rot, J.: Proceedings Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, EXPRESS/SOS 2020, and 17th Workshop on Structural Operational SemanticsOnline, 31 August 2020. In: EXPRESS/SOS (2020)
13. Dmitriev, S.: Language oriented programming: the next programming paradigm. JetBrains onBoard Online Mag. **1**, 1–14 (2004). http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
14. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Cerioli, M. (ed.) Fundamental Approaches to Software Engineering, pp. 49–63. Springer, Berlin (2005)
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer, Berlin (2006)
16. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation. Springer, Berlin (2015)
17. Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.): Graph Transformations: 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27–October 2, 2010. Proceedings. Lecture Notes in Computer Science. Springer (2010). https://doi.org/10.1007/978-3-642-15928-2
18. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: language and environment. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools, pp. 551–603. World Scientific (1999)
19. Eysholdt, M., Behrens, H.: Xtext: Implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, pp. 307–309. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1869542.1869625
20. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018). https://doi.org/10.1145/3127323
21. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html (2005). Accessed 05 July 2020
22. Futamura, Y.: Partial evaluation of computation process-an approach to a compiler–compiler. Higher-Order Symb. Comput. **12**(4), 381–391 (1999). https://doi.org/10.1023/A:1010095604496
23. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008). https://doi.org/10.1016/j.scico.2007.08.002
24. Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. Softw. Syst. Model. (2018). https://doi.org/10.1007/s10270-018-0665-6
25. Knowles, C.: Salmon cabin and sawtooths. https://www.flickr.com/photos/theknowlesgallery/4756008375/ (2010). Available under Attribution 2.0 Generic (CC BY 2.0) https://creativecommons.org/licenses/by/2.0/
26. Kolovos, D., Rose, L., García-Domínguez, A., Paige, R.: The Epsilon Book. Published online: http://eclipse.org/epsilon/doc/book/ (2015). Last update: February 4, (2015)
27. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: Ghosh, S. (ed.) Models in Software Engineering, pp. 240–255. Springer, Berlin (2010)
28. Lybecait, M.: Meta-Model Based Generation of Domain-Specific Modeling Tools. Dissertation, TU Dortmund University, Dortmund, Germany (2019). https://doi.org/10.17877/DE290R-20418. http://hdl.handle.net/2003/38499
29. Lybecait, M., Kopetzki, D., Steffen, B.: Design for 'X' through model transformation. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), LNCS, vol. 11244, pp. 381–398. Springer (2018). https://doi.org/10.1007/978-3-030-03418-4_23
30. Lybecait, M., Kopetzki, D., Zweihoff, P., Fuhge, A., Naujokat, S., Steffen, B.: A tutorial introduction to graphical modeling and metamodeling with cinco. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), LNCS, vol. 11244, pp. 519–538. Springer (2018). https://doi.org/10.1007/978-3-030-03418-4_31
31. Margaria, T.: From computational thinking to constructive design with simple models. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), LNCS, vol. 11244, pp. 261–278. Springer (2018). https://doi.org/10.1007/978-3-030-03418-4_16
32. Margaria, T., Steffen, B.: Business process modelling in the jABC: the one-thing-approach. In: Cardoso, J., van der Aalst, W. (eds.) Handbook of Research on Business Process Modeling. IGI Global, Pennsylvania (2009)

33. Margaria, T., Steffen, B.: Simplicity as a driver for agile innovation. Computer **43**(6), 90–92 (2010). https://doi.org/10.1109/MC.2010.177
34. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional, Boston (2010)
35. Müller-Olm, M., Schmidt, D., Steffen, B.: Model-checking—a tutorial introduction. In: Proceedings of the 6th International Symposium on Static Analysis (SAS '99), pp. 330–354 (1999). https://doi.org/10.1007/3-540-48294-6_22
36. Naujokat, S.: Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools. Dissertation, TU Dortmund, Dortmund, Germany (2017). https://doi.org/10.17877/DE290R-18076. http://hdl.handle.net/2003/36060
37. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: A simplicity-driven approach to full generation of domain-specific graphical modeling tools. Softw. Tools Technol. Transf. **20**(3), 327–354 (2017). https://doi.org/10.1007/s10009-017-0453-6
38. Plotkin, G.D.: A Structural Approach to Operational Semantics. Tech. rep., University of Aarhus (1981). DAIMI FN-19
39. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance, pp. 479–485. Springer, Berlin (2004)
40. Rensink, A., Kuperus, J.: Repotting the geraniums: on nested graph transformation rules. ECEASST **18**, 4–15 (2009). https://doi.org/10.14279/tuj.eceasst.18.260
41. Sánchez Cuadrado, J.: Towards a family of model transformation languages. In: Hu, Z., de Lara, J. (eds.) Theory and Practice of Model Transformations, pp. 176–191. Springer, Berlin (2012)
42. Sierralupe, D.G.: Middle fork bike path. https://www.flickr.com/photos/sierralupe/29262085202/ (2016). Available under Attribution 2.0 Generic (CC BY 2.0) https://creativecommons.org/licenses/by/2.0/
43. Steffen, B.: Property-oriented expansion. In: Cousot, R., Schmidt, D.A. (eds.) Third International Symposium on Static Analysis (SAS '96), Lecture Notes in Computer Science, vol. 1145, pp. 22–41. Springer, Berlin (1996). https://doi.org/10.1007/3-540-61739-6_31
44. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000. Springer, Berlin (2019). https://doi.org/10.1007/978-3-319-91908-9_17
45. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. LNCS Trans. Found. Mastering Change (FoMaC) **1**(1), 22–46 (2016). https://doi.org/10.1007/978-3-319-46508-1_3
46. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
47. The Atlas Transformation Language. https://www.eclipse.org/atl/atlTransformations/ (2017). Accessed 09 Apr 2020
48. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007). https://doi.org/10.1016/j.scico.2007.05.004. Special Issue on Model Transformation
49. Ward, M.P.: Language oriented programming. Softw. Concepts Tools **15**(4), 147–161 (1994)
50. Watson, I.: Day 161 - keys. https://www.flickr.com/photos/dagoaty/4707352284/ (2010). Available under Attribution 2.0 Generic (CC BY 2.0) https://creativecommons.org/licenses/by/2.0/
51. Wing, J.M.: Computational thinking. Commun. ACM **49**(3), 33–35 (2006). https://doi.org/10.1145/1118178.1118215
52. Xtend - Modernized Java. http://xtend-lang.org. Accessed 8 Feb 2019
53. Zweihoff, P.: Cinco Products for the Web. Master thesis, TU Dortmund (2015)
54. Zweihoff, P., Naujokat, S., Steffen, B.: Pyro: Generating Domain-Specific Collaborative Online Modeling Environments. In: Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019) (2019). https://doi.org/10.1007/978-3-030-16722-6_6

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.