Composition of Graph-Transformation-Based DSL Definitions by Amalgamation

Francisco Durán(✉)

University of Málaga, Málaga, Spain
duran@lcc.uma.es

Abstract. Given a graph-grammar formalization of DSLs, we build on graph transformation system morphisms to define parameterized DSLs and their instantiation by an amalgamation construction. Results on the protection of the behavior along the induced morphisms allow us to safely combine definitions of DSLs to build more complex ones. We illustrate our proposal on our e-Motions definition of the Palladio DSL. The resulting DSL allows us to carry on performance analysis on Palladio models.

1 Introduction

In Model-Driven Engineering (MDE) [43], models are used to specify, simulate, analyze, modify, and generate code. One of the key ingredients making this approach particularly attractive is the use of domain-specific languages (DSLs) [49] for the definition of such models. DSLs offer concepts specifically targeted at a particular domain, which allow experts in such domains to express their problems and requirements in their own languages. On the other hand, the higher amount of knowledge embedded in these concepts allows for much more complete and specialized generation of executable solution code from DSL models [30].

The application of these techniques to different domains has resulted in the proliferation of DSLs of very different nature: the more specific for a particular domain a DSL is, the more effective it is. However, DSLs are only viable if their development can be made efficient. With this goal in mind, DSLs are often defined by specifying their syntax in some standard formalisms, such as MOF, thus facilitating the use of generic frameworks for the management of models, including their composition, the definition of model transformations, use of model editors, etc.

Syntax is however just part of the story. Without a definition of the operational behavior of the defined DSLs, we will not be able to simulate or analyze the defined models. In recent years, different formalisms have been proposed for the definition of the behavior of DSLs, including UML behavioral models [19,22], abstract state machines [3,10], or in-place model transformations [9,39]. Between all these approaches, we find the use of in-place model transformations particularly powerful, not only because its expressiveness, but also because it facilitates its integration with the rest of the MDE environment and tools.

S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 1–20, 2014.
DOI: 10.1007/978-3-319-12904-4_1

While we have reasonably good knowledge of how to modularize DSL syntax, the modularization of language semantics is an as yet unsolved issue. Given a graph-grammar [6,14,42] formalization of DSLs, we build on graph transformation system (GTS) morphisms to define composition operations on DSLs. Specifically, we define parameterized GTSs, that is, GTSs which have other GTSs as parameters. The instantiation of such parameterized GTSs is then provided by an amalgamation construction. We present formal results about GTSs and GTSs morphisms between them. Specifically, we are interested on how these morphisms preserve or protect behavior, and what behavior-related properties may be guaranteed on the morphisms induced by the amalgamation construction defining the instantiation of parameterized GTSs. Of particular interest for our goals is the identification of the circumstances in which we can guarantee protection of behavior when DSLs get instantiated.

In the rest of the paper, we propose the use of parameterized DSLs, we present their implementation in the e-Motions system, and show its potential presenting the definition of the e-Motions implementation of a significant part of the Palladio DSL. Although we motivate and illustrate our approach using the e-Motions language [37], our proposal is language-independent, and all the results are presented for GTSs and adhesive HLR systems [16,34]. e-Motions graphical specifications are translated into Maude specifications [38]. Given this transformation, models in DSLs developed in e-Motions, may be "simulated" in accordance to the given semantics. Since the resulting specification is a valid theory in rewriting logic, Maude's formal tools, as its reachability analysis tool or its model checker, may be used on it.

The rest of the paper is structured as follows. Section 2 introduces behavior-reflecting and -protecting GTS morphisms, the construction of amalgamations in the category of GTSs and GTS morphisms, and several results on these amalgamations. Section 3 presents the e-Motions definition of the Palladio DSL and how the composition operations presented in Sect. 2 are used to provide mechanisms to carry on performance-related monitoring and analysis of systems. The paper presents some related work in Sect. 4 and finishes with some conclusions and future work in Sect. 5.

2 Graph Transformation and GTS Amalgamations

Graph transformation [14,42] is a formal, graphical and natural way of expressing graph manipulation based on rewriting rules. In graph-based modelling (and meta-modelling), graphs are used to define the static structures, such as class and object ones, which represent visual alphabets and sentences over them. A more detailed presentation of the results in this section may be found in [11].

2.1 Rules, Rule Morphisms, and Rule Amalgamations

Our formalisation is developed for weak adhesive high-level replacement (HLR) categories [14], making it much more general. The concepts of adhesive and

(weak) adhesive HLR categories abstract the foundations of a general class of models, and come together with a collection of general semantic techniques [16,34]. Thus, e.g., given proofs for adhesive HLR categories of general results such as the Local Church-Rosser, or the Parallelism and Concurrency Theorem, they are automatically valid for any category which is proved an adhesive HLR category. The category of typed attributed graphs, the one of interest to us, was proved to be adhesive HLR in [18].

In the DPO approach to graph transformation, a rule with application conditions p is of the form $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$ with graphs L, K, and R, called, respectively, left-hand side, interface, and right-hand side, some kind of monomorphisms (typically, inclusions) l and r, and ac a *(nested) application condition* on L. A graph transformation system (GTS) is a pair (P, π) where P is a set of rule names and π is a function mapping each rule name p into a rule $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$.

An application of a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$ to a graph G via a match $m : L \to G$, such that m satisfies ac, written $m \models ac$, is constructed as two gluings (1) and (2), which are pushouts in the corresponding graph category, leading to a direct transformation $G \overset{p,m}{\Longrightarrow} H$.

$$
\begin{array}{ccccccc}
ac & \triangleright & L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\
 & & {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\
 & & G & \longleftarrow & D & \longrightarrow & H
\end{array}
$$

Application conditions may be positive or negative. Positive application conditions have the form $\exists a$, for a monomorphism $a : L \to C$, and demand a certain structure in addition to L. Negative application conditions of the form $\nexists a$ forbid such a structure. A match $m : L \to G$ satisfies a positive application condition $\exists a$ if there is a monomorphism $q : C \to G$ satisfying $q \circ a = m$. A matching m satisfies a negative application condition $\nexists a$ if there is no such monomorphism. Given an application condition $\exists a$ or $\nexists a$, for a monomorphism $a : L \to C$, another application condition ac can be established on C, giving place to nested application conditions [25]. Given an application condition ac on L and a monomorphism $t : L \to L'$, then there is an application condition $\mathsf{Shift}(t, ac)$ on L' such that for all $m' : L' \to G$, $m' \models \mathsf{Shift}(t, ac) \leftrightarrow m = m' \circ t \models ac$.

$$
ac \triangleright \; L \xrightarrow{\ t\ } L' \triangleleft \; \mathsf{Shift}(t, ac)
\qquad
\begin{array}{ccc}
 & & \\
{\scriptstyle m}\searrow & & \swarrow{\scriptstyle m'} \\
 & G &
\end{array}
$$

To improve readability, we assume projection functions ac, lhs and rhs, returning, respectively, the application condition, left-hand side and right-hand side of a rule. Thus, given a rule $r = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, $ac(r) = ac$, $lhs(r) = L$, and $rhs(r) = R$.

We only consider injective matches, that is, monomorphisms. If the matching m is understood, a DPO transformation step $G \overset{p,m}{\Longrightarrow} H$ will be simply written $G \overset{p}{\Longrightarrow} H$. A transformation sequence $\rho = \rho_1 \dots \rho_n : G \Rightarrow^* H$ via rules $p_1, \dots, p_n$ is a sequence of transformation steps $\rho_i = (G_i \overset{p_i,m_i}{\Longrightarrow} H_i)$ such that $G_1 = G$, $H_n = H$, and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$. The category of transformation sequences over an adhesive category $\mathbf{C}$, denoted by $\mathbf{Trf}(\mathbf{C})$, has all graphs in $|\mathbf{C}|$ as objects and all transformation sequences as arrows.

Parisi-Presicce proposed in [36] a notion of rule morphism very similar to the one below, although we consider rules with application conditions, and require the commuting squares to be pullbacks instead of pushouts.

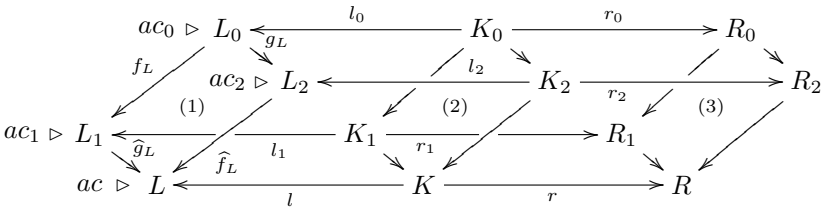
Definition 1 ***(From [11], Rule morphism).*** *Given graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i,\ ac_i)$, for $i = 0, 1$, a rule morphism $f : p_0 \to p_1$ is a tuple $f = (f_L, f_K, f_R)$ of graph monomorphisms $f_L : L_0 \to L_1$, $f_K : K_0 \to K_1$, and $f_R : R_0 \to R_1$ such that the squares with the span morphisms l_0, l_1, r_0, and r_1 are pullbacks, as in the diagram below, and such that $ac_1 \Rightarrow \mathsf{Shift}(f_L, ac_0)$.*

$$\begin{array}{ccccccccc} p_0 & : & ac_0 & \triangleright & L_0 & \xleftarrow{l_0} & K_0 & \xrightarrow{r_0} & R_0 \\ {\scriptstyle f}\downarrow & & & & {\scriptstyle f_L}\downarrow & pb & {\scriptstyle f_K}\downarrow & pb & \downarrow{\scriptstyle f_R} \\ p_1 & : & ac_1 & \triangleright & L_1 & \xleftarrow[l_1]{} & K_1 & \xrightarrow[r_1]{} & R_1 \end{array}$$

Asking that the two squares are pullbacks means, precisely, to preserve the "structure" of objects. I.e., we preserve what should be deleted, what should be added, and what must remain invariant. Of course, pushouts also preserve the created and deleted parts, but they reflect this structure as well, which we do not want in general. With componentwise identities and composition, rule morphisms define the category **Rule**.

A key concept in the constructions in Sect. 2.3 is that of *rule amalgamation* [2]. The amalgamation of two rules p_1 and p_2 glues them together into a single rule $\tilde{p}$ to obtain the effect of the original rules. I.e., the simultaneous application of p_1 and p_2 yields the same successor graph as the application of the amalgamated rule $\tilde{p}$. The possible overlapping of rules p_1 and p_2 is captured by a rule p_0 and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$.

Definition 2 ***(From [11], Rule amalgamation).*** *Given graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, for $i = 0, 1, 2$, and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$, the amalgamated production $p_1 +_{p_0} p_2$ is the production $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$ in the diagram below, where subdiagrams (1), (2) and (3) are pushouts, l and r are induced by the universal property of (2) so that all subdiagrams commute, and $ac = \mathsf{Shift}(\widehat{f}_L, ac_2) \wedge \mathsf{Shift}(\widehat{g}_L, ac_1)$.*



Notice that in the above diagram all squares are either pushouts or pullbacks (by the van Kampen property [34]) which means that all their arrows are monomorphisms (by being an adhesive HLR category).

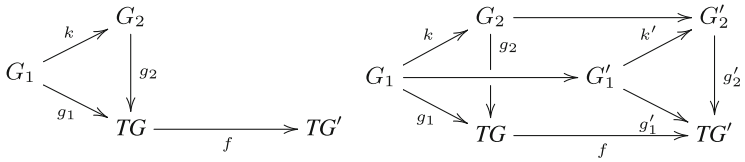
2.2 Typed Graph Transformation Systems

A (directed unlabeled) *graph* $G = (V, E, s, t)$ is given by a set of nodes (or vertices) V, a set of edges E, and source and target functions $s, t \colon E \to V$. Given graphs $G_i = (V_i, E_i, s_i, t_i)$, with $i = 1, 2$, a graph homomorphism $f \colon G_1 \to G_2$ is a pair of functions $(f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. With componentwise identities and composition this defines the category **Graph**.

Given a distinguished graph TG, called *type graph*, a *TG-typed graph* (G, g_G), or simply *typed graph* if TG is known, consists of a graph G and a typing homomorphism $g_G : G \to TG$ associating with each vertex and edge of G its type in TG. However, to enhance readability, when the typing morphism g_G can be considered implicit, we will often refer to a typed graph (G, g_G) just as G. A TG-typed graph morphism between TG-typed graphs $(G_i, g_i : G_i \to TG)$, with $i = 1, 2$, denoted $f \colon (G_1, g_1) \to (G_2, g_2)$, is a graph morphism $f \colon G_1 \to G_2$ which preserves types, i.e., $g_2 \circ f = g_1$. $\mathbf{Graph}_{TG}$ is the category of TG-typed graphs and TG-typed graph morphisms, which is the comma category **Graph** over TG.

If the underlying graph category is adhesive (resp., adhesive HLR, weakly adhesive) then so are the associated typed categories [14], and therefore all definitions in Sect. 2.1 apply to them. A TG-typed graph transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$ is a span of injective TG-typed graph morphisms and a (nested) application condition on L. Given TG-typed graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, with $i = 1, 2$, a typed rule morphism $f \colon p_1 \to p_2$ is a tuple (f_L, f_K, f_R) of TG-typed graph monomorphisms such that the squares with the span monomorphisms l_i and r_i, for $i = 1, 2$, are pullbacks, and such that $ac_2 \Rightarrow \mathsf{Shift}(f_L, ac_1)$. TG-typed graph transformation rules and typed rule morphisms define the category $\mathbf{Rule}_{TG}$, which is the comma category **Rule** over TG.

Following [6], we use forward and backward retyping functors to deal with graphs over different type graphs. A graph morphism $f \colon TG \to TG'$ induces a forward retyping functor $f^{>} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$, with $f^{>}(g_1) = f \circ g_1$ and $f^{>}(k \colon g_1 \to g_2) = k$ by composition, as shown in the diagram in Fig. 1(a). Similarly, such a morphism f induces a backward retyping functor $f^{<} : \mathbf{Graph}_{TG'} \to$



(a) Forward retyping functor. (b) Backward retyping functor.

Fig. 1. Forward and backward retyping functors.

Graph$_{TG}$, with $f^<(g'_1) = g_1$ and $f^<(k' : g'_1 \to g'_2) = k : g_1 \to g_2$ by pullbacks and mediating morphisms as shown in the diagram in Fig. 1(b). Since, as said above, we refer to a TG-typed graph $G \to TG$ just by its typed graph G, leaving TG implicit, given a morphism $f : TG \to TG'$, we may refer to the TG'-typed graph by $f^>(G)$.

A typed graph transformation system over a type graph TG, is a graph transformation system where the given graph transformation rules are defined over the category of TG-typed graphs. Since we deal with GTSs over different type graphs, we will make explicit the given type graph. This means that, from now on, a typed GTS is a triple (TG, P, π) where TG is a type graph, P is a set of rule names and π is a function mapping each rule name p into a rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$ typed over TG.

The set of transformation rules of a GTS specifies a behavior in terms of the derivations obtained via such rules. A GTS morphism defines then a relation between its source and target GTSs by providing an association between their type graphs and rules.

Definition 3 ***(From [11], GTS morphism).*** *Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is given by a morphism $f_{TG} : TG_0 \to TG_1$, a surjective mapping $f_P : P_1 \to P_0$ between the sets of rule names, and a family of rule morphisms $f_r = \{f^p : f^>_{TG}(\pi_0(f_P(p))) \to \pi_1(p)\}_{p \in P_1}$.*

Given a GTS morphism $f : GTS_0 \to GTS_1$, each rule in GTS_1 extends a rule in GTS_0. However if there are internal computation rules in GTS_1 that do not extend any rule in GTS_0, we can always consider that the empty rule is included in GTS_0, and assume that those rules extend the empty rule. Notice that to deal with rule morphisms defined on rules over different type graphs we retype one of the rules. Typed GTSs and GTS morphisms define the category **GTS**.

2.3 GTS Amalgamations and Preservation of Behavior

Given a GTS morphism $f : GTS_0 \to GTS_1$, we say that it *reflects* behavior if for any derivation that may happen in GTS_1 there exists a corresponding derivation in GTS_0.

Definition 4 ***(From [11], Behavior-reflecting GTS morphism).*** *Given transformation systems* $GTS_i = (TG_i, P_i, \pi_i)$*, for* $i = 0, 1$*, a GTS morphism* $f\colon GTS_0 \to GTS_1$ *is behavior-reflecting if for all graphs* G*,* H *in* $|\mathbf{Graph}_{TG_1}|$*, all rules* p *in* P_1*, and all matches* $m\colon lhs(\pi_1(p)) \to G$ *such that* $G \overset{p,m}{\Longrightarrow} H$*, then* $f^{<}_{TG}(G) \overset{f_P(p), f^{<}_{TG}(m)}{\Longrightarrow} f^{<}_{TG}(H)$ *in* GTS_0*.*

We call *extension morphisms* to those morphisms between GTSs that only add to the transformation rules elements not in their source type graph. All extension GTS morphisms are behavior-reflecting [11].

Definition 5 ***(From [11], Extension GTS morphism).*** *Given graph transformation systems* $GTS_i = (TG_i, P_i, \pi_i)$*, for* $i = 0, 1$*, a GTS morphism* $f\colon GTS_0 \to GTS_1$*, with* $f = (f_{TG}, f_P, f_r)$*, is an extension morphism if* f_{TG} *is a monomorphism and for each* $p \in P_1$*,* $\pi_0(f_P(p)) \equiv f^{<}_{TG}(\pi_1(p))$*.*

When a DSL is extended with alien elements that do not interfere with its behavior, e.g., to measure or to verify some property, we need to guarantee that such an extension does not change the semantics of the original DSL. Specifically, we need to guarantee that the behavior of the resulting system is exactly the same, that is, that any derivation in the source system also happens in the target one (behavior preservation), and any derivation in the target system was also possible in the source one (behavior reflection). The following definition of behavior-protecting GTS morphism captures the intuition of a morphism that both reflects and preserves behavior, that is, that establishes a bidirectional correspondence between derivations in the source and target GTSs.

Definition 6 ***(From [11], Behavior-protecting GTS morphism).*** *Given transformation systems* $GTS_i = (TG_i, P_i, \pi_i)$*, for* $i = 0, 1$*, a GTS morphism* $f\colon GTS_0 \to GTS_1$ *is behavior-protecting if for all graphs* G *and* H *in* $|\mathbf{Graph}_{TG_1}|$*, all rules* p *in* P_1*, and all matches* $m : lhs(\pi_1(p)) \to G$*,* $g^{<}_{TG}(G) \overset{g_P(p), g^{<}_{TG}(m)}{\Longrightarrow} g^{<}_{TG}(H) \iff G \overset{p,m}{\Longrightarrow} H$

We find in the literature definitions of behavior-preserving morphisms as morphisms in which the rules in the source GTS are included in the set of rules of the target GTS (see, e.g., [24,28]). Although these morphisms trivially preserve behavior, they are not useful for our purposes. Notice that, in our case, in addition to adding new rules, we are enriching the rules themselves.

GTS amalgamation provides a very convenient way of composing GTSs. Theorem 1 below establishes behavior-related properties on the induced morphisms.

Definition 7 ***(From [11], GTS Amalgamation).*** *Given typed graph transformation systems* $GTS_i = (TG_i, P_i, \pi_i)$*, for* $i = 0, 1, 2$*, and GTS morphisms* $f\colon GTS_0 \to GTS_1$ *and* $g\colon GTS_0 \to GTS_2$*, the amalgamated GTS* $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ *is the GTS* $(\widehat{TG}, \widehat{P}, \widehat{\pi})$ *constructed as follows. We first construct the pushout of typing graph morphisms* $f_{TG}\colon TG_0 \to TG_1$ *and* $g_{TG}\colon TG_0 \to TG_2$*, obtaining morphisms* $\widehat{f}_{TG}\colon TG_2 \to \widehat{TG}$ *and* $\widehat{g}_{TG}\colon TG_1 \to \widehat{TG}$*. The pullback of set morphisms* $f_P : P_1 \to P_0$ *and* $g_P : P_2 \to P_0$ *defines morphisms* $\widehat{f}_P\colon \widehat{P} \to P_2$

and $\widehat{g}_P \colon \widehat{P} \to P_1$. *Then, for each rule* p *in* $\widehat{P}$, *the rule* $\widehat{\pi}(p)$ *is defined as the amalgamation of rules* $\widehat{f^{>}_{TG}}(\pi_2(\widehat{f}_P(p)))$ *and* $\widehat{g^{>}_{TG}}(\pi_1(\widehat{g}_P(p)))$ *with respect to the kernel rule* $\widehat{f^{>}_{TG}}(g^{>}_{TG}(\pi_0(g_P(\widehat{f}_P(p)))))$.

$$\begin{array}{ccc} GTS_0 & \xrightarrow{\quad f \quad} & GTS_1 \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \widehat{g}} \\ GTS_2 & \xrightarrow{\quad \widehat{f} \quad} & \widehat{GTS} \end{array}$$

The following result gives conditions under which behavior-related guarantees can be established on the morphisms induced by the amalgamation construction.

Theorem 1 ***(From [11]).*** *Given typed transformation systems* $GTS_i = (TG_i, P_i, \pi_i)$, *for* $i = 0, 1, 2$, *and the amalgamation* $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ *of GTS morphisms* $f \colon GTS_0 \to GTS_1$ *and* $g \colon GTS_0 \to GTS_2$, *if* f *is a behavior-reflecting GTS morphism, then* $\widehat{f}$ *is a monomorphism, and if* g *is an extension and behavior-protecting morphism, then* $\widehat{g}$ *is behavior-protecting as well.*

$$\begin{array}{ccc} GTS_0 & \xrightarrow{\quad f \quad} & GTS_1 \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \widehat{g}} \\ GTS_2 & \xrightarrow{\quad \widehat{f} \quad} & \widehat{GTS} \end{array}$$

3 Non-functional Properties as Parameterized Domain Specific Languages

In previous work [12,48], we have explored the modular definition of non-functional properties as parameterized DSLs in the e-Motions framework [37]. These ideas were further exploited in [35] to provide a modular reimplementation of a substantive part of the Palladio Architecture Simulator [26] to perform predictive analysis of architectural software models. In particular, we re-implemented the Palladio Component Model [1], its workload model, and parts of its stochastic expressions model.

We explicitly modeled simulations as graph transformations in the e-Motions framework, and then, each NFP to be analyzed was modeled as an independent, parameterized DSL ready to be composed with the base Palladio model. The modular definition of NFPs as separate, parameterized DSLs allows its reuse, but also makes it easy to define additional NFPs to be analyzed. For a particular analysis problem, the relevant NFP DSLs can then be selected from a library and composed as required.

The results presented in Sect. 2.3 provides guarantees for preservation of semantics under composition, that is, the consideration of additional NFPs (satisfying certain restrictions) do not change the behavior of the system being modeled.

In this section, we introduce Palladio, e-Motions, and then the definition of the Palladio DSL in the e-Motions system. We pay special attention to the definition of observers and how they are 'woven' with the Palladio system to enrich the definition of its behavior for the observation of NFPs.

3.1 The Palladio DSL

The Palladio Architecture Simulator [26] is a predictive software analysis tool. It consists of a number of metamodels, foremost the Palladio Component Model (PCM) [1], that allow the high-level modeling of component-based architectures and their properties relevant for performance and reliability analysis. Palladio supports predictive analyses by transformation into a program that runs a simulation of the architecture's behavior, and by transforming to formalisms more amenable to analysis—e.g., Queuing Petri Nets.

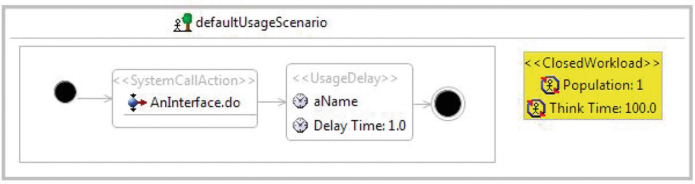
Figure 2 shows the usage model and the component specification of a very simple example, provided as part of the distribution of the Palladio Architecture Simulator. The usage model in Fig. 2(a) specifies the way tasks arrive into the system. In Palladio, the work load may be either closed or open. To be *closed* (ClosedWorkload object) means that the number of requests is fixed by the population attribute, and their corresponding inter-arrival rate given by the think time attribute. Alternatively, an OpenedWorkload object represents an infinite stream of arrivals. According to the usage model in Fig. 2(b), each work arriving to the system consists on a system call action to a component, AnInterface.do, and then a delay with a fixed time of 1.0 time units.

Figure 2(b) shows the specification of the component, in which the control flow may branch into one of two flows. Each branch is associated with a particular branch probability to indicate the likelihood of a particular branch being taken. Finally, resource demands, i.e. CPU or HDD, are expressed as probability distributions. This is the kind of information required to perform execution-time analysis on the component's behavior as is standard in software performance engineering (see, e.g., [45]). In addition, we could model failure information to support reliability analysis.

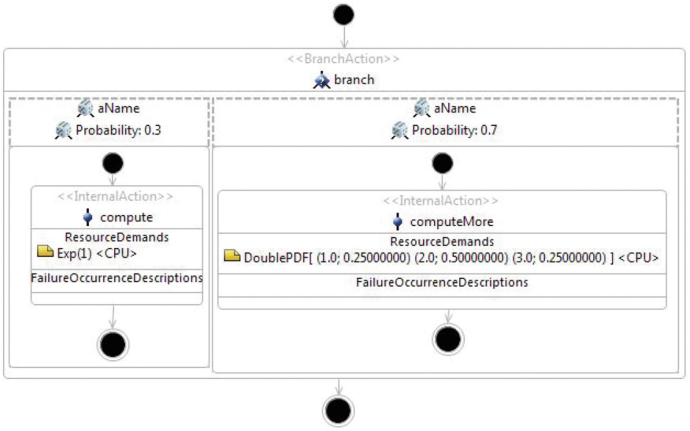
The Palladio Simulator offers the results of the analysis of performance and reliability of the system being analyzed in different formats.

3.2 The e-Motions System

e-Motions [37] is a graphical framework that supports the specification, simulation, and formal analysis of real-time systems. It provides a way to graphically specify the dynamic behavior of DSLs using their concrete syntax, making this task very intuitive. The abstract syntax of a DSL is specified as an Ecore metamodel, which defines all relevant concepts—and their relations—in the language. Its concrete syntax is given by a GCS (Graphical Concrete Syntax) model, which attaches an image to each language concept. Then, its behavior is specified with (graphical) in-place model transformations.



(a) Usage Model.



(b) Resource-Demanding Service-Effect specification (RDSEFF).

Fig. 2. *Minimum Example*: Workload and component specification in Palladio.

In-place transformations are defined by rules, each of which represents a possible *action* of the system. These rules are of the form $[\text{NAC}]^* \times \text{LHS} \rightarrow \text{RHS}$, where LHS (left-hand side), RHS (right-hand side) and NAC (negative application conditions) are model patterns that represent certain (sub-)states of the system. The LHS and NAC patterns express the conditions for the rule to be applied, whereas the RHS represents the effect of the corresponding action. A LHS may also have positive conditions, which are expressed, as any expression in the RHS, using OCL [40]. Thus, a rule can be applied, i.e., triggered, if a match of the LHS is found in the model, its conditions are satisfied, and none of its NAC patterns occurs. If several matches are found, one of them is non-deterministically chosen and applied, giving place to a new model where the matching objects are substituted by the appropriate instantiation of its RHS pattern. The transformation of the model proceeds by applying the rules on sub-models of it in a non-deterministic order, until no further transformation rule is applicable.

e-Motions provides a model of time, supporting features like duration, periodicity, etc., and mechanisms to state action properties. There are two types of rules to specify time-dependent behavior, namely, *atomic* and *ongoing* rules. Atomic rules represent atomic actions with a duration. Atomic rules with duration zero are called *instantaneous* rules. Ongoing rules represent actions that

progress continuously over time while the rule's preconditions (LHS and not NACs) hold. Both atomic and ongoing rules can be scheduled, or be given an execution interval. From a DSL definition, e-Motions generates an executable Maude [5] specification which can be used for simulation and analysis [38]. Other tools in the Maude formal environment, as its model checker or its reachability analysis tool, can also be used on this specification.

3.3 An e-Motions Re-implementation of Palladio

As for any DSL, the definition of the PCM includes its abstract syntax, its concrete syntax and its behavior. Since Palladio has been developed following MDE principles, and specifically it is implemented using the Eclipse Modeling Framework, its metamodel may be used as abstract syntax definition of Palladio in e-Motions.[1] Palladio models consists of several views, namely UsageModel, System, etc., corresponding to the different developer roles participating in the architecture of a system. These models are conformant to metamodels Core PCM, StoEx, Units, ... used by the different Eclipse plug-ins in the PCM Bench. As we will see in Sect. 3, using the PCM as abstract syntax will allow us to take models generated in the Palladio Simulator into e-Motions, and to use them to perform simulations in the e-Motions definition of Palladio.

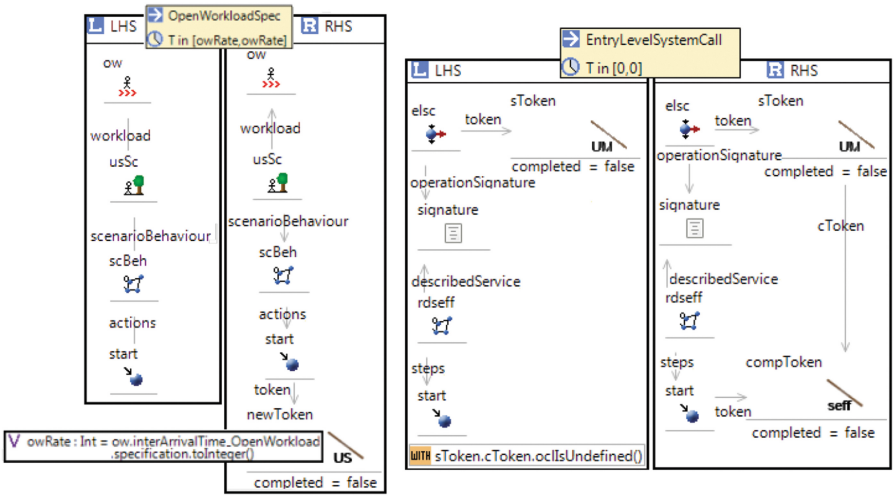
The concrete syntax is provided by a GCS model in which each concept in the abstract syntax of the DSL being defined is linked to an image. Since these images are used to graphically represent Palladio models in e-Motions, we have used the same images that the Palladio Simulator uses to represent these concepts. This way, we maintain the PCM's look in the e-Motions definition (see rules in Fig. 3).

In e-Motions, we describe how systems evolve by describing all possible changes of the models by corresponding visual time-aware in-place transformation rules. We may visualize each execution of a Palladio model has a token moving around such model. An action with a token has the control of execution. In fact, there might be several concurrent executions, since new tasks may keep arriving to the system, depending on its work load. The execution of each of these tasks proceeds independently, as far as the required resources are available.

For illustration purposes, Fig. 3 shows two of the rules defining the behavior of Palladio in e-Motions. As above explained, an open workload specifies an infinite stream of tasks arriving at the system with some inter-arrival time given by a random variable with some probability distribution. Each generated task executes the specified scenario, and then leave the system. Figure 3(a) shows the OpenWorkloadSpec rule, which specifies the behavior of a UsageScenario usSc with an OpenWorkload ow. When the rule is triggered, a new token is added to the first action of the system, i.e., the start action. The rule is fired every owRate, which is a local variable whose value is given by ow's random variable.

A ScenarioBehavior, which is included in a UsageScenario, as the one shown in Fig. 2, describes the behavior of the system components by using actions Start,

[1] The actual metamodel used is a conservative extension of the PCM to include additional concepts such as tokens, see below. The interested reader is referred to [35] for details.



(a) OpenWorkload rule. (b) Component call.

Fig. 3. New task rule specification.

Stop, EntryLevelSystemCall, Branch, and Loop. Figure 3(b) models the EntryLevelSystemCall action, which is used to invoke an operation in a component. If a (sub)-state matches the LHS of the rule, the SToken object associated to the EntryLevelSystemCall action remains in this action, while a new CToken is created and linked to the start action of the invoked component (effectively building up a call stack). As the rule's header shows, this rule is instantaneous (it takes zero time).

The complete e-Motions definition of the Palladio DSL is available at http://atenea.lcc.uma.es/Palladio.

Once the whole DSL has been defined, and given a model as initial state, it may be simulated by applying the rules describing its behavior. This model does not collect information on NFPs, and therefore is not ready for performance analysis. We enrich them later, as explained in the following section.

3.4 Parameterized DSL for NFP Observation

Troya, Rivera and Vallecillo proposed in [47] an approach for the specification and monitoring of non-functional properties of DSLs using *observers*. Observers are objects with which we extend the e-Motions definition of systems for the analysis of NFPs by simulation, such as mean and maximum cycle times, busy and idle cycles of operation units, throughput, mean-time between failures, etc. We explored in [12,48] how to define observers generically and independently from any system, so that they can afterwards be woven and merged with different systems. Given systems described as DSLs and generic DSLs defining the different observers, we can use the composition mechanisms presented in Sect. 2.3 to combine them. The result is that we can use the combined enriched system DSL to

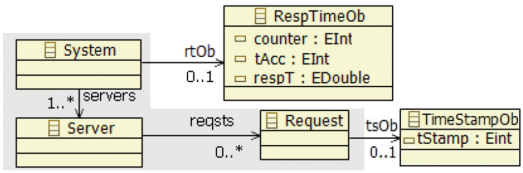
monitor NFPs of our systems. Theorem 1 proves that, given very natural requirements on the observers and the instantiating mappings, the system thus obtained is a conservative enrichment of the original system, in the sense that the observers added *do not change the behavior of the system.*

Given an e-Motions definition of Palladio as the one presented in Sect. 3.3, we can then enrich it with the definition of the observers we wish, which can be selected from a library of generically specified observers. Specifically, we can select those observers that monitor the properties available in the Palladio Simulator, but also others that monitor other properties. The NFPs chosen can then be analysed by simulation.

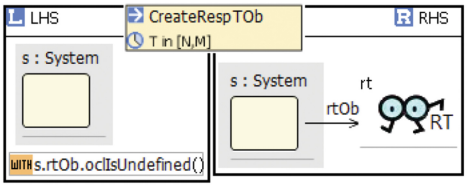
Let us consider a generic DSL for monitoring the *response time*, which is one of the properties available in Palladio. Response time can be defined as the time that elapses since a request arrives to a system until it is served. Hence, the same generic notion allows us to measure the response time of information packets being delivered through a network, of cars being manufactured in a production line, or of passengers checking-in in an airport. Given a system description, to measure response time, we just need to register the time at which requests arrive to the system, and the time at which they are completed. With this data and a simple calculation, we can easily get the response time.

A generic DSL achieving this is shown in Fig. 4. Its abstract syntax (the metamodel in Fig. 4(a)) contains three generic and two concrete classes—generic classes are shown with a shaded background. System, Server and Request are parameter classes to be instantiated by specific classes, as explained below. The System class represents the whole system, which is composed of a set of Servers. These, in turn, can have Requests that they have to process. The class RespTimeOb represents the observer for measuring the response time. Note that there is yet another observer in this metamodel, TimeStampOb, used to store the times at which Requests arrive.

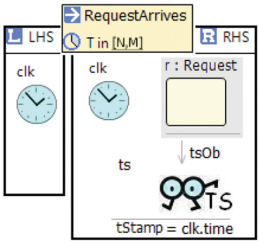
The behavior of this DSL is defined by the three in-place transformation rules in Fig. 4, in which parametric concepts have no concrete syntax, they are depicted as boxes, and have a shaded background. Observer objects have a concrete syntax, that will also be used to depict them in the woven rules (see below). Rule CreateRespTOb deals with the creation of the response time observer. Its LHS includes a condition that avoids the creation of new observer objects if there is one, ensuring that only one of these observers is created per instantiated object. The observer is associated to the system in its RHS. Rule RequestArrives generates a time stamp observer whenever a new Request appears. The observer gets associated to the Request and keeps the time at which it appears in the system—note the presence of the system object Clock, which provides the current time. Finally, rule CompletedRequest computes the response time every time a Request is consumed—the Request and its associated observer have disappeared in the RHS. Attribute counter of RespTimeOb keeps the number of completed Requests, while tAcc contains the addition of cycle times of all Requests, i.e., the time they have spent in the system. Finally, attribute respT uses the former two attributes to calculate the response time of the System.



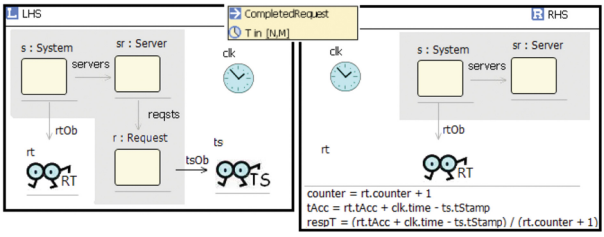
(a) Abstract syntax



(b) Behavior: CreateRespTOb



(c) Behavior: RequestArrives



(d) Behavior: CompletedRequest

Fig. 4. Response Time observer DSL definition.

3.5 Adding Observers to System Specifications

To add observers to our e-Motions specifications, we may compose the observer DSLs with the DSL of our system, the e-Motions definition of Palladio in our case. Let us use the amalgamation construction in Sect. 2.3 for it. Let us call $DSL_{Observer}$ to the Response Time DSL from Sect. 3.4, and let us consider the inclusion morphism from its parameter sub-DSL, DSL_{Par}. Given this inclusion morphism and a binding morphism B from DSL_{Par} to the Palladio DSL, $DSL_{Palladio}$, we can build its amalgamation as shown in Fig. 5. The result are morphisms $\hat{i}$ and $\widehat{B}$ to the DSL $\widehat{Palladio}$, which is the Palladio DSL extended with the response-time observer objects. Its metamodel is the Palladio metamodel enriched with the additional classes as indicated in the mappings, and the rules defining its behavior enriched with the observer objects.

The morphism B is just a mapping from elements in the parameter DSL into elements in the Palladio DSL. This is done by defining a correspondences model (see [12]). For example, for weaving the metamodel of response time with the metamodel of our Palladio implementation in e-Motions, the Request class is mapped to Token. Regarding rules, we basically need to map each rule in the

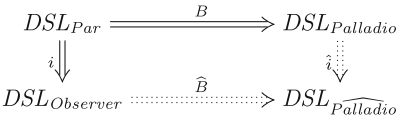


Fig. 5. Amalgamation in the category **GTS**.

source DSL to a rule in the target one. The mapping defined for the metamodel does most of the rest. The RequestArrives rule (Fig. 4(c)) is woven with the OpenWorkloadSpec rule of our Palladio system (Fig. 4(a)), that represents the arrival of a new Token in the system. Rule CreateRespTOb of the observer DSL is woven with an identity rule, triggering the creation of observer objects if they were not already created. Finally, rule CompletedRequest (Fig. 4(d)) is woven with the StopUsageModel rule, which just models the elimination of a token upon its arrival to a stop action.

Theorem 1 provides a checkable condition for verifying the conservative nature of an extension in our example, namely if B is a behavior-reflecting GTS morphism and i is an extension and behavior-protecting morphism, then $\widehat{i}$ is behavior-protecting as well.

Once the observers DSL are defined and checked, they can be used as many times as wished. To use them, we just need to provide the morphism binding the parameter DSL and the target system.

4 Related Work

Graph transformation systems (GTSs) were proposed as a formal specification technique for the rule-based specification of the dynamic behavior of systems [13]. Different approaches exist for modularization in the context of the graph-grammar formalism [6,14,42]. All of them have followed the tradition of modules inspired by the notion of algebraic specification module [17]. A module is thus typically considered as given by an export and an import interface, and an implementation body that realizes what is offered in the export interface, using the specification to be imported from other modules via the import interface. For example, Große-Rhode, Parisi-Presicce, and Simeoni introduce in [24] a notion of *module* for typed graph transformation systems, with interfaces and implementation bodies; they propose operations for union, composition, and refinement of modules. Other approaches to modularization of graph transformation systems include PROGRES Packages [44], GRACE Graph Transformation Units and Modules [33], and DIEGO Modules [46]. See [29] for a discussion on these proposals. For the kind of systems we deal with, the type of module we need is much simpler. For us, a module is just the specification of a system, a GTS, without import and export interfaces. Then, we build on GTS morphisms to compose these modules, and specifically we define parameterized GTSs.

We find different forms of GTS morphisms in the literature, taking one form or another depending on their concrete application. Thus, we find proposals

centered on refinements [23,24,28], views [21], and substitutability [20]. See [20] for a first attempt to a systematic comparison of the different proposals and notations. None of these notions fit our needs, and none of them coincide with our behavior-aware GTS morphisms.

As far as we know, parameterized GTSs and GTS morphisms, as we discuss them, have not been studied before. Heckel and Cherchago introduce parameterized GTSs in [27], but their notion has little to do with our parameterized GTSs. In their case, the parameter is a signature, intended to match service descriptions. They however use a double-pullback semantics, and have a notion of substitution morphism which is related to our behavior preserving morphism.

The way in which we think about composition of reusable DSL modules is related to work in aspect-oriented modeling (AOM). In particular, our ideas for expressing parameterized metamodels are based on the proposals in [4,32]. Most AOM approaches use syntactic notions to automate the establishment of mappings between different models to be composed, often focusing primarily on the structural parts of a model. While our mapping specifications are syntactic in nature, we focus on composition of behaviors and provide semantic guarantees. In this sense, our work is perhaps most closely related to the work on MATA [50] or semantic-based weaving of scenarios [31].

The idea of generic DSL has also been used in the context of model management by different authors. E.g., [8,41] use generic metamodel *concepts* as an intermediate, abstract metamodel over which model management specifications are defined, enabling the application of the operations thus defined to any metamodel satisfying the requirements imposed by the concept.

5 Conclusions

Our work was originally motivated by the specification of non-functional properties (NFPs), such as performance or throughput, in DSLs. We have been looking for ways in which to encapsulate the ability to specify non-functional properties into reusable DSL modules. Troya et al. used the concept of observers in [47,48] to model non-functional properties of systems described by GTSs in a way that could be analyzed by simulation. In [12,48], we have built on this work to allow the modular encapsulation of such observer definitions in a way that can be reused in different DSL specifications. We then formalized and generalized the composition operations needed in [11], were we provided a full formal framework of such language extensions.

In [35], we addressed the performance analysis problem by presenting a modular, model-based partial reimplementation of one well-known analysis framework—the Palladio Architecture Simulator. We have specified key DSLs from Palladio in e-Motions, describing the basic simulation semantics as a set of graph-transformation rules. Different properties to be analyzed have been encoded as separate, parameterized DSLs, independent of the definition of Palladio. We have then composed these DSLs with the base Palladio DSL to generate specific simulation environments. Models created in the Palladio IDE can be fed directly into our simulation environment for analysis.

We have demonstrated two main benefits of our approach: (1) The semantics of the simulation and the non-functional properties to be analyzed are made explicit in the respective DSL specifications, and (2) because of the compositional definition, it is easy to add definitions of new non-functional properties and their analyses. More importantly, our proposal provides a place were to experiment with new features and tailor solutions for specific problems at a very low development cost.

As future work, we plan to provide methods to check the preconditions of Theorem 1, and automatically checkable conditions that imply these, so that behavior protection of an extension can be checked effectively. This will enable the development of tooling to support the validation of language or transformation compositions. We also plan to study relaxations of our definitions so as to allow cases where there is a less than perfect match between the base DSL and the DSL to be woven in.

We plan to incorporate additional features to our definition of Palladio, as, for example, full resource models, and failures and reliability analysis. Indeed, we foresee generic definitions of selectable features, such as resource handling and deployment strategies, etc. We also plan to experiment with other NFPs, such as reliability or security, and to use our flexible setting for the analysis of dynamic systems, where components and resources are dynamically added to or removed from the system under study.

Acknowledgements. This work is an overview of work developed in collaboration with A. Moreno-Delgado, F. Orejas, J. Troya, A. Vallecillo, and S. Zschaler. I am grateful to all of them. This work is partially funded by Project TIN2011-23795 and by U. de Málaga, Campus de Excelencia Intl. Andalucía Tech.

References

1. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the Palladio component model. In: Proceedings of 6th International Workshop on Software and Performance (WOSP'07). ACM (2007)
2. Boehm, P., Fonio, H.-R., Habel, A.: Amalgamation of graph transformations with applications to synchronization. In: Ehrig, H., Floyd, C., Nivat, M., Thatcher, J. (eds.) TAPSOFT 1985. LNCS, vol. 185, pp. 267–283. Springer, Heidelberg (1985)
3. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
4. Clarke, S., Walker, R.J.: Generic aspect-oriented design with Theme/UML. In: Aspect-Oriented Software Development, pp. 425–458. Addison-Wesley (2005)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The category of typed graph grammars and its adjunctions with categories of derivations. In: Cuny, J., et al. [7], pp. 56–74
7. Cuny, J., Ehrig, H., Engels, G., Rozenberg, G. (eds.): Graph Grammars 1994. LNCS, vol. 1073. Springer, Heidelberg (1996)

8. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Softw. Syst. Model. **12**(3), 453–474 (2013)
9. de Lara, J., Vangheluwe, H.: Automating the transformation-based analysis of visual languages. Formal Asp. Comput. **22**(3–4), 297–326 (2010)
10. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), April 2006
11. Durán, F., Orejas, F., Zschaler, S.: Behaviour protection in modular rule-based system specifications. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 24–49. Springer, Heidelberg (2013)
12. Durán, F., Zschaler, S., Troya, J.: On the reusable specification of non-functional properties in DSLs. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 332–351. Springer, Heidelberg (2013)
13. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2005)
15. Ehrig, H., et al. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. II. World Scientific, Singapore (1999)
16. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004)
17. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. Springer, Heidelberg (1990)
18. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
19. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
20. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 38–63. Springer, Heidelberg (2005)
21. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A combined reference model- and view-based approach to system specification. Intl. J. Softw. Eng. Knowl. Eng. **7**(4), 457–477 (1997)
22. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
23. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Spatial and temporal refinement of typed graph transformation systems. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 553–561. Springer, Heidelberg (1998)
24. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Formal software specification with refinements and modules of typed graph transformation systems. J. Comput. Syst. Sci. **64**(2), 171–218 (2002)

25. Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
26. Happe, J., Koziolek, H., Reussner, R.: Facilitating performance predictions using software components. IEEE Softw. **28**(3), 27–33 (2011)
27. Heckel, R., Cherchago, A.: Structural and behavioural compatibility of graphical service specifications. J. Logic Algebraic Program. **70**(1), 15–33 (2007)
28. Heckel, R., Corradini, A., Ehrig, H., Löwe, M.: Horizontal and vertical structuring of typed graph transformation systems. Math. Struct. Comput. Sci. **6**(6), 613–648 (1996)
29. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of modularity concepts for graph transformation systems. In: Ehrig et al. [15], chap. 17, pp. 669–690
30. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Softw. Syst. Modell. **9**(3), 375–402 (2010)
31. Klein, J., Hélouët, L., Jézéquel, J.-M.: Semantic-based weaving of scenarios. In: Proceedings of 5th International Conference on Aspect-Oriented Software Development (AOSD'06). ACM (2006)
32. Klein, J., Kienzle, J.: Reusable aspect models. In: Proceedings of Aspect-Oriented Modeling Workshop (2007)
33. Kreowski, H., Kuske, S.: Graph transformation units and modules. In: Ehrig et al. [15], chap. 15, pp. 607–638
34. Lack, S., Sobociński, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
35. Moreno-Delgado, A., Durán, F., Zschaler, S., Troya, J.: Modular DSLs for flexible analysis: an e-Motions reimplementation of palladio. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 132–147. Springer, Heidelberg (2014)
36. Parisi-Presicce, F.: Transformations of graph grammars. In: Cuny et al. [7], pp. 428–442
37. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09), pp. 51–55. IEEE (2009)
38. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 174–190. Springer, Heidelberg (2010)
39. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 54–73. Springer, Heidelberg (2009)
40. Roldán, M., Durán, F.: Dynamic validation of OCL constraints with mOdCL. ECEASST, 44 (2011)
41. Rose, L.M., Guerra, E., de Lara, J., Etien, A., Kolovos, D.S., Paige, R.F.: Genericity for model management operations. Softw. Syst. Model. **12**(1), 201–219 (2013)
42. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. I. World Scientific, Singapore (1997)
43. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006)
44. Schürr, A., Winter, A., Zündorf, A.: The PROGRES-approach: language and environment. In: Ehrig et al. [15], chap. 13, pp. 487–550
45. Smith, C.U., Williams, L.G. (eds.): Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Object-Technology Series. Addison-Wesley, Boston (2002)

46. Taentzer, G., Schürr, A.: DIEGO, another step towards a module concept for graph transformation systems. Electron. Notes Theoret. Comput. Sci. **2**, 277–285 (1995)
47. Troya, J., Rivera, J.E., Vallecillo, A.: Simulating domain specific visual models by observation. In: Proceedings of Spring Simulation Multiconference (SpringSim'10), pp. 128:1–128:8. ACM (2010)
48. Troya, J., Vallecillo, A., Durán, F., Zschaler, S.: Model-driven performance analysis of rule-based domain specific visual models. Inf. Softw. Technol. **55**(1), 88–110 (2013)
49. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
50. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: a unified approach for composing UML aspect models based on graph transformation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 191–237. Springer, Heidelberg (2009)