Electronic Communications of the EASST
Volume 82 (2022)

11th International Symposium
on Leveraging Applications of Formal Methods, Verification and Validation
-
Doctoral Symposium, 2022

SOS-Supported Graph Transformation

Sebastian Teumert, Marvin Krause, Bernhard Steffen

22 pages

Guest Editors: Sven Jörges, Salim Saay, Steven Smyth
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

SOS-Supported Graph Transformation

Sebastian Teumert, Marvin Krause, Bernhard Steffen

TU Dortmund University

Abstract: In this paper, we propose a simplicity-oriented approach for model-to-model transformations of graphical languages. Key to simplicity is decomposing the rule system into two rule sub-systems that separate purpose-specific aspects (transformation and computation), and specifying these rule systems as a graphical language. For the transformational aspect, we use a compiler-like generation approach, while taking Plotkin's Structural Operational Semantics (SOS) as inspiration for the computational aspect. We define these rule systems as inference rules for pattern-based transformations of typed, hierarchical graphs. Using typed graphs allows patterns to easily distinguish between the elements of the source graph. The resulting rule system (named *SOS-Supported Graph Transformation*, or `SOS-GT`) supports a well-structured and intuitive specification of complex model-to-model transformations adequate for a variety of use cases. We illustrate these rules with an example of transforming the WebStory language (WSL, an educational tool) to a Kripke Transition System (KTS) suitable for model checking, and give an overview over more applications in the end of the paper.

Keywords: Multi-level transformations, Model-to-model transformation, Graph rewriting, Structural Operational Semantics, Abstraction, Rule systems, Meta language, Graph pattern

1 Introduction

A commonly faced problem in the world of model-driven development and language-driven development (LDE) [SGNM19] is interpreting a model. A model may be interpreted to execute it, to generate code from it, to convert it to a textual representation or to convert it to another (graphical) model, either in the same or another language (e.g. for language evolution and co-evolution). In this paper, we present a novel approach to the latter problem using a rule system in which the purpose-specific aspects of transformation and computation are realized as separate subsystems. An important driver of this approach is simplicity [MS10] and cross-domain work of experts similar to the approach used in [KLNS21].

Both aspects are addressed with rule systems that are based on inference rules. The computational aspects are addressed by a rule system that is inspired by Plotkin's Structural Operational Semantics (SOS) [Plo81], while the transformational aspect is realized by a rule system that uses pattern-matching to find elements in the source model and uses template-based production rules to generate elements in a separate target model. Thus, the source model is not *rewritten*, but only inspected, while the target model is constructed incrementally.

The subject is of great interest for the LDE community, because the aim of LDE is to enable domain experts to express themselves in a way that is accessible to them, alleviating the need to

acquire extensive programming knowledge.

Current state-of-the art transformation tools are usually textual and often require a significant amount of textual programming and complexity [Ren04, BNBK07, ERRS10, ERT99]. By bringing both the transformation and computation onto the same meta-level as the domain-specific languages (DSL) the domain experts use, those experts can express both aspects in a way that utilizes their domain knowledge and expertise in the (graphical) languages they have already acquired, instead of forcing them to learn a new, unfamiliar skill.

We demonstrate the usefulness of the rule system using the same example as [KLNS21] - the transformation of a graphical language called WebStory Language (WSL), designed for teaching purposes, to a Kripke transition system (KTS) - leading to the possibility of model checking the original language.

To this end, we introduce the required background information in Section 2, where we introduce the language workbench we used (Section 2.1), the WSL (Section 2.3), the target language KTS (Section 2.4) and the underlying data structure of all models (Section 2.2). Our main result is found in Section 3, where we first lay out some requirements (Section 3.1), introduce the intermediate language (Section 3.2) and rule system (Section 3.3), the context and semantic function (Section 3.4), both sub rule-systems (Section 3.5, Section 3.6) and give an overview over the whole transformation process (Section 3.7). In Section 4, we evaluate the approach and highlight its limitations, while we outline the future work in **??** and offer some conclusions and perspectives in Section 6.

2 Background

First, we discuss the basics required for understanding the main contributions presented in Section 3. Working with languages requires powerful frameworks for the realization and evolution of the required domain-specific development environments, often called *language workbenches* [Fow05]. Section 2.1 introduces CINCO [NLKS17], the language workbench used throughout this paper, while Section 2.2 discusses the type of model that CINCO uses and which can be used with this transformation approach in more detail. Section 2.3 introduces the WebStory language, an educational language that is used as example throughout the paper, while Section 2.4 introduces Kripke-Transition Systems, which are used as example for a transformation target throughout this paper.

2.1 CINCO Meta Tooling Suite

The CINCO *Meta Tooling Suite*[1] [NLKS17] is a tool for generating domain-specific graphical modeling environments, which we used for our implementation and to define our graphical rule language. CINCO works with typed hierarchical attribute graphs which can be defined by describing their nature on the meta level. It is built upon the Eclipse Modeling Framework (EMF) [SBPM08] and the RCP [MLA10] and uses Xtext-based languages for the metamodel definition as well as integration of textual DSLs in a service-oriented fashion [Nau17].

[1] Cinco SCCE Meta Tooling Suite. http://cinco.scce.info.

2.2 Typed Hierarchical Attribute Graphs

A *Typed Attribute Graph* (G,t) is a graph $G = (V,E)$ and a graph morphism $t : G \rightarrow ATG$ [EEPT06, HEGO10]. $ATG = (TG,Z)$ is an *Attributed Type Graph* made up of a *Type Graph* TG, which contains the node and edge types and a data signature Z which holds the attribute types. We use an *E-Graph* which extends G by connecting *data nodes* to its elements using *attribute edges* and enables the possibility to assign attribute values. We also need the concept of containment which is not offered by the Typed Attribute Graph itself. Therefore a containment relation is represented by an additional edge type in TG called the *contains*-edge [KLNS21]. This makes the graph hierarchical and it will be called *Typed Hierarchical Attribute Graph* (THAG) in the following.

In this paper we are working with models that can be described as THAGs. In CINCO, the types are given by the *metamodel* as described in the so-called *Meta Graph Language* (MGL) and allow for a subtyping relationship. The models that are transformed by SOS-Supported Graph Transformation (`SOS-GT`, cf. Section 3) are instances of the given metamodel. The patterns used in the rule system also refer to these types to restrict which elements are a match for each pattern.

2.3 WebStory Language

The WebStory Language (WSL) is a graphical language built for show case and teaching purposes which has been created with CINCO [LKZ+18]. With the WSL it is possible to define simple click adventure-like games in the web browser where the user can click on certain areas on a background image. When clicking such a click area, the data flow is evaluated and the user is presented with a new screen. The screen that is reached may depend on the value of variables if the data flow gos through a `Condition` node. The value of variables may be changed if the data flow goes through a `ModifyVariable` node in between two screens.

Consider Figure 1, which shows an example WSL model that we will refer to throughout this paper. All annotations (letters $A-F$, numbers 1-8, m_1, m_2, c_1, c_2) are not part of the original WSL model and only added for explanation purposes.

The WSL is made up of the following building blocks: The big rectangles with the images inside are so-called `Screens`. `Screens` define the background images the user can explore in the browser.

The green triangle is the `StartMarker` which is connected to the first `Screen` the user will see. Inside of the `Screen` are `ClickAreas` displayed in purple and numbered with the numbers 1-8 which define the clickable areas on the different background images. There can be an arbitrary number of `ClickAreas` inside of every `Screen`. Note that the `Screen` *A* and *B* contain two `ClickAreas` each and `Screen` *C-F* are all covered by a single `ClickArea`. There are two types of `ClickArea` available, `RectangleClickArea` and `EllipseClickArea`, which both inherit from the abstract super type `ClickArea`. In this example, however, only `RectangleClickAreas` are used.

The blue circles are `Variables`. In this example there are the two `Variables` *key* (left) and *gold* (right). `Variables` are of type boolean and have the default value `false`. The value of a `Variable` can be changed with the `ModifyVariable` node which looks simi-

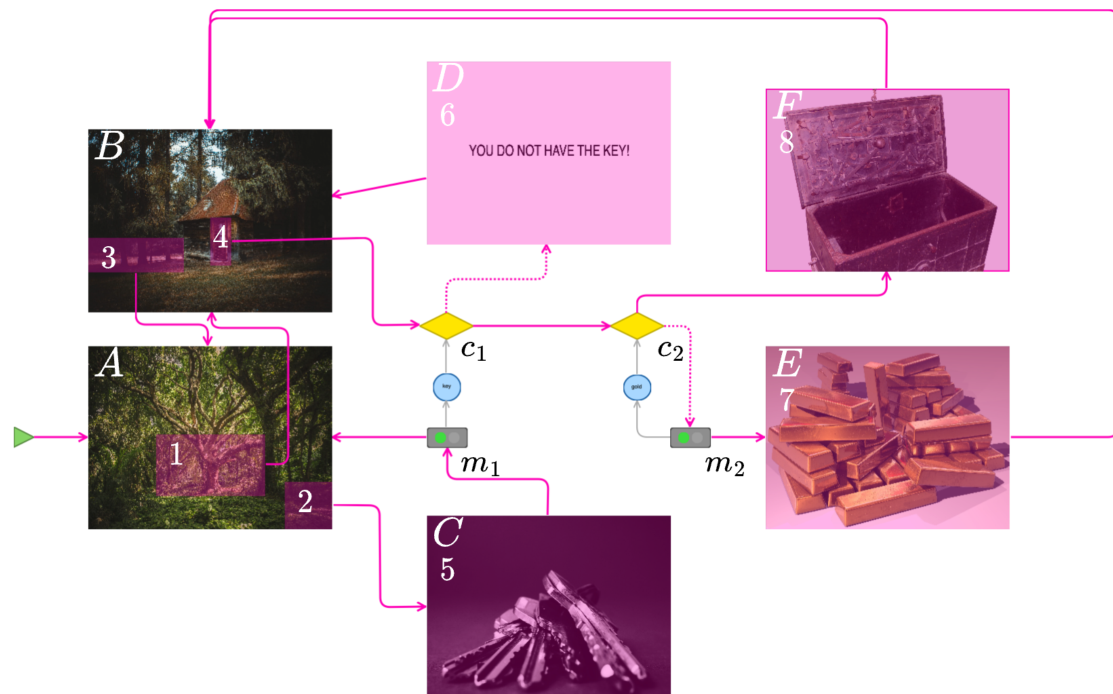

Figure 1: An example WSL model with six screens and two variables[2].

lar to a little traffic light and are also subsequently annotated here with m_1 and m_2. A green `ModifyVariable` assigns the value `true` to the connected `Variable`, a red one (which is not included in this example) assigns the value `false`. The yellow diamonds (c_1 and c_2) are `Condition` nodes and read from the `Variables` they are connected to. There are two outgoing edges for each `Condition`: The `FalseTransition` (dotted) which is traversed when the `Variable` has the value `false` and the `TrueTransition` (continuous) which is traversed when the `Variable` has the value `true`.

The purple edges all symbolize control flow. For example, by clicking on a `ClickArea`, a new `Screen` can be reached directly or the value of a `Variable` can be written or read first. The grey edges represent data flow, i.e. writing to or reading from a `Variable`.

[2] Images are taken from the following sources (all last accessed on 2023-03-10):

- *Nahaufnahme der Schlüssel* by George Becker (available under Pexels license): https://www.pexels.com/de-de/foto/nahaufnahme-der-schlussel-333837/
- *Truhe, Schatzkiste, Mittelalter* by Momentmal (available under Pixabay license): https://pixabay.com/de/photos/truhe-schatzkiste-mittelalter-2512108/
- *Wooden House in a Forest* by Mateas Petru (available under Pexels license): https://www.pexels.com/photo/wooden-house-in-a-forest-673788/
- *Goldbarren* Lot by Pexels (available under Creative Commons Zero license): https://www.pexels.com/de-de/foto/goldbarren-lot-47047/
- *Grüner Laubbaum* by veeterzy (available under Pexels license): https://www.pexels.com/de-de/foto/gruner-laubbaum-38136/

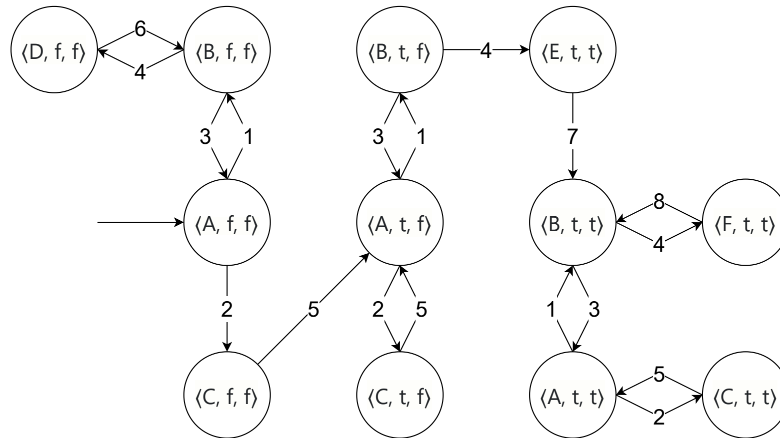

Figure 2: The KTS of the WSL model shown in Figure 1.

In this example, the user explores a forest. He starts on `Screen` A and finds a forest hut on `Screen` B by clicking on `ClickArea` 1. If the user tries to open the door (`ClickArea` 4), the value of the `Variable` *key* is read. At this point, the value is `false` and the user is redirected to `Screen` D and told to find the keys for the door. Every click now leads back to `Screen` B. In order to open the door, the user has to go back to `Screen` A via `ClickArea` 3 and then to `Screen` C via `ClickArea` 2 to find the keys. Each click now writes the value `true` into the `Variable` *key* and redirects the user to `Screen` A again. If the user tries to open the door one more time, the `TrueTransition` is traversed and the value of the `Variable` *gold* is evaluated. Since this has the value `false`, a `ModifyVariable` node is reached immediately afterwards, which writes the value `true` to the `Variable` *gold* before the user finally finds the treasure on `Screen` E. Another click takes the user back to `Screen` B. If the user tries to find the treasure again, he will only find an empty treasure chest (`Screen` F), because the evaluation of the `Variable` *gold* this time returns `true`.

2.4 Kripke Transition System

Kripke Transitions Systems (KTSs) (a generalization of both Kripke structures and labeled transition systems [MSS99]), had already been identified as providing an adequate semantic model structure for graphical program models [MS09].

Given AP a set of atomic propositions a Kripke Transition System (KTS) is a quintuple $KTS = (S, I, Act, R, L)$ where

- S is a set of states,
- $I \subseteq S$ is a set of initial states,
- Act is a set of actions,
- $R \subseteq S \times Act \times S$ is a transition relation and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

Figure 2 shows the KTS of the WSL model displayed in Figure 1. Here the labeling consists of the current `Screen` and the values of the `Variables` *key* and *gold* (with f = `false` and t = `true`). The only initial state ($\langle A, f, f \rangle$) is marked with an incoming edge without a source. The actions represent the clicking of the different `ClickAreas` and are shown here as edge labels with the numbers representing the `ClickArea` clicked.

3 SOS-Supported Graph Transformation

In this section we look in detail at the results of our work. For this purpose, we first describe the goals we set in advance before we continue with the individual components of our rule system and the functioning of the transformation calculus.

3.1 Goals

The main goal to be achieved was the conceptualization of a calculus that allows to transform one typed hierarchical attribute graph into another. As already described in Section 1, some approaches already exist for this purpose, whereby we mainly picked up the work of Kopetzki et al. [KLNS21] and wanted to continue or rather develop it one step further. Our minimum requirement was therefore to be able to achieve at least equivalent results. For this, we used the same example, the transformation from WebStory Language to KTS, as a guideline and wanted to be able to produce a model that could be validated in a simple way using standard model checkers, e.g. GEAR [BMRS09].

In addition, we have considered other use cases of such a calculus which may not have been covered before. This included, for instance, the simplification of models, i.e. the transformation within a single model type in which, for example, unneeded information is removed from the model or structures are represented differently (e.g. serialization). An inherently similar application is forward migration of models, i.e., updating graph models to another version through the transformation process, where the source and target models then often differ in only a few aspects.

Unlike many other approaches, we wanted to avoid so-called rewriting, i.e. the transformation directly in the source model, and leave the source model untouched during the entire transformation. Thus, the source model should be read-only while the results are written exclusively to the target model, making our approach work in a compiler-like manner.

Apart from that, a user of our tool should be able to define the transformation itself by creating graphical transformation rules. These rules should be easy to design and understand for domain experts, or at least support communication about the transformation process with them. In this way, it should be possible to work out a descriptive and easy to realize solution for the use cases described above, which in practice are often of a very technical and sometimes highly complicated nature. As in [KLNS21], the graphical rule language is inspired by Plotkin's SOS or, more generally, by inference rules.

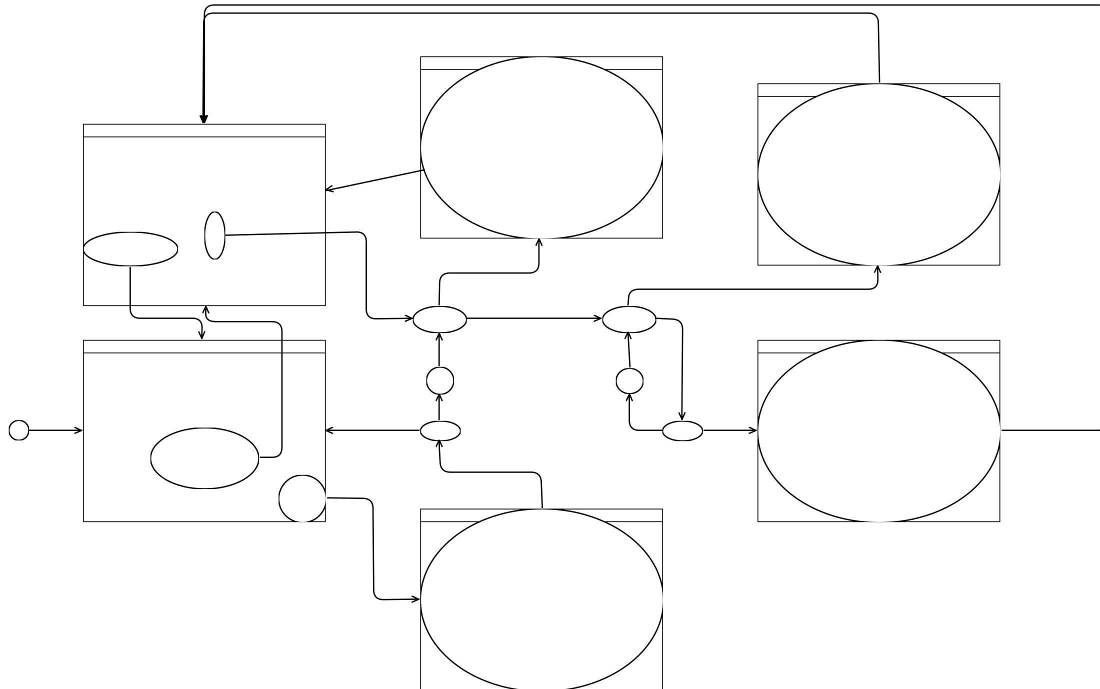

Figure 3: The WSL model of Figure 1 transformed into an IL model.

3.2 Intermediate Language

Both for the definition of the rules and for the transformation, we decided to use a uniform *Intermediate Language* (IL) instead of working directly on the source or target language. Every source model is first transformed into this IL and each target model has been generated from an IL model. The IL model consists only of three simple elements: `ILNode`, `ILContainer` and `ILEdge`, which are then parameterized to represent the elements of the original source model. This preserves any information from the original model and $\mathrm{IL}(A)$ becomes a parameterized metamodel of a language A. Figure 3 shows the IL model of WSL model seen earlier in Figure 1.

Utilizing an IL is desirable both from an implementation as well as a conceptual point of view. On the conceptual layer, it allows for IL enrichment. Thus, source models can be pre-processed and enriched with additional information prior to transformation. It also allows for enrichment of the target model when needed, e.g. to better discriminate target elements during transformation. Utilizing an IL also significantly eases implementation, both for languages in our own ecosystem, as well as languages defined in other ecosystems. By utilizing the IL, we are free to import any model that is compatible with being looked at as a typed, hierarchical graph. Here, a language is compatible when it is as strong as the definition, or weaker. For example, languages that have no hierarchies or no attributes are still compatible with our definition and can be translated to our IL. Even though both the IL and our rule system are model types defined in CINCO, this opens up the possibility to partially decouple from CINCO models in the long term and to open up to other ecosystems.

3.3 Rule System

In order to achieve all of these goals, a rule system with two types of rules is used. The rule system is based on *inference rules*, where a *premise* must hold in order for the *conclusion* to be reached, under an optional *(side-) condition*. For the method presented herein, we derived two different kinds of inference rules, namely *transformation rules* and *computation rules*.

These rules are defined as transformation not directly on the source and target model, but the intermediate language introduced above. As such, the complete ruleset represents the transformation from $\mathrm{IL}(A)$ to $\mathrm{IL}(B)$, where A is the source metamodel and B is the target metamodel. Hence we write $RS(A,B) : \mathrm{IL}(A) \mapsto \mathrm{IL}(B)$, where $RS(\cdot,\cdot)$ is the whole ruleset consisting of both transformation and computation rules.

$$\frac{\text{Configuration (-Transition)} \qquad [\text{Condition}]}{\text{Source Pattern} \rightarrow \text{Target Production}}$$

Figure 4: General Scheme for Transformation Rules

Transformation rules (see Figure 4) are used to *match* a pattern in the source model under a specific *configuration* and *produce* a certain structure in the target model when the premise for the transformation holds and the (side-) condition is true. The premise of a transformation rule is checked via the computation rules and describes a required transition from one *configuration* to another (Section 3.4). Transformation rules are discussed in more detail in Section 3.5.

$$\frac{\text{Source Pattern} \qquad [\text{Condition}]}{\text{Start-Configuration} \rightarrow \text{End-Configuration}}$$

Figure 5: General Scheme for Computation Rule

Computation rules (see Figure 5) describe the transition from a *starting configuration* to an *end configuration* in their conclusion, under the premise that the structure described in the *source pattern* of the rule can be found in the source graph and the (side-) condition holds. They are discussed in more detail in Section 3.6

3.4 Configurations and State of the Computation

The state of the current transformation process is recorded in a *working set* of *configurations*. A configuration is a tuple $\langle l, \sigma \rangle$, where l is a location in the source model ($l \in V \cup E$) and σ is a key-value *store* to be tracked during computation. In order for the process to terminate, σ needs to have a finite domain.

At the beginning of the transformation process, an initial σ needs to be specified. The starting configuration is $\langle \varepsilon, \sigma_0 \rangle$, where ε is a special value for *any* location and σ_0 is the initial store, which can be specified by the user.

We define two different operations for the store, which shall be used later: interpretation and substitution. For *interpretation* of a variable v under a given store σ, we write $[\![v]\!](\sigma)$ and mean ”The value stored under the key v in the store σ”. For *substitution*, we write $\sigma\{v/k\}$ and mean ”the new store created by replacing the value v stored under the key k in the store σ”.

We also define the transition between two subsequent configurations as $\rightarrow^*$. We say that $a \rightarrow^* b$ if and only if either $a = b$ or there exists a series of one or more computation steps as evaluated by the computation rules, that transforms a to b[3]. Note that this relation is *transitive*, so that $a \rightarrow^* b \wedge b \rightarrow^* c \Rightarrow a \rightarrow^* c$.

3.5 Transformation Rules

In transformation rules, the conclusion consists of a source pattern and a target production, with the intended semantic that when the source pattern is found in the source model, then the target production should be found in the generated target model as well. The premise places constraints on the current configuration or the required transition of the configuration to another configuration.

Transformation rules are evaluated by matching their source pattern under the current configuration, then checking the premise using the computation rules, and then, when the premise and side condition hold, identifying the already existing target elements of the target production in the target model, and generating all missing elements that have not yet been generated.

3.5.1 Source Pattern

Source patterns are described as partial graphs in the IL of the source metamodel, and consist of nodes, edges and containers. Each element of the source pattern is annotated with a *label* and *type* from the metamodel of the source model. The labels are later used in the premise and condition as metavariables. When a source pattern is successfully matched in the source model, these metavariables then refer to the concretely matched element in the source graph. The type annotations are used to constrain which kinds of elements can be matched in the source graph. Type annotations allow for subtyping: in order for an element to be allowed to match, it must have the same type as the type annotation, or its type must be compatible with the type annotation in a standard subtyping relationship [4]. The subtyping relationship of the source metamodel is enriched with a new supremum, called `Any`. This element is the supertype of every element in the source metamodel and can thus be used as a wildcard for type assertions.

Additionally, constraints can be placed on the *attributes* of elements[5]. Elements of the source model which do not fulfill these constraints on their attributes cannot be used to instantiate a match for the given pattern.

A match in the source model is found if and only if the elements have the same structure as in the source pattern. They must match in (1) Type, (2) Attributes, (3) Containment and

[3] In the future, we might also define the relation $a \rightarrow^k b$, which restricts execution to exactly k steps, or relations like $a \rightarrow^{<k} b$ and $a \rightarrow^{>k} b$ with the analogous semantics.

[4] Since we implemented this in CINCO, the subtyping rules of CINCO do apply in our case.

[5] We do not prescribe any concrete mechanism for this. In our implementation, attribute constraints are attached to each rule and specified using a small DSL that can access the metavariables from the source pattern.

(4) Connections. Note that source patterns are partial and of an existential nature. It is legal for a node or container to have *more* edge connections than shown in the source pattern, but not less. It is legal for nodes and containers to be contained in other elements not in the source pattern, but if the source pattern prescribes a concrete containment relationship between two or more elements, this relation must be found as given (transitive containment is not legal). Source patterns must always be matched completely, with every element uniquely identified for any given instance of the match.

Source patterns *may* contain a special element, called the *anchor*. If an element of the source pattern is marked as the *anchor*, the rule is said to be a *local* rule, otherwise, it is said to be a *global* rule. Global rules can match the *any* (ε) location in a configuration, while local rules cannot. In order for a source pattern of a local rule to be a match, its anchor *must* be matched with the location of the current configuration.

3.5.2 Premise and Condition

The premise of a transformation rule either consists of a single configuration $\langle l, \sigma \rangle$, or a configuration transition $\langle l, \sigma \rangle \rightarrow^* \langle t, \tau \rangle$. The leftmost configuration is called the *start configuration*, and the (optional) right hand configuration is called the *end configuration*. The location of the start configuration always refers to a label in the source pattern. The variables introduced in the configurations can be referenced in the (side-) condition. Both start and end configuration (if it exists) must also fulfill the (side-) condition.

Valid constraints to be placed upon the configurations can either constrain the type of the location by a type assertion, or the values stored in the store (using the $[\![\cdot]\!](\cdot)$ relation)[6]. The $\rightarrow^*$ relation is trivially true if the start configuration already fulfills the constraints of the end configuration, since zero evaluation steps are explicitly included in its definition. Otherwise, the computation rules are used to determine if the $\rightarrow^*$ relation specified in the premise holds.

3.5.3 Target Production

Target productions are described as partial models in the IL of the target model, and consist of nodes, edges and containers. Each element of the target production is annotated with a *label* and *type* from the metamodel of the target model. All types must be concrete, non-abstract and thus instantiable types of the target metamodel. Using non-instantiable types in the target production is illegal, since the presence of even one such uninstantiable element makes the whole target model uninstantiable. The labels can be used to reference the elements, e.g. to describe the values of their *attributes*[7]. Furthermore, elements in the target production are annotated with a configuration under which they are produced. These configurations do not necessarily need to be start- or end configuration, but can be any valid configuration.

Only those elements of the target production which do not yet exist in the target model are generated. If an element of a target production already exists in the target model, the existing

[6] In our concrete implementation, this includes boolean terms, numeric terms and string comparisons, as well as nested terms with $\wedge$, $\vee$ and brackets.

[7] Again, we do not prescribe a concrete mechanism for this. Our implementation uses a simple DSL that utilizes the metavariables of the pattern to assign values to its attributes.

element is instead re-used and the rest of the target production is generated around the already existing elements. In order to do this, the target production is first treated like a pattern and matched within the target model. A match in the target is found, if and only if the elements have the same structure as in the target production: they must match in (1) Type, (2) Attributes, (3) Containment, (4) Connections and (5) Configuration. Note that target productions are partial and of an existential nature. It is legal for a node or container to have *more* edge connections then shown in the target production, but also less, since the missing edge connections will then be created. It is legal for nodes and containers to be contained in other elements not in the target production, but if the target production prescribes a concrete containment relationship between two or more elements, this relation must be found as given (transitive containment is not legal). The match is always *maximal*, meaning that if it is possible to match an element of the target model to the match, the element is included in the match.

After a (partial) match for the target production is found, then the missing elements are generated. Note that elements in the target model are *never* mutated, they are never deleted, reparented or their attributes changed. Edges are never reconnected to other elements once created. If a target production matches an element in all but their attributes, the element is not a match. The attributes are not overridden, but instead a new element is generated that has the appropriate attributes[8].

3.5.4 Transformation Rules for WebStory to KTS

In this section, we show and informally describe a concrete example of a set of transformation rules defined using the schema laid out before, in our concrete syntax as used in our implementation of this calculus. Figure 6 shows the two transformation rules as used for the transformation of WSL to KTS.

The `Initial-Screen` rule matches a `Screen` (with the label L) in the source model that directly follows the (unique) `StartMarker` element (with the label s) in its source pattern. The pattern does not have a designated anchor, thus the rule is a global rule and can match the *any* (ε) location. In the premise, it only uses a single configuration $\langle L, \sigma \rangle$. The configuration has to exist, and the (side-) condition asserts that the interpretation of all `Variables` that exist in the source model (where `Variable` is a type in the WSL metamodel, cf. Section 2.3) must be $true$. If premise and condition hold, the rule produces the initial state of the KTS in the target model, under the configuration $\langle L, \sigma \rangle$.

Moving on to the `Screen-to-Screen` rule, this rule matches a `ClickArea` (with the label a), contained in some `Screen` (with the label L), as well as *any* element it is connected to (with the label l). Since a is asserted to be a `ClickArea`, which is an abstract type with two concrete subtypes – `RectangleClickArea` and `EllipseClickArea`, it can match nodes from both of these types. The `Screen` labeled L has a bold outline, indicating that this source pattern has a designated anchor of L. Thus, it is only applicable for configurations whose location is a screen and can be matched to L. The premise and condition together assert that after zero or more steps in the $\rightarrow^*$ relation (evaluated via computation rules), the end-configuration is in a location labeled S for which the type assertion `S : Screen` holds. When this is true,

[8] Setting attributes is not shown in the image, but in both rules, the generated states are labelled with a letter representing the `Screen`, and boolean values representing the currently stored values for each `Variable` in the store.

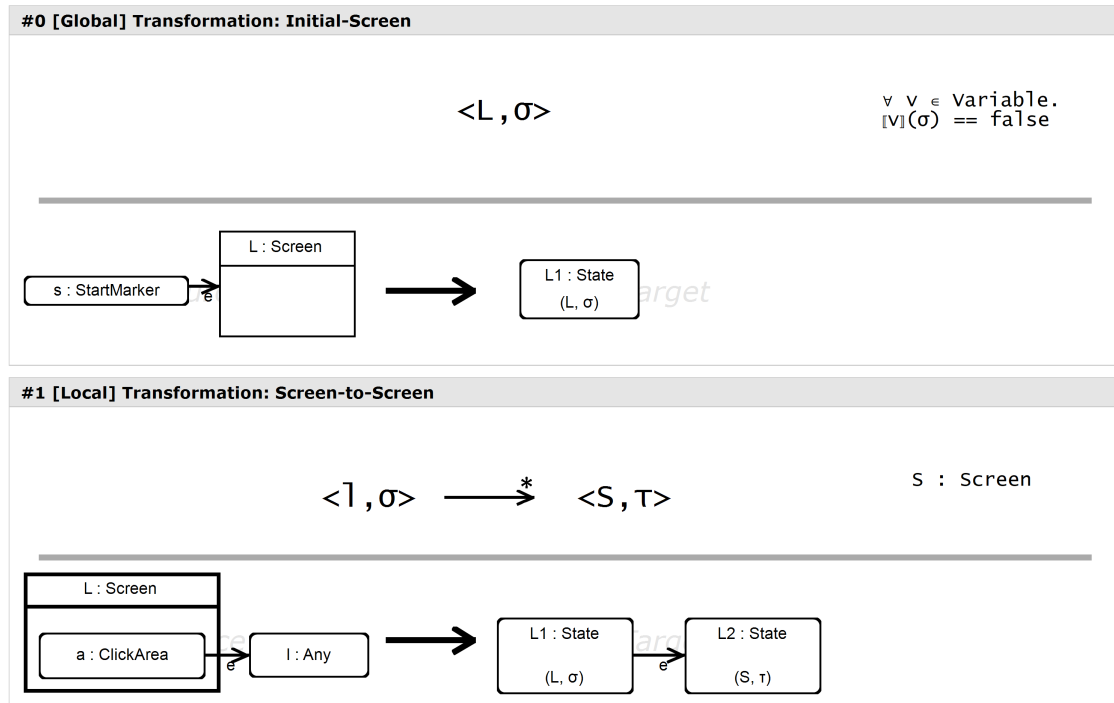

Figure 6: Transformation rules to transform WSL to KTS, with the two rules `Initial-Screen` and `Screen-to-Screen`.

the target production specifies the goals that the target model should meet. It should contain a KTS state representing the starting configuration, as well as a KTS state representing the end configuration, and both shall be connected via a transition edge.

3.6 Computation Rules

Computation rules are used to evaluate the $\rightarrow^*$ relation and are thus utilized when the premise of a transformation rule needs to be checked and does not trivially hold (after zero steps). Hence, their conclusion describes the transition of one configuration to another configuration, which are called the start and end configuration, respectively. The premise of a computation rule is a source pattern, matching a pattern in the source model.

3.6.1 Conclusion

Computation rules are always invoked with a given configuration – either from a transformation rule, or with the configuration reached by a previous computation rule. The start configuration has the form $\langle l, \sigma \rangle$ and introduces meta-variables for the location and store of the start configuration, which can be used later in the condition and in the source pattern. The end configuration is given as $\langle l, \sigma \rangle$ as well. The location of the end configuration refers to a metavariable introduced in the source pattern, or, when the location should be unchanged, the location as given in the start

configuration. The store of the end configuration can either refer to the store as used in the start configuration (if the store does not change), or introduce a new metavariable for the final store, which can be used to describe how the new store should look like. The condition (see below) can place additional requirements on the final store, expressed using the store substitution introduced in Section 3.4.

3.6.2 Source Pattern and Condition

The premise of a computation rule is a source pattern which works analogous to source patterns in transformation rules. They are partial graphs to be matched in the source model, annotated with labels and type assertions. As in the source patterns of transformation rules, these labels can later be used in the condition, but also in the description of the configuration transition. Type assertions support subtyping as well, and the same matching rules as for source patterns in transformation rules are used for source patterns in computation rules. Unlike in transformation rules, the anchor *is not optional* though, and source patterns in computation rules *always* have a designated anchor element in their pattern.

Conditions in computation rules work analogous to conditions in transformation rules (cf. Section 3.5.2).

3.6.3 Computation Rules for WebStory to KTS

Figure 7 shows the three computation rules as used for the transformation of WSL to KTS. To recall, in the WSL, there are three possible target nodes that `DataFlow` edges can reach: `Screen` containers, `ModifyVariable` nodes and `Condition` nodes. If a `DataFlow` edge directly connects a `ClickArea` to another `Screen`, the transformation of the `Screen-to-Screen` rule is directly applicable. The computation rules reflect the other possibilities in which a WSL model may be constructed. Thus, there is one rule for the `ModifyVariable` node. In case of `Condition` nodes, there are two possible paths after them, depending on the value the given variable has in the store at that moment (a *boolean* value), resulting in a total of three computation rules needed to fully cover all possible constructions of the WSL.

The `ModifyVariable` rule is applicable when the location of the current configuration is a `ModifyVariable` node (labeled m). The type constrained on m is expressed in the source pattern. The node labeled m there is also bold because it represents the anchor of the pattern. The pattern also matches the `Variable` to be modified as v, and the subsequently reached node (which again, can either be a `Screen`, `ModifyVariable` or `Condition` node) as l, which can be of *any* type. The conclusion shows the transition from the start configuration to an end configuration in which both location and store have changed. The location changes to the element labeled l, while in the store τ, the value of the variable v is replaced with the value of the `ModifyVariable` node[9].

Either of the `Condition-[True|False]` rules is applicable when the current configuration contains a location that is of type `Condition`, which is matched as anchor L in the source pattern. Either rule matches the `Variable` node with the label v. However, the (side-)

[9] In our implementation, we use $[\![n]\!]$ to mean "the value of the attribute `value` of the node referenced by the label n, and $[\![n|atr]\!]$ to mean "the value of the attribute atr of the node referenced by the label n.

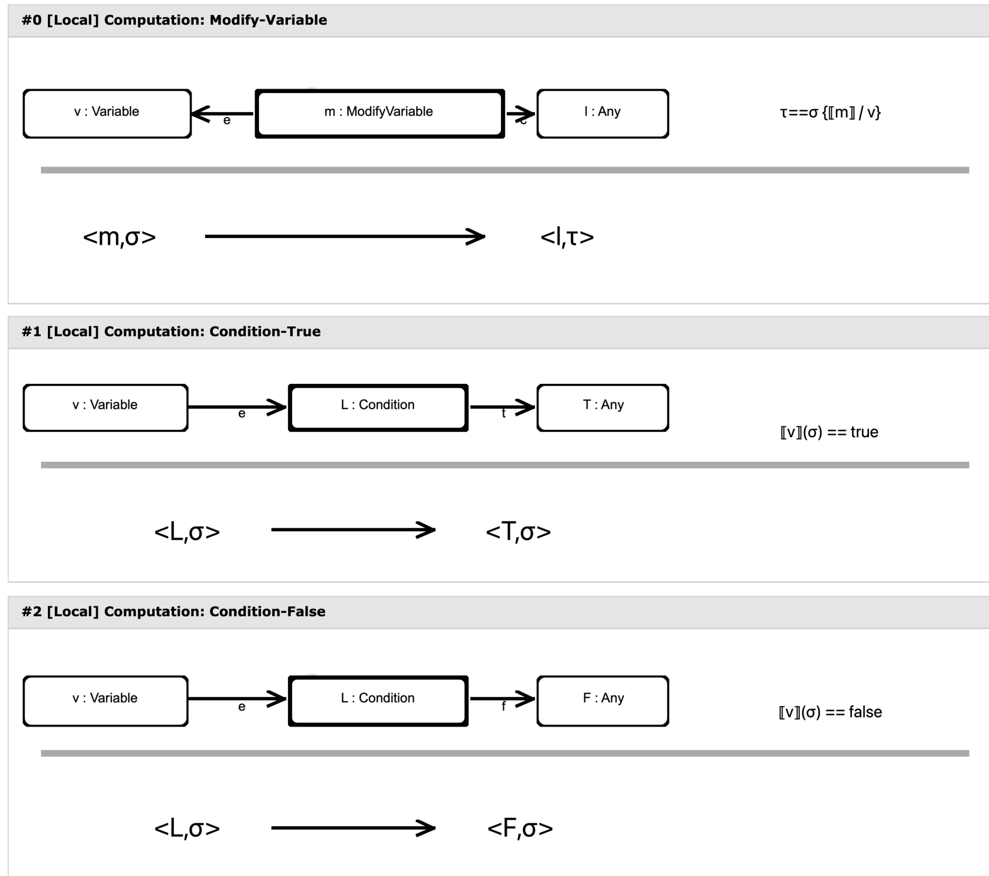

Figure 7: Computation rules for calculating the configuration transitions ($\rightarrow^*$) in WSL.

condition constrains the rules to one of the two possible valuations of v in the store σ. The `Condition-True` rule is only applicable when the variable is valued as *true*. The source pattern of this rule matches the `TrueTransition` edge and an arbitrary target node of that edge as T. Thus, in the conclusion, the end configuration is a configuration in which the store is unchanged, but the location progresses to the target of the `TrueTransition` branch. The `Condition-False` rule works analogous.

3.7 Transformation Process

In this section, a complete overview of the transformation process (utilizing both rule types) is given. For this purpose, we first describe the algorithm in a descriptive manner and then examine individual transformation steps using WSL to KTS as an concrete example.

3.7.1 General Approach

To transform a model of a language A to a model of language B, the following inputs are needed:

- A ruleset RS specified on the IL representations of A and B: $RS(A,B) : \text{IL}(A) \mapsto IL(B)$,
- a source model of type A transformed into the intermediate IL representation $\text{IL}(A)$,
- the initial store value σ_0

The algorithm then constructs the target model in $\text{IL}(B)$ with the following steps:

1. A working set WS is initialized with the starting configuration $\langle \varepsilon, \sigma_0 \rangle$. A set D holding the already finished ("done") configuration is initialized empty. We also initialize an empty target model of type $\text{IL}(B)$.
2. A configuration C is drawn randomly from WS. At the beginning we only have the starting configuration but more configurations will be discovered during the algorithm execution.
3. All applicable rules for C are applied. A rule is applicable if
   - the anchor of the rule is compatible with C,
   - the source pattern can be matched in $\text{IL}(A)$,
   - the premise holds,
   - the side condition holds and
   - the execution would add information to the *current* target model.

   With each rule execution, the target model is updated accordingly. If a new end configuration is discovered by applying a rule, it is added to WS as long as it is not already marked as "done" (included in D).
4. We add C to the set D of processed configurations, since all rules have now been applied for this configuration and also remove it from WS.
5. Steps 2-4 are repeated as long WS is not empty.
6. As every applicable rule has been applied for every discovered configuration the transformation process has finished and the target model is returned.

3.7.2 Exemplary transformation steps of WebStory to KTS

In this section, we demonstrate the exemplary steps of the transformation of a WSL model to a KTS model. The complete ruleset for this transformation has been shown in two parts, in Figure 6 (transformation rules) and Figure 7 (computation rules). The model under consideration is the model as shown in Figure 1.

First we need to transform the WSL model to the intermediate representation of type $\text{IL}(WSL)$. During the conversion the IL representation is enriched with additional letters for each screen [10].

[10] In our implementation, this is achieved with the term var $n =$ 'A'; $\forall S \in Screen.S.name = n{+}{+}$, but we do not prescribe any particular way to enrich IL for this calculus.

The resulting IL is shown in Figure 3. All information including the new screen letters are stored inside of String attributes of the model elements. In order to be able to name the individual elements we use the same annotations as in Figure 1 during the following example. The initial store valuation σ_0 shall be $[\![gold]\!] := false$ and $[\![key]\!] := false$, since this corresponds to how variables are initialized in WSL.

The evaluation starts by adding the initial configuration $\langle \varepsilon, \sigma_0 \rangle$ to the working set. As it is the only configuration present in the working set, it is the only candidate to be used as current configuration, and thus drawn from the set. Afterwards, both transformation rules are checked for applicability. Since the current configuration contains the *any* (ε) location, only global rules are applicable. Such a global rule is found in `Initial-Screen`, and the source pattern of the rule is matched in the source model. Since in case of WSL, the `StartMarker` is unique, there is only one unique match that is found, with s as the `StartMarker` and L as the `Screen` A. The target model is empty, therefore all elements in the target production are generated into the target model, resulting in a target model containing of a single KTS state corresponding to $\langle A, f, f \rangle$. Since all work for the initial configuration has been done, it is removed from the working set. The newly discovered configuration $\langle \mathrm{A}, \sigma_0 \rangle$ is added to the working set, resulting in $W = \{\langle A, \sigma_0 \rangle\}$ and $D = \{\langle \varepsilon, \sigma_0 \rangle\}$.

Now assume that a few steps later we have already produced the target model shown in Figure 8 (c). Here the configuration $\langle \mathrm{A}, \sigma_0 \rangle$ is already done and the nodes for $\langle B, f, f \rangle$ (top) and $\langle C, f, f \rangle$ (bottom) are already added to the target IL model. Assume we draw the configuration $\langle B, f, f \rangle$ from the working set and find a match for the `Screen-to-Screen` rule with click area 3 and screen A (cf. Figure 8 (a) & (b)). The premise is trivially fulfilled as the successor of the click area already is of type `Screen`. Applying the rule now adds everything from the target template that is not yet included in the target model. Since both the node for state $\langle B, f, f \rangle$ and the node for state $\langle A, f, f \rangle$ are already contained in the target model (Figure 8 (c)), only the edge between them (shown in red in Figure 8 (b)) is added in this case leading to the final target model of Figure 8 (d).

Another transformation step where the premise is not trivially fulfilled would be the match with screen A, click area 4 and the configuration $\langle B, f, t \rangle$, i.e. the keys have been found already (cf. Figure 9). For this match, the rule application needs to invoke the computation rules to determine if there exists a chain of computation rules that has a screen in their location. Such a chain can be found using `Condition-True`, which will match c_2 for its end configuration and then the `Condition-False` rule, which will match m_2, followed by the `Modify Variable` rule, which then will match the screen E in its end configuration. At this point, the check succeeds and corresponding a KTS state will be generated in the target model, while the configuration $\langle E, \{key = true, gold = true\} \rangle$ will be added to the working set.

4 Evaluation and Limitations

The calculus presented herein can successfully transform a WSL model to a KTS and does so without rewriting or in-place mutation. No element generated in the target model is ever mutated, the model is constructed by small steps until a final fixpoint is reached in which no other elements can be generated. The transformation is independent of the order of rule application, since

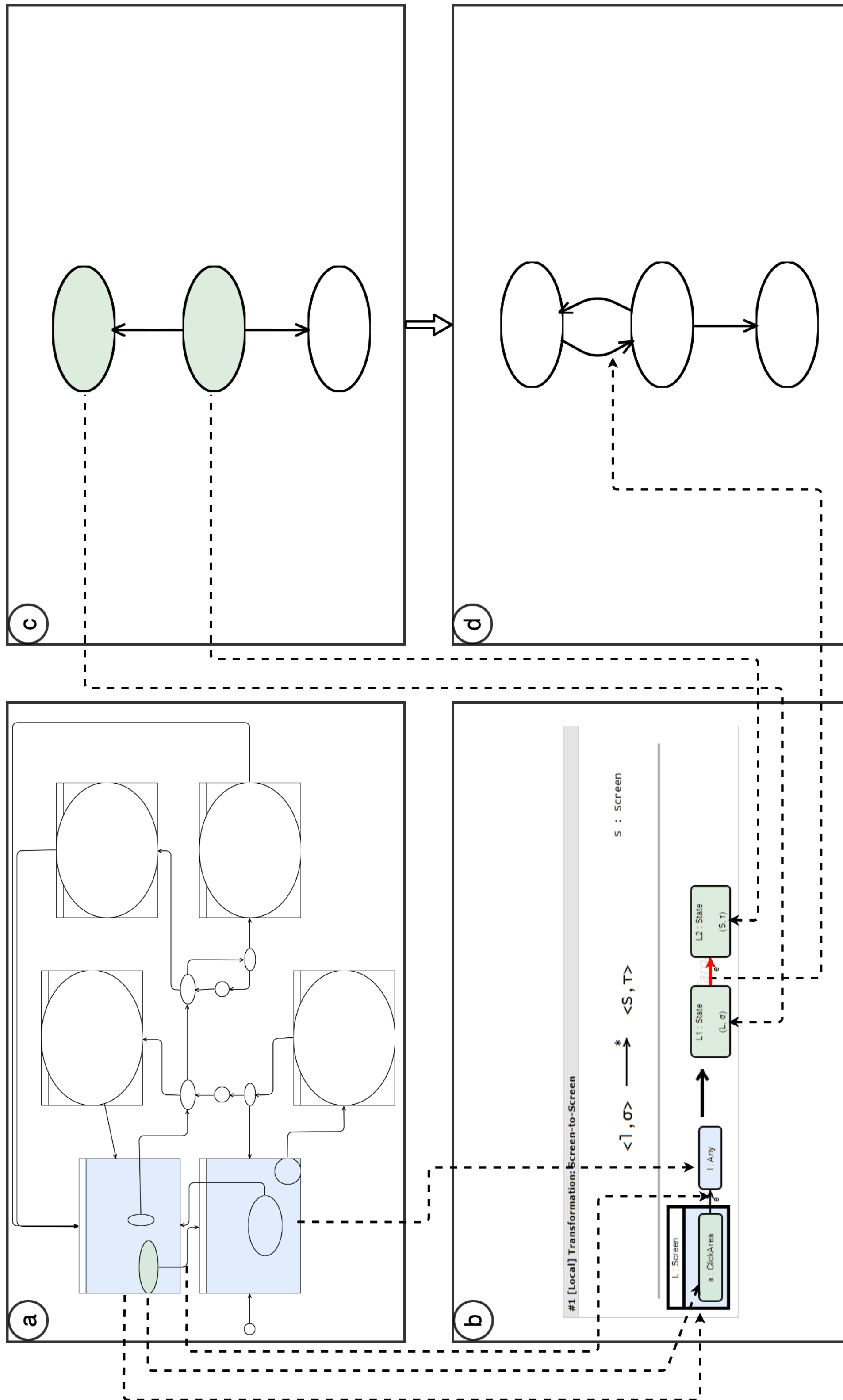

Figure 8: Application of the `Screen-to-Screen` rule (b) using the match in the source (WSL) as highlighted in (a), re-using the existing nodes in the target (KTS) as shown in (c) and producing the missing edge leading to the new target model in (d).

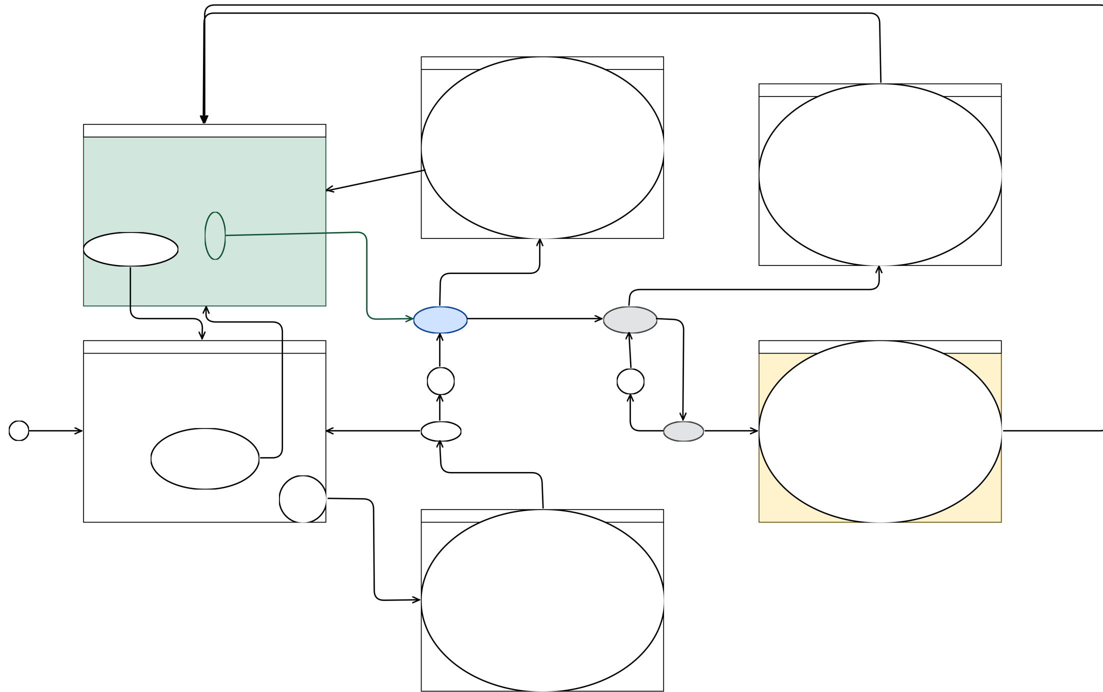

Figure 9: Source IL(WSL) model with highlights for the match of the `Screen-to-Screen` rule and the chain of computations that fulfill the premise $\langle l, \sigma \rangle \rightarrow^{*} \langle S, \tau \rangle \, \wedge \, S : \texttt{Screen}$.

elements are generated in an additive fashion, but never more then once. Thus, the application of individual rules is *commutative*. Hence, the fix point found by this transformation is unique.

The method presented herein works well for the use-case presented above, and we have already very promising results for forward-migration and model serialization. This method seems to be well suited for a wide-range of problems and produces rule sets that are reasonably easy to understand.

A limitation is the restriction of patterns to only positive matches, and not allowing negations in any kind or shape. However, conditions can include negations and rule out some elements individually, but negations cannot be applied to the pattern itself. It is thus not possible to match nodes *without* a certain parent or *without* a certain edge connection. This greatly simplifies the calculus, both in the mental model and implementation, but makes certain transformations more difficult to achieve. Preliminary investigations have shown, however, that this is not a severe restriction in practice, owing to the fact that the calculus promotes step-by-step construction of the target model in an incremental way due to being based on inference rules. This, together with the fact that the source model is never changed, eliminates most problems that occur due to the absence of negations.

5 Related Work

SOS-GT has notable differences to other works. Compared to GROOVE, it does not mutate models in-place, but instead constructs a new target model step-by-step. SOS-GT also offers a unique way to add auxiliary computations with its computation rules, behavior which needs more complex modeling in GROOVE. Similar considerations hold true for AGG, Triple-Graph-Grammars and related techniques.

JetBrains Meta Programming System (MPS)[11] is a language workbench that can be used to create DSLs and also features model transformation capabilities[12]. However, to our knowledge, MPS cannot match complex patterns and instead only matches a single model element (called *concept* in MPS lingo). MPS features a mechanism reminiscent of computation rules, where a context can be passed to evaluate a more complex condition. These conditions are specified in Java code and could possibly be used to implement the same behavior as checked with computation rules.

6 Conclusions and Perspectives

In this paper, we have introduced a rule scheme for model-to-model transformations based on inference rules, called SOS-GT. Characteristic for our approach is its conceptual simplicity which is the result of

- decomposing the rule system into two simple rule sub-systems, one for treating the graph transformation aspect in a pattern matching style and an SOS-inspired one for the computational aspect, and
- clearly separating source and target model in a way reminiscent of traditional code generation: The source model is only read, and the target model is monotonically built up. In particular this means that existing model structures are guaranteed to persist during the entire model-to-model transformation, a property important to prove the commutativity of rule application.

Currently, we are investigating how far this approach carries. Extending our example scenario to capture language migration is possible without problems in a similar fashion as sketched in [Kop19]. More challenging is our intended application to CI/CD: The generation of YAML code from CI/CD pipeline models specified in the graphical DSL Rig currently depends on a hand-written code generator [Teu21, TTS+21]. Specifying such a code generator with SOS-GT is non-trivial and requires extensions, not only to comprise textual languages as transformation target. It is our goal to solve this problem while maintaining as much of the current simplicity as possible.

In general, different application scenarios will require different extensions and therefore domain-specific versions of SOS-GT in order to maintain simplicity. We are convinced that this kind of domain-specific decomposition may drastically improve the performance of transformation-based approaches.

[11] https://www.jetbrains.com/mps/

[12] https://www.jetbrains.com/help/mps/generator-user-guide-demo4.html#reductionrule

Bibliography

[BMRS09] M. Bakera, T. Margaria, C. Renner, B. Steffen. Tool-supported enhancement of diagnosis in model-driven verification. *Innovations in Systems and Software Engineering* 5:211–228, 2009.
doi:10.1007/s11334-009-0091-6

[BNBK07] D. Balasubramanian, A. Narayanan, C. van Buskirk, G. Karsai. The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST* 1, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[ERRS10] H. Ehrig, A. Rensink, G. Rozenberg, A. Schürr (eds.). *Graph Transformations: 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - October 2, 2010. Proceedings*. Lecture Notes in Computer Science. Springer, 2010.
doi:10.1007/978-3-642-15928-2

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. 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.

[Fow05] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html, 06 2005. [Online; last accessed 2023-03-31].

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*. MDI '10, p. 22–31. Association for Computing Machinery, New York, NY, USA, 2010.
doi:10.1145/1866272.1866277

[KLNS21] D. Kopetzki, M. Lybecait, S. Naujokat, B. Steffen. Towards Language-to-Language Transformation. *Int. Journal on Software Tools for Technology Transfer*, Jun 2021.
doi:10.1007/s10009-021-00630-2

[Kop19] D. Kopetzki. *Generation of domain-specific language-to-language transformation languages*. PhD Thesis, TU Dortmund University, Dortmund, NRW, 2019.
doi:10.17877/DE290R-21179

[LKZ+18] M. Lybecait, D. Kopetzki, P. Zweihoff, A. Fuhge, S. Naujokat, B. Steffen. A Tutorial Introduction to Graphical Modeling and Metamodeling with Cinco. In *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation,*

Part I Modeling (ISoLA 2018). Lecture Notes in Computer Science 11244, pp. 519–538. Springer, 2018.
doi:10.1007/978-3-030-03418-4_31

[MLA10] J. McAffer, J.-M. Lemieux, C. Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition, 2010.

[MS09] T. Margaria, B. Steffen. Business Process Modelling in the jABC: The One-Thing-Approach. In Cardoso and Aalst (eds.), *Handbook of Research on Business Process Modeling*. IGI Global, 2009.

[MS10] T. Margaria, B. Steffen. Simplicity as a Driver for Agile Innovation. *Computer* 43(6):90–92, 2010.
doi:10.1109/MC.2010.177

[MSS99] M. Müller-Olm, D. Schmidt, B. Steffen. Model-Checking - A Tutorial Introduction. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*. Pp. 330–354. 1999.
doi:10.1007/3-540-48294-6_22

[Nau17] S. Naujokat. *Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools*. Dissertation, TU Dortmund University, Dortmund, Germany, Aug. 2017.
doi:10.17877/DE290R-18076

[NLKS17] S. Naujokat, M. Lybecait, D. Kopetzki, B. Steffen. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. *Software Tools for Technology Transfer* 20(3):327–354, 2017.
doi:10.1007/s10009-017-0453-6

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, 1981. DAIMI FN-19.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance*. Pp. 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, Boston, MA, USA, 2008.

[SGNM19] B. Steffen, F. Gossen, S. Naujokat, T. Margaria. Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages. In Steffen and Woeginger (eds.), *Computing and Software Science: State of the Art and Perspectives*. Lecture Notes in Computer Science 10000. Springer, 2019.
doi:10.1007/978-3-319-91908-9_17

[Teu21] S. Teumert. Visual Authoring of CI/CD Pipeline Configurations. Bachelor's Thesis, TU Dortmund University, 4 2021.
https://archive.org/details/visual-authoring-of-cicd-pipeline-configurations

[TTS+21] T. Tegeler, S. Teumert, J. Schürmann, A. Bainczyk, D. Busch, B. Steffen. An Introduction to Graphical Modeling of CI/CD Workflows with Rig. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation*. Pp. 3–17. Springer International Publishing, Cham, 2021.
doi:10.1007/978-3-030-89159-6_1