Chapter 9
GTSMorpher: Safely Composing Behavioural Analyses Using Structured Operational Semantics

Steffen Zschaler and Francisco Durán

Abstract We are seeing an increase in the number of different languages and design tools used for designing and implementing such systems, fuelled by research in domain-specific modelling languages leading to increasingly more reliable and production-ready environments for *language-oriented programming* (LOP). While LOP has undeniable benefits for the efficiency and effectiveness of software development, it creates new problems for software analysis: most existing analysis tools are tied to a specific representation of the software to be analysed. LOP is predicated on developing bespoke representations for each type of problem. This requires analysis tools to be, at least partially, reimplemented and adapted for each new such language.

One approach is to build transformations that compile a model in a given language into a representation that can be handled by a given analysis tool (cf. Chap. 5 of this book). A key challenge here is to ensure that these transformations correctly reflect the semantics of the original language in the analysis-tool-specific representation. This is non-trivial and becomes even more challenging when more than one analysis tool is to be applied to a given system design.

In this chapter, we present a different approach, where analyses are directly represented as *executable domain-specific modelling languages* (xDSMLs), making their operational semantics explicit as graph-transformation rules. Powerful composition operations provide support for the independent and reusable development of analysis tools and languages, which can then be woven at will. In previous work, we have developed the formal foundations for this approach and have shown the conditions under which such composition is safe, even when combining multiple different analyses. In this chapter, we introduce *GTSMorpher*, a software tool that allows xDSMLs and their compositions to be expressed in the context of the

S. Zschaler (✉)
King's College London, London, UK
e-mail: szschaler@acm.org

F. Durán
University of Málaga, Málaga, Spain
e-mail: duran@lcc.uma.es

R. Heinrich et al. (eds.), *Composing Model-Based Analysis Tools*,
https://doi.org/10.1007/978-3-030-81915-6_9

Eclipse Modelling Framework. We demonstrate the use of *GTSMorpher* through case studies.

This case-study chapter illustrates concepts introduced in Chap. 4 and addresses Challenge 1 in Chap. 3 of this book.

9.1 Introduction

Quality properties of software-intensive systems are increasingly important. At the same time, we are seeing an increase in the number of different languages and design tools used for designing and implementing such systems, fuelled by research in *domain-specific modelling languages* (DSMLs) leading to increasingly more reliable and production-ready environments for *language-oriented programming* (LOP) [War94]. LOP takes Naur's insight that all programming is theory building [Nau86] and follows it to its natural consequence, contending that software should be developed in problem-specific languages rather than general-purpose programming languages. While LOP has undeniable benefits for the efficiency and effectiveness of software development, it creates new problems for software analysis: most existing analysis tools are tied to a specific representation of the software to be analysed. LOP, on the other hand, is predicated on developing bespoke representations for each type of problem. This requires analysis tools to be, at least partially, reimplemented and adapted for each new such language.

One approach is to build transformations that compile a model in a given language into a representation that can be handled by a given analysis tool (cf. Chap. 5 of this book [Hei+21], [GM04]). A key challenge here is to ensure that these transformations correctly reflect the semantics of the original language in the analysis-tool-specific representation. This is non-trivial and becomes even more challenging when more than one analysis tool is to be applied to a given system design.

In this chapter, we present an alternative approach, predicated on the idea that modelling a language's semantics explicitly—producing an *executable domain-specific modelling language* (xDSML)—makes it possible to reason about these semantics when developing analysis tools. We introduce *GTSMorpher*, a tool, and DSML for specifying graph-transformation systems and their algebraic composition. We use graph transformations [Cor+97] to capture a language's operational semantics and then combine and reuse them with semantic guarantees. The approach indeed enables a modular approach to analysis (see Chaps. 4 and 5 of this book [Hei+21]), in which different analyses can be combined into one modelling language, so that different analyses can be enabled depending on what a project requires.

Graph-transformation systems (GTSs) were proposed in the late seventies as a formal technique for the rule-based specification of the dynamic behaviour of systems [Ehr79]. Recent uses of GTSs in the context of *model-driven engineering* (MDE) have proposed more practical uses of different forms of parametric GTSs for reusing model transformations, and reusing and composing DSML definitions. For example, in [LG13], de Lara and Guerra propose the use of transformation templates expressed over *metamodel concepts* that can then be instantiated. A metamodel concept defines structural requirements on a metamodel that allow a transformation to be executed. Metamodel semantics are not captured by metamodel concepts. In [DZT13, Dur+17], Durán et al. propose a more general form of parametrised GTSs where the parameter is not just a type graph, but a complete GTS, and where composition of GTSs is based on a GTS amalgamation construction. In the same way metamodel concepts gather the structural requirements, the set of rules of parameter GTSs are behavioural requirements over the concrete GTSs used in their instantiation. Thus, parametrised GTSs extend the metamodel concept notion to include the behavioural semantics of the metamodels.

GTS morphisms (see, e.g., [Eng+97, Ehr+06, GPS98b, EHC05]) are a key ingredient of GTSs and GTS compositions. The use of GTS morphisms enables useful syntactic and semantic guarantees. For example, morphisms are used in [LG13] so that transformations can be guaranteed to be syntactically reusable. In the case of [Dur+17], the use of suitable morphisms enables guarantees on behaviour protection of amalgamated GTSs. However, graph morphisms and GTS morphisms require a strong structural similarity between source and target graphs and GTSs, which hinders their applicability.

The need for powerful and flexible mechanisms for relating GTSs, to broaden opportunities for GTS reuse, has been attempted to solve in different ways. In the case of models, represented as graphs, this has been resolved more or less pragmatically by supporting a specific, fixed set of adaptations to be applied prior to applying the morphism (see, e.g., [LG13, Lar+07, DMC12, LG14]). To support complete GTSs, rules must also be related in a flexible manner. In [GPS98a, GPS98b], Große-Rhode et al. introduce temporal and spatial refinement relations, in which rules are refined into either sequences or amalgamations of rules. However, despite the introduction of derived attributes and links as in [DMC12] or [LG14], and the behavioural relations provided for GTS morphisms as in [Dur+17], we do not find a satisfactory solution until the proposal of *GTS families* in [ZD17].

Often, even where there is an intuitive match, no morphism can be established, due to structural mismatches. In [ZD17], Zschaler and Durán propose the use of *GTS transformers* to refactor GTSs with the goal of resolving these mismatches between source and target GTSs so that GTS morphisms can be defined. GTS transformers are basically functions and can successively be applied to our source GTS to find the one on which the morphism can be defined. This basic idea is systematised with the notion of GTS families. Given a set of transformers T, the T-family of a GTS GTS_0 is the set of GTSs reachable from GTS_0 using the transformers in T. The problem of defining a mapping morphism between a GTS GTS_0 and a target GTS GTS_1 then amounts to finding a GTS in the family of GTS_0 from which the morphism can be defined. This way, the problem becomes a model-based search problem [Joh+19]. In this chapter, however, instead of blindly searching for such matches, we use the capabilities of the *GTSMorpher* tool to specify the explicit transformation steps to be applied.

This approach offers a powerful reuse opportunity for model-based analysis tools when systems are developed using xDSML-based specifications. The possibilities for the modularisation of analyses as a parametrised GTS have been previously shown in, e.g., [Dur+17]. In [Mor+14], Moreno-Delgado et al. showed how the approach can be applied to reimplement the analysis provided by the Palladio simulator (see [Reu+16] and Chap. 11 of this book [Hei+21]). However, while the theory has been developed, for this approach to become practically viable, tool support is required. In this chapter, we introduce *GTSMorpher*, a tool, and DSML for specifying graph-transformation systems and their algebraic composition. We show how *GTSMorpher* can be used to specify weavings of simple graph-transformation systems as per [Dur+17] as well as of GTS families [ZD17]. A new case study in Sect. 9.4 shows a reimplementation of the Karlsruhe Architectural Maintainability Prediction (KAMP) approach [Ros+15] using the *GTSMorpher* tool. The tool ensures the correctness of weaving specifications and outputs GTSs in the Henshin format [Str+17] that can be executed or analysed further.

We have shown in previous work [Dur+17] that the same composition mechanism can also be used to combine multiple analyses on top of one xDSML. For example, [DZT13] shows an example of capturing performance analysis in this form. Generally, we encode analyses using the idea of history-determined variables [AL94]—variables whose current value can be inferred from the current and past values of other variables. In an MDE context, we encode these as observer objects in our models, using additional *Observer* metaclasses or metaassociations in the metamodel as suggested by Troya in [Tro+13]. This is combined with an operational semantics expressed using graph-transformation rules specifying how model state changes over time (which is similar to Abadi/Lamport's temporal logic of actions (TLA) [Lam94] approach). As in TLA, updates to the observers (history-determined variables) are simply included in the update parts of the graph-transformation rules giving the xDSML's operational semantics. This way, properties such as performance, reliability, efficiency, etc. of a modelled system can easily be read off at any point by inspecting the values captured by observer objects and links.

In the remainder, Sect. 9.2 gives a motivating example, which we will use in Sect. 9.3 to introduce the *GTSMorpher* tool. In Sect. 9.4 we walk through a more complex case study before concluding in Sect. 9.5.

9.2 Motivating Example

In this section, we present a simplified example of how a model-based analysis can be modelled using an xDSML in a way where this can be safely composed into different xDSMLs and, thus, easily reused.

Consider the example of a simple xDSML specifying production-line systems. Figure 9.1 gives an overview of the language's metamodel. It can be seen that in the language we can specify production-line systems by connecting various types

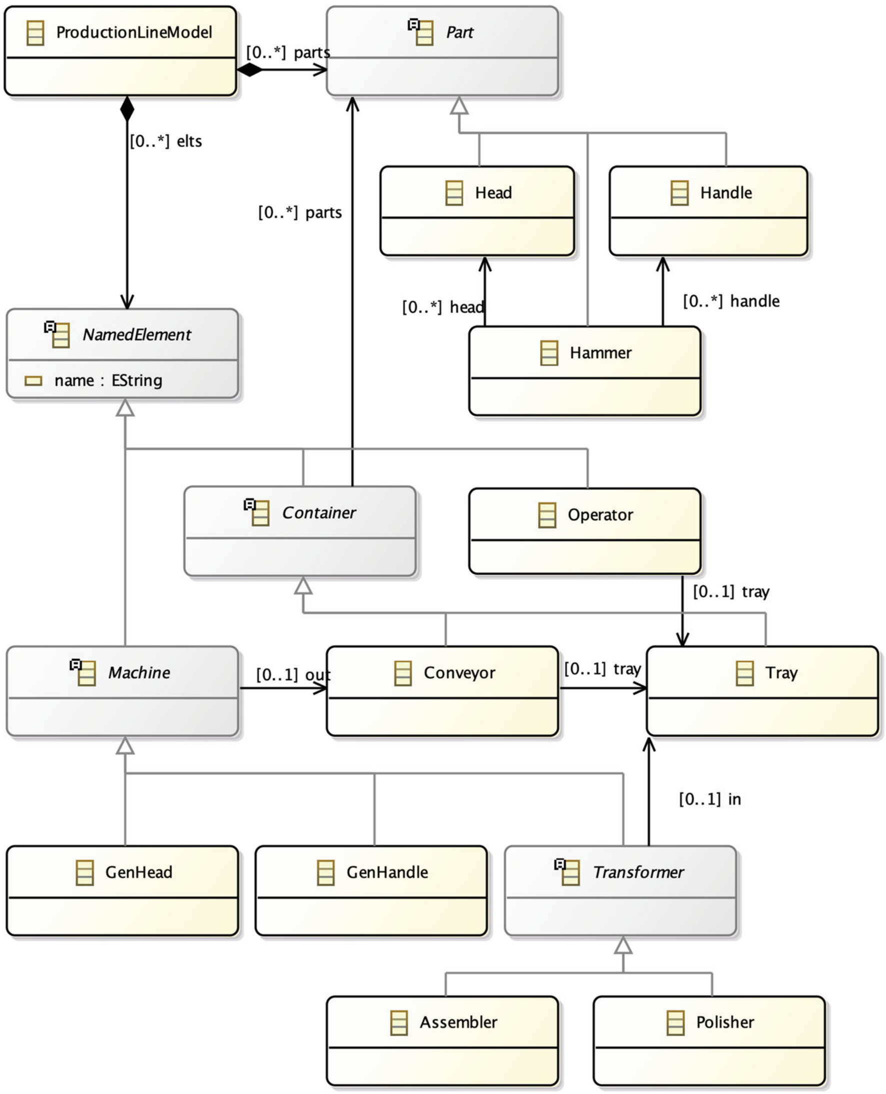


Fig. 9.1 Metamodel for the PLS xDSML

of machines via different kinds of containers. Different kinds of parts are produced and manipulated by the machines and transported via the containers. The operational semantics of this xDSML can be captured using several graph-transformation rules. Figure 9.2 shows such rules, specifying operational semantics of the *production-line system* (PLS) language. In particular, the behaviour of the Polisher machine is specified by the Polish rule (in the bottom right corner of Fig. 9.2). From these rules,

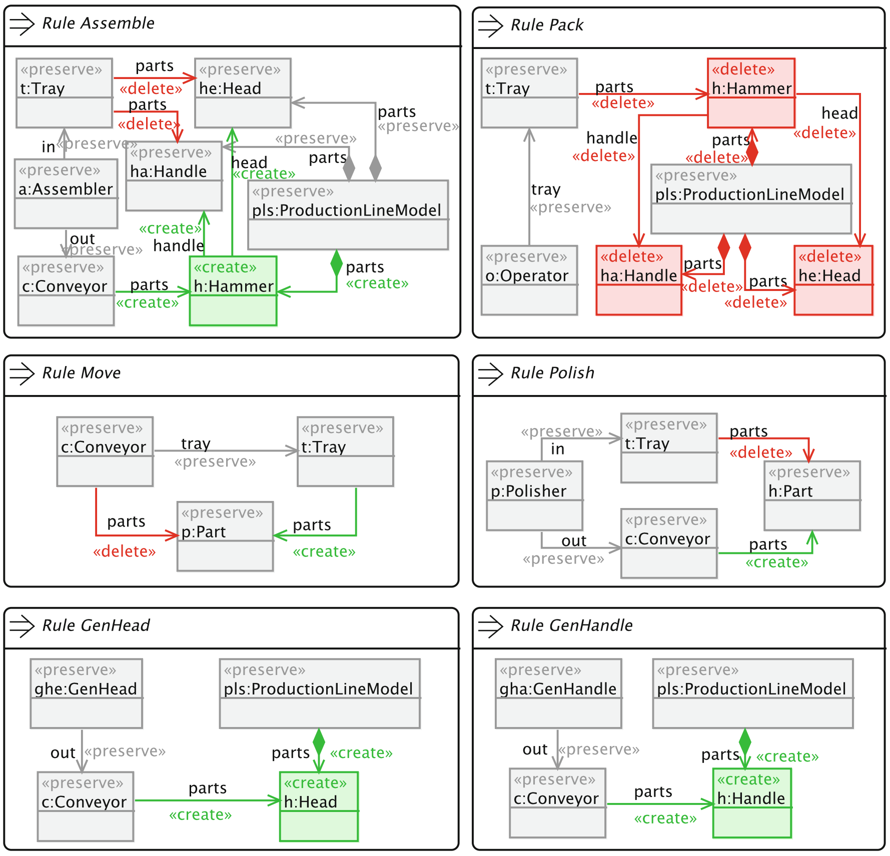


Fig. 9.2 PLS's rules expressed in Henshin [Str+17]. Henshin uses colour coding and textual labels to compactly present all parts of a graph-transformation rule. Elements represented in grey (and labelled preserve) are matched by the rule, but not changed. Elements in green (and labelled create) are added, while elements in red (and labelled delete) are removed

we can, for example, generate a simulation of a given production-line system for further analysis.

Let us now consider specifying an analysis of production-line systems. As a very simplistic example, we will specify an analysis that allows to keep track of parts manipulated by a specific machine. This can, for example, be used to track reliability or performance of any given machine. Rather than changing the PLS xDSML to introduce the relevant observer objects and associations directly, we want to specify our analysis in a reusable format that can be woven into the PLS xDSML, but also into other xDSMLs. Figure 9.3 shows how we might capture this in a metamodel. Note the green association (made) indicating the new observer association we need to add to the metamodel to capture elements manipulated by a given server. In the following, we will consider everything in the metamodel not coloured in green

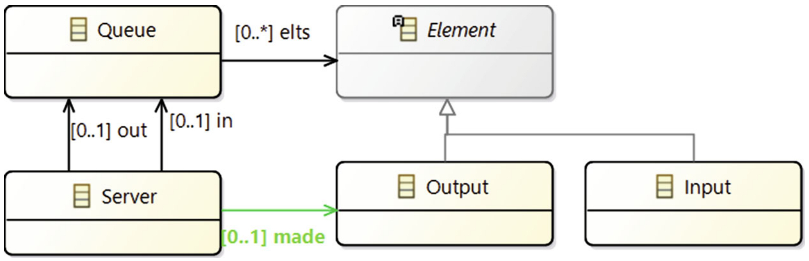


Fig. 9.3 Analysis metamodel

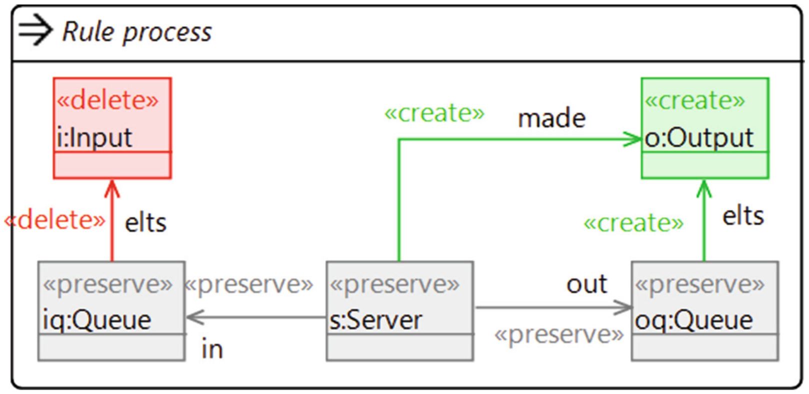


Fig. 9.4 Rule specifying the analysis based on the abstract server metamodel

the interface of our analysis xDSML (and, technically, will annotate it with the @Interface annotation). To compose our analysis into the PLS xDSML, we will need to establish a mapping instantiating every interface concept with a concept in the PLS metamodel. Figure 9.4 shows a rule specifying the semantics of our analysis: whenever a Server produces a part on its out Queue, it will record this fact by establishing a made link.

In Sect. 9.3, we introduce our *GTSMorpher* tool and show how it can be used to specify these xDSMLs and their composition so that the analysis is included in the result. After that, we will walk through a more complex analysis composition and reuse case study.

9.3 The *GTSMorpher* Tool

In this section, we give a brief walk-through of *GTSMorpher* using the example from Sect. 9.2, before we apply *GTSMorpher* to a new example of analysis composition in the next section.

GTSMorpher supports the formal specification and analysis of GTSs and morphisms between them (*GTS morphisms*) as well as the automated composition of GTSs based on GTS morphisms. This enables the existing theory on GTS morphisms and GTS amalgamation to be applied to real-world GTSs, which would otherwise be impractical as the size and complexity of even simple GTS specifications quickly make it difficult for a human to validate correctness or compute amalgamations manually. In addition to this, *GTSMorpher* provides a number of features to simplify the specification of GTS morphisms and amalgamations: Code-completion support makes it easier to correctly reference various constituent parts of a GTS, while morphism auto-completion, interface morphisms, and GTS-family support [ZD17] allow very compact specifications of complex morphisms and amalgamations. Where analyses are specified as GTSs (or xDSMLs), *GTSMorpher* supports the automated weaving of analyses into arbitrary xDSMLs, enabling analysis reuse across DSMLs.

To support the specification of GTSs and GTS morphisms, *GTSMorpher* provides a textual DSML for specifying algebraic manipulations of GTSs. A GTS is encoded as a type graph (an Ecore metamodel [Ste+09]) and, optionally, a module of Henshin graph-transformation rules [Str+17]. *GTSMorpher* supports the specification of plain GTSs as well as GTS families [ZD17], as well as the expression of GTS morphisms and GTS amalgamations [Dur+17], which can be reused as inputs for further morphism and amalgamation definitions. GTSs produced from any *GTSMorpher* specification can be exported as Ecore metamodels and Henshin modules for use in further analysis and execution. *GTSMorpher* has been developed in the Xtext language workbench and can be obtained from its Github repository.[1]

The foundation of safe composition of GTSs lies in the notion of GTS morphisms—mappings between the elements of two GTSs that ensure the structure of the GTSs is preserved. We, therefore, start by showing how a GTS morphism is expressed in the *GTSMorpher* DSML. On top of GTS morphisms, we can weave GTSs by computing the pushout of a suitable span of GTSs and GTS morphisms. In this section, we show how this can be expressed and controlled in *GTSMorpher*.

9.3.1 Specifying GTS Morphisms

GTSs and GTS morphisms are expressed in .gts files. These are text files using the syntax below (syntax completion is available throughout the Eclipse editor).

[1] *GTSMorpher* is available at https://github.com/gts-morpher/gts_morpher.

Basic GTS Syntax

The easiest way to specify a GTS is through a GTS literal as below:

```
gts PLS {
  metamodel: "pls"
  behaviour: "plsRules"
}
```

Here, PLS can be an arbitrary, optional name for the GTS that may later be used to reference the GTS. The **metamodel** clause references an Ecore package defining the metamodel of the GTS. The **behaviour** clause references a Henshin module the rules of which are considered to be the rules of the GTS. It is acceptable to leave out the behaviour clause. Some alternative forms of specifying GTSs exist; these all differ primarily by what is specified between the curly braces: we will discuss GTS families and GTS amalgamation later.

Any GTS specification may be annotated with two modifiers:

1. **export**: This annotation indicates that the .ecore (and optionally the .henshin) file of the GTS should be generated into the src-gen/ folder of the containing Eclipse project.
2. **interface_of**: These GTSs are formed from the original metamodel and rules by only considering a sub-GTS typable over the metamodel elements explicitly annotated with @Interface. This is particularly useful for GTS amalgamation as described below.

Finally, a GTS specification can reference another named GTS. This is particularly useful when referencing a pre-defined GTS from a mapping specification.

Basic Morphism Syntax

A GTS morphism is specified as a mapping between two GTSs, using a **map** clause as shown in Listing 9.1.

Here, **from** and **to** each specify a GTS. The block in curly braces after **from** and **to** is actually a GTS specification (see above) with the **gts** keyword left out.

The mandatory **type_mapping** section describes the type-graph morphism part of the GTS morphism by providing a clan morphism between the two metamodels [Lar+07]. This is achieved through a list of mapping statements that map a class, reference, or attribute.[2]

Similarly, the optional **behaviour_mapping** section describes rule mappings. If the GTSs do not have rules, the **behaviour_mapping** clause should also be left out and the file only specifies a clan morphism between the metamodels. Each rule mapping is started using the keyword rule followed by the name of the rule in the source GTS,

[2] The careful reader will have noticed the metaclasses InputQueue and OutputQueue being referenced here. These will be explained later, when we introduce the definition of GTS families.

```
map {
  from interface_of {
    Server
  }

  to {
    metamodel: "pls"
    behaviour: "plsRules"
  }

  type_mapping {
    class server.Server => pls.Polisher
    class server.InputQueue => pls.Tray
    class server.OutputQueue => pls.Conveyor
    // reference YYY => XXX
    // attribute YYY => XXX
    // ...
  }

  behaviour_mapping {
    rule process to polish {
      object iq => t
      object o => pt2
      object s => p
      object oq => c
      link [s->iq:in] => [p->t:in]
      link [oq->o:elts] => [c->pt2:parts]
      link [iq->i:elts] => [t->pt:parts]
      link [s->oq:out] => [p->c:out]
      object i => pt
    }
  }
}
```

Listing 9.1 Syntax for specifying GTS morphisms

the keyword **to**, and the name of the rule in the target GTS. Each rule mapping again contains a list of mappings for objects, links, and slots (attribute constraints) in the rule as well as for rule parameters.

Extensive validation is performed for any mapping specification, including to check whether it represents a (potential) GTS morphism. Eclipse error and warning markers provide information and hint about the results of these checks. Slot mappings are considered valid if the associated expressions are syntactically identical, subject to parameter renaming.

Morphism Auto-Completion and Unique Auto-Completion

The system will create error markers if type or behaviour mappings are not complete. As it can be quite tedious to type out all parts of the mapping, it is possible to ask the system to automatically complete a partial mapping by adding the keyword **auto−complete** at the start of the specification:

```
auto−complete map { ... }
```

As long as the mappings specified do not break the conditions for a GTS morphism, the system will attempt to complete the morphism automatically. The user can request for the completed morphisms to be exported as .gts files for inspection. Auto-completion uses a backtracking algorithm tentatively adding mappings and checking if morphism properties are still maintained. Mappings are not selected randomly: the structure of the metamodel and rules and existing mappings are taken into account to identify mappings that are likely to maintain morphism properties.

Users can claim that only a unique auto-completion to a morphism exists by adding the **unique** keyword:

```
auto−complete unique map { ... }
```

Checking whether a unique auto-completion exists is expensive as it may require searching the complete space of possible mappings (as opposed to checking if a completion is possible, where we can stop once one completion has been found). To avoid interfering with the editing experience, *GTSMorpher* will initially only add a warning marker to the **unique** keyword to show that this claim has not been checked yet. To check unique completability, users must explicitly request a validation. If auto-completion is not unique, an error marker will be added to the file. This provides quick-fix suggestions for mappings to add to sufficiently constrain the possible auto-completions. Suggestions are provided in order of potential impact; the top suggestion should offer the quickest path to unique auto-completion.

Mapping with Virtual Rules

When a rule in the source GTS cannot be mapped to any rule in the target GTS, it can be mapped to a virtual rule, automatically generated by *GTSMorpher*. This is useful, for example, where we want to produce amalgamations that introduce new rules into an existing GTS. In such a case, there is no rule that can be mapped to, but the amalgamation still requires a complete morphism. Mapping a rule to a virtual rule is indicated using a rule mapping of the following form (we will call such mappings "to-virtual mappings"):

```
rule init to virtual
```

Note that **virtual** is a language keyword, rules named "virtual" are not supported. From such a rule mapping, *GTSMorpher* will generate a virtual rule with the same structure as the source rule and use that in the mapping. Note that to-virtual rule mappings cannot specify any element mappings; these are all implicit, because the rule is dynamically generated only when needed. At the same time, there is only one valid mapping between source rule and virtual rule, so there is no need to specify any explicit element mappings.

Mapping to arbitrary virtual rules may affect behaviour-preservation properties of the morphism [Dur+17]. To ensure behaviour in the target GTS is preserved, it is possible to constrain virtual rules to be identity rules; that is their left- and right-hand sides must be identical. Adding an identity rule to a GTS does not change the behaviours modelled apart from adding stuttering steps. Only identity rules can be mapped to virtual identity rules, of course, and the tool will check this. To specify a rule mapping to a virtual identity rule (a "to-identity rule mapping") the following form of rule mappings should be used (where init is the name of a rule in the source GTS):

```
rule init to virtual identity
```

Note that the word **identity** is a keyword in the morphism language. It is therefore not possible to map rules named "identity".

Where possible, auto-completion will consider completing by introducing to-virtual or even to-identity rule mappings. This behaviour can be restricted by claiming auto-completion is possible using only to-identity rule mappings or without using to-virtual mappings at all, to ensure behaviour preservation:

- **auto−complete to−identity−only map** { ... } claims that only to-identity mappings might need to be introduced.
- **auto−complete without−to−virtual map** { ... } claims that no to-virtual mappings will need to be introduced to complete the morphism.

Conversely, rule mappings can be established from virtual empty source rules. This is useful where the target GTS contains rules that cannot be matched by any of the source rules—for example where the target GTS contains more rules than the source GTS, as is the case when reusing the specifications of non-functional properties as described in [DZT13]. There is no need to consider identity source rules or any other more complex source rules: For empty source rules rule morphisms trivially exist.

A rule mapping from an empty source rule (a "from-empty rule mapping") is defined as follows:

```
rule empty to do
```

where do is the name of a rule in the target GTS. **empty** is a keyword in the language and cannot be the name of a rule.

Auto-completion can consider introducing from-empty rule mappings automatically. Note that this is very likely to reduce the chances of producing *unique* auto-completions as from-empty mappings can be trivially introduced and can be trivially complemented with to-virtual mappings to ensure all rules in both GTSs have a mapping. In order to produce more intuitive behaviour, *GTSMorpher* will

```
gts_family ServerFamily {
  {
    metamodel: "server"
    behaviour: "serverRules"
  }

  transformers: "transformerRules"
}
```

Listing 9.2 Syntax for specifying GTS families

(1) not try to introduce from-empty mappings if a mapping with an actual source rule can be found, and (2) only try to introduce from-empty rule mappings if explicitly instructed to do so. The following syntax allows from-empty mappings to be included:

```
auto-complete allow-from-empty map { ... }
```

9.3.2 GTS Families

You can specify that the source or target of a GTS morphism should be taken from a GTS family by providing the definition of the family and the sequence of transformers to apply to the family's root GTS when picking the GTS you actually want. GTS families are described in more detail in [ZD17]. Intuitively, the T-GTS family of a GTS GTS_0 is the set of GTSs reachable from GTS_0 using the transformers in T.

To specify a GTS family, replace the GTS specification with one that follows the format shown in Listing 9.2. In it, **metamodel** and **behaviour** describe the root GTS of the family as usual. Although the transformers introduced in [ZD17] can be specified in different ways, here we restrict ourselves to transformers specified using Henshin rules. **transformers** references a Henshin module (typed over Ecore and Henshin) with the transformer rules of the GTS family. GTS family specifications can be used anywhere a GTS is expected.

We can then specify a specific GTS in this family by specifying the sequence of transformers to be applied on the root GTS of the family. Listing 9.3 shows this in an example. AdaptedServer picks a specific variant of our Server GTS that can be mapped cleanly onto the PLS xDSML. The **using** clause indicates the sequence of transformer applications, including their actual parameters, to be used in deriving the correct GTS from inside the family. Specifically, it introduces the separate InputQueue and OutputQueue subclasses of Queue that are needed for mapping to Tray and Conveyor. We do not show the transformers used in this example here. These

```
gts AdaptedServer {
  family: ServerFamily

  using [
    addSubClass (server.Queue, "InputQueue"),
    addSubClass (server.Queue, "OutputQueue"),
    reTypeToSubClass (serverRules.process, server.Queue,
                      server.InputQueue, "iq"),
    reTypeToSubClass (serverRules.process, server.Queue,
                      server.OutputQueue, "oq"),
    mvAssocDown (server.Server.in, server.InputQueue),
    mvAssocDown (server.Server.out, server.OutputQueue)
  ]
}
```

Listing 9.3 AdaptedServer is a GTS in the ServerFamily GTS family

```
gts ServerPLS {
  weave(dontLabelNonKernelElements, preferMap2TargetNames): {
    map1: interface_of(AdaptedServer)
    map2: Server2PLS
  }
}
```

Listing 9.4 Syntax for GTS amalgamation

can be found on the *GTSMorpher* repository. Examples of some other transformers will be shown later.

9.3.3 GTS Amalgamation

Once a valid morphism has been described (either as a complete map or by using unique auto-completion), GTS amalgamation can be performed (as per [Dur+17]). Where the source GTS is declared using **interface_of**, amalgamation will assume an inclusion to be defined by the @Interface annotations.

GTS amalgamation is specified in a special form of GTS specification shown in Listing 9.4. **map1** and **map2** are expected to, together, define a *span*; that is both mappings must have the same source GTS. No further checks of the morphisms are undertaken, and no guarantees are given w.r.t. semantics preservation of the amalgamation step (although we are working on supporting this in future versions of *GTSMorpher*). Both **map1** and **map2** can be defined either by referencing an existing named mapping or by using the **interface_of** keyword.

The **weave** clause can be extended with parameters specifying the rules to use when generating names for the amalgamated model elements. By default, weaving will preserve the names of all model elements that contributed to a given woven element. If these names are all identical, the new model element will have the same name. Otherwise, all names will be joined together using underscores as the separator. Names of model elements that are not mapped from the kernel GTS will be prefixed with left__ (for **map1**) or right__ (for **map2**), respectively, to indicate their provenance. Through parameters, **weave** can be instructed to give preference to names defined in one of the GTSs involved. If any naming option leads to names that are not unique within their scope, the weaver will fall back to the default naming strategy for these elements. The choices we have made in the example above will result in the woven xDSML to use the PLS names wherever possible.

9.4 An Application Example

This section shows how the mechanisms introduced in the previous sections may be useful in the development of generic tools with minimal effort. Specifically, we illustrate how to exploit the capabilities of GTSMorpher by developing an alternative implementation of the *Karlsruhe Architectural Maintainability Prediction* (KAMP) approach [Ros+15]. KAMP evaluates the maintainability of IT systems based on the metamodel of their architectures. More precisely, assuming a component-based architecture, and given an initial request for change, it predicts the change propagation in the software architecture model. In the KAMP approach, components are considered black boxes. Although no knowledge about component internals is required, the model of the software architecture is supposed to include information on both technical and organisational tasks—including source code files, test cases, build configurations, etc.—and contain explicit interface specifications that bind them in the software process. This information and change propagation rules are then used by KAMP to calculate the change propagation in the software architecture automatically. As a result of the process, KAMP gives a list with all the structural and organisational tasks to execute the change request.

A complete implementation of the approach requires a detailed distinction of elements and tasks, so that specific and friendly information is provided to the final user. However, the core of the tool is quite simple; it just "taints" those elements affected by a given change. By using the propagation rules, this "tainting" of elements leads to the identification of all elements affected by an initial change request. However, given a DSML description, possibly including both a metamodel and transformation rules describing its behaviour, the application of the approach would require the modification of the model on which the propagation is to be performed. This is, for example, the approach followed to implement the technique on the Palladio system (cf. [Ros+15]).

To define the KAMP approach generically, so that we can apply it to any DSML description, we just need the possibility of tainting elements and propagating such tainting. In other words, the KAMP approach is defined just by the DSML defined by the metamodel in Fig. 9.5 and the propagation rule in Fig. 9.6. To be able to

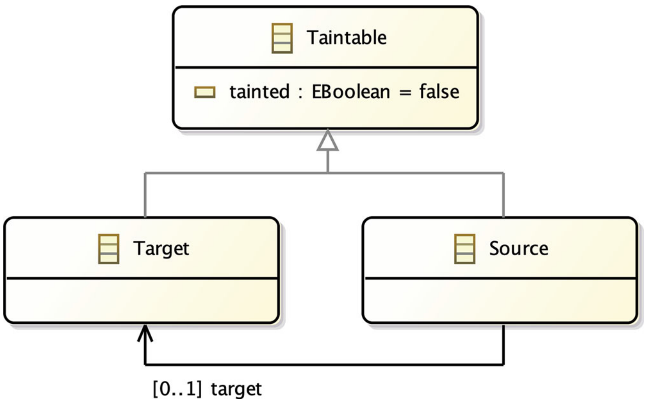


Fig. 9.5 KAMP's metamodel (kamp)

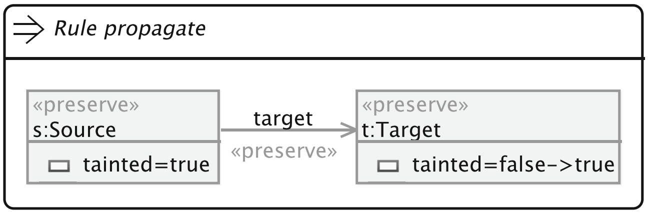


Fig. 9.6 KAMP's rules (kampRules)

propagate the tainting on any specific system, we just need to be able to instantiate the KAMP DSML on the specific places on which change is propagated in the system. The good news is that we only need to indicate the specific propagation points, since the propagation will happen always in the same way. Even more, we can assure that the modified system thus obtained behaves in exactly the same as the original system.

9.4.1 Making PLS Taintable

Instead of using a complex system, we show in the rest of the section how to apply the KAMP approach to the PLS language introduced in Sect. 9.2. To do it, we need to first extend the PLS language so that elements may be tainted, and then introduce the propagation rules on any specific propagation point. Notice that if the attribute was introduced together with the propagation rules, we would get a different attribute on each instantiation. Instead, we first introduce the attribute, a boolean attribute tainted, and then each propagation rule operating on such same attribute. With the machinery provided by GTSMorpher this is very simple. We just

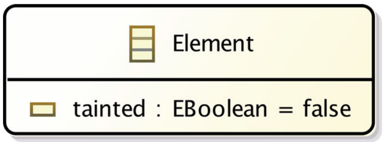


Fig. 9.7 Taintable's metamodel (taintable)

```
auto-complete unique allow-from-empty map ITaintable2PLS {
  from interface_of{Taintable}
  to PLS
  type_mapping {
    class taintable.Element => pls.NamedElement
  }
}
```

Listing 9.5 GTS morphism to enable PLS to become taintable

need a GTS with the metamodel depicted in Fig. 9.7 and no rules. In this metamodel, the only element not annotated with @Interface is precisely the tainted attribute.

```
gts Taintable {
  metamodel: "taintable"
}
```

To be able to taint any element of the PLS language we just need to instantiate the generic Taintable GTS with the PLS, and specifically by mapping the Element class to the NamedElement class, thus giving the tainted attribute to all named elements of the PLS. Given the PLS GTS defined as

```
gts PLS {
  metamodel: "pls"
  behaviour: "plsRules"
}
```

we can instantiate the Taintable generic GTS just by providing the GTS morphism ITaintable2PLS from the GTS interface_of {Taintable} to the PLS GTS that maps the class Element to the class NamedElement as shown in Listing 9.5. Notice the use of the allow-from-empty directive. GTS morphisms require injective and surjective mappings between the two rule sets. That is, for each rule in the target GTS—the PLS in this case—we need a rule in the source GTS. Since there are no rules in the interface of the Taintable GTS, empty rules are used instead. To simplify the exhaustive definition of these mappings, the combined use of the allow-from-empty and auto-complete directives automatically generates all these required mappings.

Given the ITaintable2PLS morphism and the inclusion of the interface of Taintable into itself, the amalgamation GTS TaintablePLS is constructed as shown in Listing 9.6.

```
export gts TaintablePLS {
  weave(dontLabelNonKernelElements ,preferMap2TargetNames ): {
    map1: interface_of(Taintable)
    map2: ITaintable2PLS
  }
}
```

Listing 9.6 Constructing TaintablePLS

The TaintablePLS GTS is as the PLS GTS but with an additional attribute tainted in the NamedElement class, which is inherited by all its subclasses, which can now be "tainted".

9.4.2 Adding Taint Propagation

The following step is to instantiate the KAMP GTS with the PLS using different mapping morphisms specifying the different links on which we want to propagate the tainting. Notice that now the tainted attribute is part of the interface, and therefore, it will be mapped into the homonymous attribute in the TaintedPLS GTS. In what follows we are going to carry on several instantiations to illustrate different cases.

The KAMP GTS Family

Assume we are interested in specifying change propagation due to the parts being generated. If a machine changes, the tray on which the parts generated by it are placed requires change. The transformers taking parts from such trays, as well as operators, will also need to adjust to change. This change needs to be propagated along the structure of specific instance models, since the change required by a machine implies the change on a subsequent tray, which changes transformers and operators taking parts from them. In turn, change in these transformers, which are themselves machines, will require change in subsequent trays, transformers, and operators. Notice however that the conveyors between machines and trays do not require change, since they are just moving bands to transport objects. Given the nature of the KAMP approach, tainting may be propagated as required using the relations between these elements. First, we need to define the KAMP GTS:

```
gts kampGTS {
  metamodel: "kamp"
  behaviour: "kampRules"
}
```

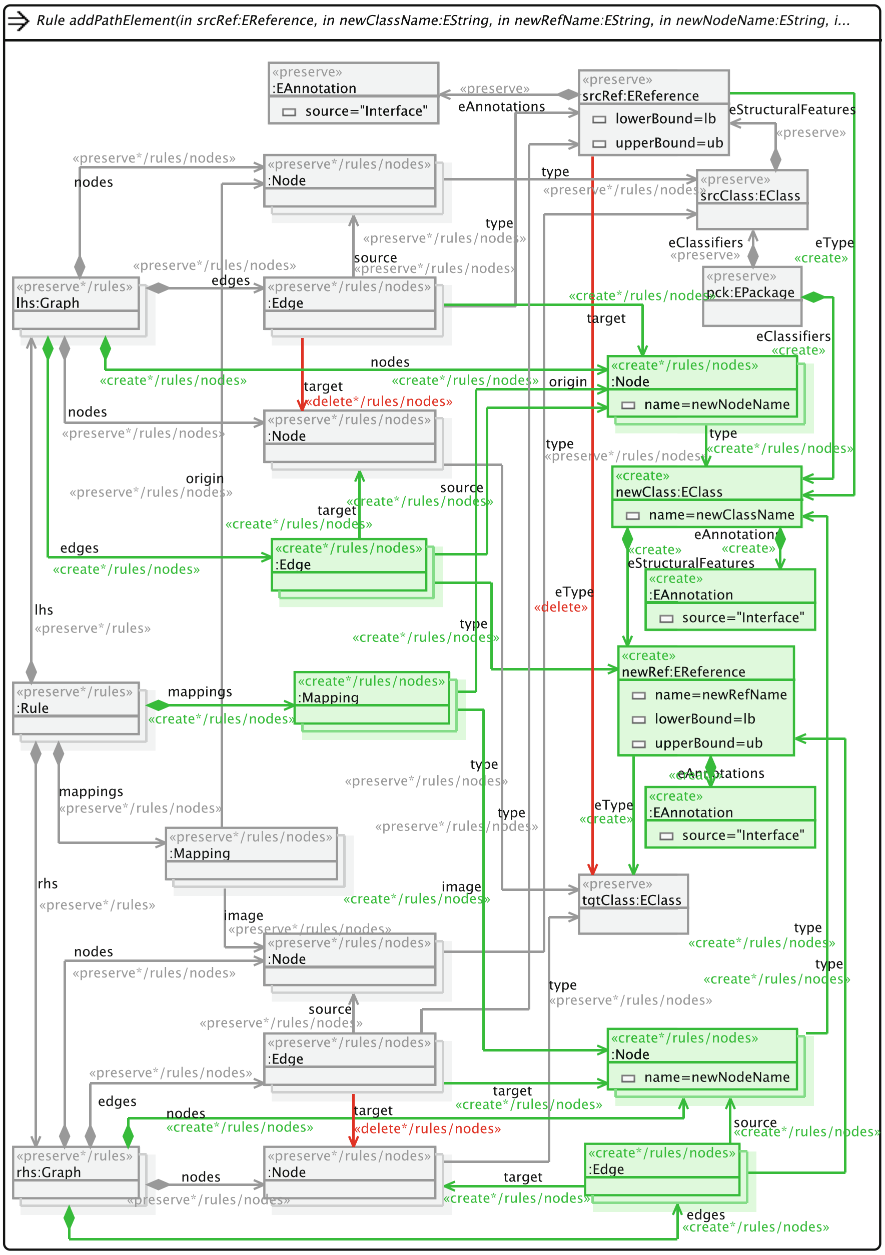
However, these relations are not always direct, nor mimic the pattern provided by the KAMP rules. In other words, no morphism can directly be defined for any of these links, and therefore all these instantiations require the introduction of GTS transformers through appropriate GTS families.

Consider for example the in association of the Machine class. In the KAMP's propagation rule, the tainting goes from Source to Target, whilst we are interested in the opposite direction for the in association of the Transformer class, since we want it to propagate from the tray objects to the subsequent transformers taking pieces from them. The same situation is found for the tray association of the Operator class. Moreover, the multiplicity of this association is 0..*, whilst the target association of the Source class in KAMP's metamodel (see Fig. 9.5) has multiplicity 0..1. Finally, the relation between a machine and its subsequent tray is not direct, since it happens through an intermediate conveyor. Of course, we could define a more general metamodel with alternative cases and corresponding alternative rules, but we do not need to. This is precisely the reason for transformers and families, to be able to specify the nature of an abstraction as the one provided by KAMP, manage the variability of situations through transformers, and then adjust the source GTS so that the instantiation may take place.

Figures 9.8, 9.9, and 9.10 define several transformer rules. The addPathElement transformer allows us to introduce an intermediary class between the source and target classes; the reverseReference transformer allows us to reverse a link; and adjustMultiplicity allows us to change the multiplicity of a link. Although some familiarity with Henshin's metamodel and with its way of specifying transformation rules is required to understand them, these rules just define changes on the metamodels and rules of the GTSs on which they are applied. For example, the adjustMultiplicity rule just specifies a change in the multiplicities of the reference specified as parameter. The most complex one of these three transformers is the addPathElement one. Given an EReference instance srcRef, between a source class srcClass and a target class tgtClass, it introduces a new class newClass as target of srcRef, and a new reference newRef from this newClass to tgtClass. Correspondingly, all those rules in which the reference appears are modified introducing new intermediate nodes of class newClass appropriately linked.

The KAMP family is then defined as the family of GTSs reachable from the KAMP GTS using the transformer rules. In general, one would want to provide a set of general transformers and expect the *GTSMorpher* tool to search for the right version of the source GTS so that the instantiation may take place. Instead, here, we explicitly control the application of transformers as pointed out above.

```
gts_family KAMPFamily {
  kampGTS
  transformers: "transformerRules"
}
```



Rule addPathElement(in srcRef:EReference, in newClassName:EString, in newRefName:EString, in newNodeName:EString, i...
«preserve» :EAnnotation source="Interface"
«preserve» srcRef:EReference lowerBound=lb upperBound=ub
«preserve» srcClass:EClass
«preserve» pck:EPackage
«create» newClass:EClass name=newClassName
«create» :EAnnotation source="Interface"
«create» newRef:EReference name=newRefName lowerBound=lb upperBound=ub
«preserve» tgtClass:EClass
«preserve*/rules» lhs:Graph
«preserve*/rules» :Rule
«preserve*/rules» rhs:Graph
«create*/rules/nodes» :Node name=newNodeName
«create*/rules/nodes» :Edge
«create*/rules/nodes» :Mapping
«preserve*/rules/nodes» :Node
«preserve*/rules/nodes» :Edge
«preserve*/rules/nodes» :Mapping
«delete*/rules/nodes»
eAnnotations
eStructuralFeatures
eClassifiers
eType
«delete»
nodes
edges
mappings
source
target
type
origin
image
lhs
rhs

Fig. 9.8 addPathElement transformer (transformerRules)

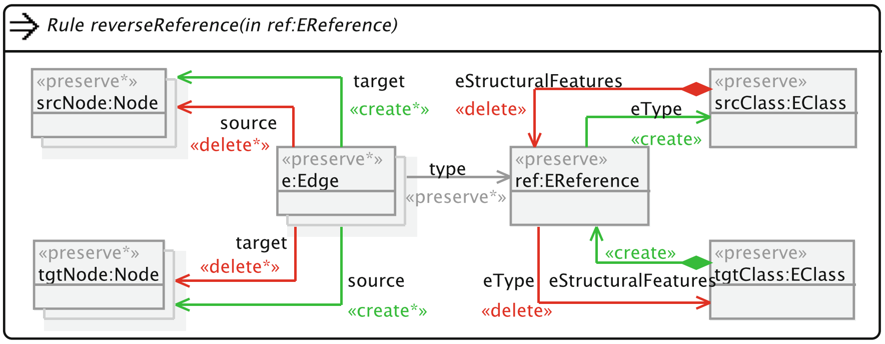
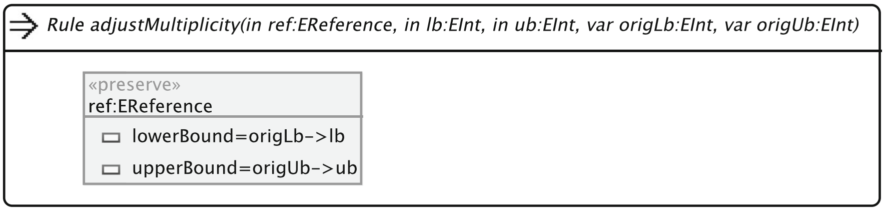


Fig. 9.9 reverseReference transformer (transformerRules)



Rule adjustMultiplicity(in ref:EReference, in lb:EInt, in ub:EInt, var origLb:EInt, var origUb:EInt)
«preserve»
ref:EReference
lowerBound=origLb->lb
upperBound=origUb->ub

Fig. 9.10 adjustMultiplicity transformer (transformerRules)

Instantiating the Propagation Rules

To propagate the tainting from a machine to its subsequent tray, we can use the addPathElement transformer. Basically, this transformer modifies the source GTS by introducing a new class between two classes linked by some association and updates any rules referencing this association. This transformer takes as arguments the name of the association to operate on, plus identifiers for the new class, reference and node, together with the multiplicity for the additional association. The following GTS PatternMachineOutTray (cf. Listing 9.7) is the result of applying this transformer on the target association of the Source class using the addPathElement transformer of the KAMPFamily family.

```
export gts PatternMachineOutTray {
  family: KAMPFamily
    using [
      addPathElement(kamp.Source.target, "newClass", "newRef",
      "newNode", 0, 1)
    ]
}
```

Listing 9.7 Constructing PatternMachineOutTray

```
auto-complete unique allow-from-empty
  map IPatternMachineOutTray2TaintablePLS {
  from interface_of{PatternMachineOutTray}
  to TaintablePLS
    type_mapping {
      class kamp.Source => pls.Machine
      class kamp.Target => pls.Tray
    }
}
```

Listing 9.8 A morphism from PatternMachineOutTray to the taintable PLS, ready for weaving tainting

```
export gts TaintedPLSMachineOutTray {
  weave(dontLabelNonKernelElements, preferMap2TargetNames): {
    map1: interface_of(PatternMachineOutTray)
    map2: IPatternMachineOutTray2TaintablePLS
  }
}
```

Listing 9.9 Constructing TaintedPLSMachineOutTray

All elements in KAMPS's metamodel are annotated as interface. New elements introduced by transformers are also annotated as interfaces. To construct new GTSs as a result of the amalgamation of previously defined GTSs, we need to define morphisms from a kernel interface to the system on which we wish to act, in this case the TaintablePLS GTS that resulted from the previous amalgamation. The instantiating morphism can now be defined from the interface of the PatternMachineOutTray GTS to the TaintablePLS GTS as shown in Listing 9.8.
Notice the use of the auto-complete directive, with which the mapping for other elements in the interface sub-GTS is automatically calculated. In particular, notice that we do not need to provide an explicit mapping for the new path element introduced. Notice also the use of the allow-from-empty directive as above.

We can now amalgamate this morphism and the inclusion of the interface of the PatternMachineOutTray GTS into itself to generate the TaintedPLSMachineOutTray GTS as shown in Listing 9.9.

The in link of the Transformer class goes from Transformer into Tray. However, we want the tainting to propagate following the inverse direction. We can get the required instantiation of the propagation rule by using the reverseReference transformer rule to reverse the link in the source rule. As before, once the source GTS is obtained we can define the morphism and then the amalgamation GTS (cf. Listing 9.10).

The tray link of the Operator class presents a new challenge. So far, we just needed to apply one transformer to be able to build the required morphism, but in this case, we need to both reverse the link and change its multiplicity. We just need

```
export gts PatternTransformerIn {
  family: KAMPFamily
  using [
    reverseReference (kamp.Source.target)
  ]
}

auto-complete unique allow-from-empty
  map PatternTransformerIn2PLSMachineOutTray {
  from interface_of{PatternTransformerIn}
  to TaintedPLSMachineOutTray
  type_mapping {
    class kamp.Target => pls.Transformer
    class kamp.Source => pls.Tray
  }
}

export gts TaintedPLSTransformerIn {
  weave(dontLabelNonKernelElements, preferMap2TargetNames): {
    map1: interface_of(PatternTransformerIn)
    map2: PatternTransformerIn2PLSMachineOutTray
  }
}
```

Listing 9.10 Construction of the TaintedPLSTransformerIn GTS

to specify the sequence of transformers that lead to the intended target as shown in Listing 9.11.

9.4.3 *The Final Taint-Propagating PLS*

After the consecutive instantiation of the Taintable GTS and of the KAMP GTS on the specific links specified in the morphisms, we get an extended PLS GTS in which the metamodel includes a tainted attribute in the NamedElement class and propagation rules propagating the tainting along the links between named elements. The extended PLS protects the semantics of the original PLS language, but in addition it now provides this additional functionality to identify the part of the model affected by any potential change as specified. In addition to the original rules in Fig. 9.2, the extended PLS GTS now also includes the rules depicted in Fig. 9.11. The application of these rules on an instance model in which some element is tainted, specifying a change, would result in an instance model in which all elements affected by the change are tainted, in accordance to the specified tainting propagation rules.

```
export gts PatternConveyorTray {
  family: KAMPFamily
  using [
    adjustMultiplicity (kamp.Source.target, 0, -1),
    reverseReference (kamp.Source.target)
  ]
}

auto-complete unique allow-from-empty
  map PatternOperatorTray2TaintedPLSConveyorTray {
  from interface_of{PatternConveyorTray}
  to TaintedPLSTransformerIn
  type_mapping {
    class kamp.Source => pls.Tray
    class kamp.Target => pls.Operator
  }
}

export gts TaintedPLSOperatorTray {
  weave(dontLabelNonKernelElements, preferMap2TargetNames): {
    map1: interface_of(PatternConveyorTray)
    map2: PatternOperatorTray2TaintedPLSConveyorTray
  }
}
```

Listing 9.11 Constructing TaintedPLSOperatorTray

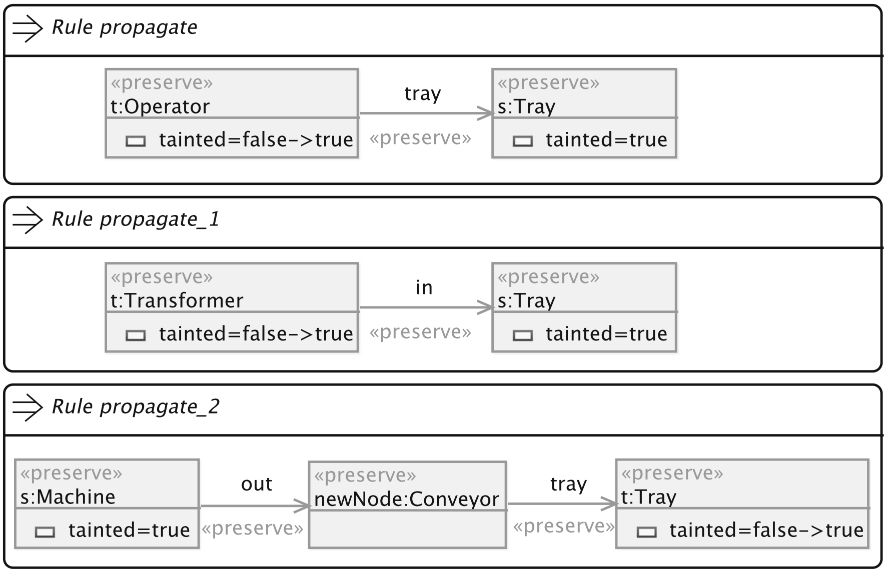


Fig. 9.11 The amalgamation adds these four rules to the behaviour of the PLS

9.5 Conclusions and Outlook

In this chapter, we have shown a tool and case study showing how the explicit specification of a language's operational semantics with graph transformations can make it possible to reuse analysis techniques for different domain-specific modelling languages. This approach allows the reuse of analysis techniques across different domain-specific languages reducing the effort required for different domains to benefit from particular analysis expertise. Because it enables a modular approach to analysis (as discussed also in Chaps. 4 and 5 of this book [Hei+21]), different analyses can be combined into one modelling language, so that different analyses can be enabled depending on what a project requires.

References

[AL94] Martin Abadi and Leslie Lamport. "An Old-Fashioned Recipe for Real Time". In: *ACM Transactions on Programming Languages and Systems* 16.5 (Sept. 1994), pp. 1543–1571.

[Cor+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. "Algebraic approaches to graph transformation I: Basic concepts and double pushout approach". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. 1997. Chap. 3.

[DMC12] Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki. "Intermodeling, Queries, and Kleisli Categories". In: *Conf. Fundamental Approaches to Software Engineering*. 2012, pp. 163–177. https://doi.org/10.1007/978-3-642-28872-2_12.

[Dur+17] Francisco Durán, Antonio Moreno-Delgado, Fernando Orejas, and Steffen Zschaler. "Amalgamation of Domain Specific Languages with Behaviour". In: *Journal of Logical and Algebraic Methods in Programming* 86 (1 2017), pp. 208–235. https://doi.org/10.1016/j.jlamp.2015.09.005.

[DZT13] Francisco Durán, Steffen Zschaler, and Javier Troya. "On the Reusable Specification of Non-functional Properties in DSLs". In: *5th Int'l Conf. on Software Language Engineering, SLE*. 2013, pp. 332–351. https://doi.org/10.1007/978-3-642-36089-3_19.

[EHC05] Gregor Engels, Reiko Heckel, and Alexey Cherchago. "Flexible Interconnection of Graph Transformation Modules". In: *Formal Methods in Software and Systems Modeling*. 2005, pp. 38–63. https://doi.org/10.1007/978-3-540-31847-7_3.

[Ehr+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006. https://doi.org/10.1007/3-540-31188-2.

[Ehr79] Hartmut Ehrig. "Introduction to the algebraic theory of graph grammars". In: *1st Graph Grammar Workshop*. 1979, pp. 1–69. https://doi.org/10.1007/BFb0025714.

[Eng+97] Gregor Engels, Reiko Heckel, Gabriele Taentzer, and Hartmut Ehrig. "A Combined Reference Model- and View-Based Approach to System Specification". In: *International Journal of Software Engineering and Knowledge Engineering* 7.4 (1997), pp. 457–477. https://doi.org/10.1142/S0218194097000266.

[GM04] Vincenzo Grassi and Raffaela Mirandola. "A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems". In: *Proc. Workshop on Models for Non-Functional Aspects of Component-Based Software*. 2004.

[GPS98a] Martin Große-Rhode, Francesco Parisi-Presicce, and Marta Simeoni. “Refinements of Graph Transformation Systems via Rule Expressions”. In: *6th Int’l Workshop Theory and Application of Graph Transformations*. 1998, pp. 368–382. https://doi.org/10.1007/978-3-540-46464-8_26.

[GPS98b] Martin Große-Rhode, Francesco Parisi-Presicce, and Marta Simeoni. “Spatial and Temporal Refinement of Typed Graph Transformation Systems”. In: *23rd Int’l Symposium Mathematical Foundations of Computer Science*. 1998, pp. 553–561. https://doi.org/10.1007/BFb0055805.

[Hei+21] Robert Heinrich, Francisco Durán, Carolyn L. Talcott, and Steffen Zschaler (eds.) *Composing Model-Based Analysis Tools*. Springer, 2021. https://doi.org/10.1007/978-3-030-81915-6.

[Joh+19] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Strüber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. “Searching for Optimal Models: Comparing Two Encoding Approaches”. In: *Journal of Object Technology* 18.3 (2019), 6:1–22. https://doi.org/10.5381/jot.2019.18.3.a6.

[Lam94] Leslie Lamport. “A Temporal Logic of Actions”. In: *ACM Transactions on Programming Languages and Systems* 16.3 (1994), pp. 872–923.

[Lar+07] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. “Attributed Graph Transformation with Node Type Inheritance”. In: *Theoretical Computer Science* 376 (2007), pp. 139–163. https://doi.org/10.1016/j.tcs.2007.02.001.

[LG13] Juan de Lara and Esther Guerra. “From Types to Type Requirements: Genericity for Model-Driven Engineering”. In: *Software and Systems Modelling* 12.3 (2013), pp. 453–474. https://doi.org/10.1007/s10270-011-0221-0.

[LG14] Juan de Lara and Esther Guerra. “Towards the flexible reuse of model transformations: A formal approach based on graph transformation”. In: *Journal of Logical and Algebraic Methods in Programming* 83.5–6 (2014). 24th Nordic Workshop on Programming Theory (NWPT 2012), pp. 427–458. issn: 2352-2208. https://doi.org/10.1016/j.jlamp.2014.08.005.

[Mor+14] Antonio Moreno-Delgado, Francisco Durán, Steffen Zschaler, and Javier Troya. “Modular DSLs for Flexible Analysis: An e-Motions Reimplementation of Palladio”. In: *Proc. 10th European Conf. on Modelling Foundations and Applications)*. 2014, pp. 132–147. https://doi.org/10.1007/978-3-319-09195-2_9.

[Nau86] Peter Naur. “Programming as Theory Building”. In: *Microprocessing and Microprogramming* 15 (1986), pp. 253–261. https://doi.org/10.1016/0165-6074(85)90032-8.

[Reu+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press, 2016.

[Ros+15] Kiana Rostami, Johannes Stammel, Robert Heinrich, and Ralf H. Reussner. “Architecture-based Assessment and Planning of Change Requests”. In: *11th International ACM SIGSOFT Conference on Quality of Software Architectures*. 2015, pp. 21–30. https://doi.org/10.1145/2737182.2737198.

[Ste+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. EMF: *Eclipse Modeling Framework*. Addison-Wesley Professional, 2009.

[Str+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. “Henshin: A Usability-Focused Framework for EMF Model Transformation Development”. In: *10th Int’l Conf on Graph Transformations*. 2017, pp. 196–208.

[Tro+13] Javier Troya, Antonio Vallecillo, Francisco Durán, and Steffen Zschaler. “Model-Driven Performance Analysis of Rule-Based Domain Specific Visual Models”. In: *Information and Software Technology* 55.1 (2013), pp. 88–110. https://doi.org/10.1016/j.infsof.2012.07.009.

[War94] Martin P.Ward. “Language-oriented programming”. In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161. URL: http://www.gkc.org.uk/martin/papers/middle-out-t.pdf.

[ZD17] Steffen Zschaler and Francisco Durán. “GTS Families for the Flexible Composition of Graph Transformation Systems”. In: *20th Int’l Conf. Fundamental Approaches to Software Engineering*. 2017, pp. 208–225. https://doi.org/10.1007/978-3-662-54494-5_12.