UNIVERSIDAD
DE MÁLAGA

E.T.S.
INGENIERÍA
INFORMÁTICA

On the Conservative Composition of DSLs: The Palladio Case Study

Máster en Ingeniería del Software e Inteligencia Artificial
Universidad de Málaga

Realizado por
Antonio Manuel Moreno Delgado

Dirigido por
Francisco Javier Durán Muñoz

Málaga, Julio de 2014

A mis abuelos Manuel y Angelita.

"It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."

— *Albert Einstein*
On the Method of Theoretical Physics, 1933

Resumen

Estudios recientes revelan que las empresas que siguen la metodología de Diseño Software Basado en Modelos lo hacen usando lenguajes específicos desarrollados para sus dominios, en lugar de hacer uso de lenguajes de modelado de propósito general, como UML [49]. Sin embargo, el desarrollo de lenguajes de dominio específico (DSL, de su acrónimo en inglés *Domain-Specific Languages*) es económicamente viable solo si el proceso se realiza de manera eficiente. Técnicas de reusabilidad y composición del software permiten la reutilización de DSL durante el ciclo de vida de su desarrollo. Basados en la formalización y resultados de [14], en este trabajo presentamos potentes operaciones de composición de DSL, su implementación para DSL definidos utilizando e-Motions e ilustramos su uso con la definición del lenguaje Palladio.

Sistemas con requerimientos de tiempo real especificados en la herramienta e-Motions pueden componerse mediante la instanciación de DSL paramétricos. Concretamente, se ha definido en e-Motions uno de los DSL para el análisis de rendimiento más conocidos, el DSL Palladio. Su comportamiento operacional, especificado en e-Motions explícitamente con reglas de transformación de grafos, es compuesto con DSL paramétricos que especifican propiedades no funcionales. Dicha composición permite extender Palladio con el análisis de cualquier propiedad no funcional definida como un DSL. Además, el comportamiento del sistema instanciado, en este caso el DSL Palladio, es protegido, en el sentido de que su comportamiento no cambia, como se ha demostrado en [14]. Las principales ventajas de nuestro enfoque son: (i) el comportamiento operacional es específicado explícitamente por un conjunto de reglas de transformación de grafos en lugar de ser dado implícitamente en las transformaciones a otros formalismos, de esta forma los expertos del dominio pueden visualizar la especificación y razonar sobre su validez; y (ii) se establece cómo la especificación de Palladio en e-Motions puede ser compuesta y extendida con propiedades no funcionales modulares y genéricas. Este enfoque de modularización y composición puede ser sistemáticamente aplicado a otros dominios.

Palabras clave. Lenguajes de dominio específico, Palladio, propiedades no funcionales, composición, sistemas de transformación de grafos, e-Motions.

Abstract

Recent studies reveal that companies who successfully applied Model-Driven Engineering largely did so by creating or using languages specifically developed for their domain, rather than using general-purpose languages such as UML [49]. However, development of new domain-specific languages (DSLs) are only viable if they can be efficiently specified. Reusability and composition techniques enable the reutilization of DSLs in their development life-cycle. Based on the formalization and results presented in [14], in this work we present powerful composition operations of DSLs, their implementation for DLSs specified in e-Motions, and we illustrate its use by defining the Palladio DSL.

Given systems with time requirements specified in the e-Motions tool, we show how we may compose them by the instantiation of previously defined parametric DSLs. Specifically, we have defined in e-Motions one of the most well-known DSLs for reliability and performance analysis, the Palladio DSL. Its operational behavior, explicitly specified by graph-transformation rules, is composed with parametric DSLs defining non-functional properties. Such composition allows us to extend Palladio with new non-functional properties defined as DSLs. Moreover, the behavior of the system being instantiated, i.e., the Palladio DSL, is protected, in the sense that they do not change its behavior, as it has been proven in [14]. Two main advantages of our approach are demonstrated with the Palladio case study: (i) operational behavior is explicitly specified by a set of graph-transformation rules, instead of being implicit in the transformations to other formalisms, thus allowing to domain experts to visualize and validate them; and (ii) we are able to compose the e-Motions specification of Palladio with new generic and modular non-functional properties.

Keywords. Domain-specific languages, Palladio, non-functional properties, composition, graph transformation systems, e-Motions.

1 Introduction

In the last ten years, Model-Driven Engineering (MDE) [42, 4] has attracted the attention of many researchers, standards developers and industry [25]. In MDE, models are first-class citizens, being used to specify, simulate and analyze systems in early stages of the software life-cycle and being able to generate final code [24]. On the other hand, domain specific languages (DSLs) [12] have become more important since they are a key ingredient in the specification and definition of such models.

In contrast to general purpose languages as Java or C++, DSLs are targeted to very narrow domains. Thus, such languages make a perfect balance between abstraction level and system specification, i.e., they allow accurate and precise definitions of systems without loosing abstraction level. DSLs are tailored to specific domain problems, and they use concepts of such domains. Thus, DSLs may be easily used by domain experts, who may lack computer skills. These advantages make DSLs a field of interest.

The closer to a specific domain a DSL is, the more effective it is. Although a plethora of DSLs has appeared in the last years, a definition of a new DSL is only viable if it can be developed efficiently. Consequently, DSLs are often defined in some standard formalism, as MOF, which can be handled by generic tools (e.g., Eclipse Modeling Framework), consisting of model transformation, editor and composition tools.

Current available MDE tools have focused on *syntax*. However, in addition to the syntax, DSLs' operational behavior must be defined in order to be able to simulate and analyze the defined models. Different formalisms have been proposed for the specification of behavior, as UML behavioral models [20], abstract state machines [13, 5], Kermeta [35], or in-place model transformation [32, 37]. Between all these approaches, we find the use of in-place model transformation particularly powerful, because, in addition to its expressiveness, it facilitates its integration with the rest of the MDE environment.

Although syntax is a well understood problem and we have reasonably good knowledge of how to modularize it, the modularization of the operational behavior is an as yet unsolved issue. In this Master's thesis, we present and extend an approach to reuse and compose DSLs, whose behavior specifications have been defined using in-place transformation rules.

Durán et al. [15, 14, 47] have built on graph transformation system (GTS) morphisms to define composition operations on DSLs. Specifically, they 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. Moreover, under given circumstances, the amalgamation of the morphisms is conservative, in the sense that the protection of behavior when DSLs get instantiated is guaranteed. Although these results are language-independent, and they are presented for GTSs and adhesive HLR systems [18, 29] in general, in this work we show an implementation of this approach using the e-Motions tool [37, 38]. e-Motions is a tool where to define DSLs with time features.

Finally, we present a real case study, as it is the e-Motions implementation of the Palladio DSL. Palladio is a DSL intended for the description of component-based systems. In Palladio, one may specify relevant information to analyze performance and reliability of such defined systems. These models may be defined by different domain experts, and be transformed to be analyzed by simulators or in other formalism. However, these transformations present two drawbacks: (i) the semantics of non-functional properties (NFPs) analyzed by Palladio is encoded into such tranformations, and therefore it is difficult for users to reason about properties being analyzed; and (ii) it is very difficult to maintain or extend the system with new properties. To cope with that, we present an e-Motions definition of Palladio, and we enrich such Palladio definition with modular and parameterized NFPs.

This Master's thesis has been developed in the context of the *Atenea* group [1], which is a subgroup of GISUM (University of Málaga). Its main research line is the modeling and analysis of distributed information systems and one of the most prominent developments in the group is the e-Motions language and tool. e-Motions was the result of the dissertation [36] of J.E. Rivera supervised by Durán and Vallecillo. Based on e-Motions, J. Troya proposed a way of monitoring and analyzing NFPs using observers [44]. However, observers remained tailored to the system at hand and they needed to be specified over and over. First steps towards the modular specification of NFPs were presented in [47, 15], and its formalization was presented in [14]. I started to work with the Atenea group on 2013, under the supervision of Francisco Durán. To date, I have been involved in several works. As part of this Master's thesis, I have participated in the implementation and evaluation of this proposal, which is presented in this document. Part of this work has already been published. Specifically, the implementation of a case study was presented in [34], and the e-Motions specification of Palladio was published in [33]. Both publications, being published in peer-reviewed conferences, which support this Master's thesis, are:

– Moreno-Delgado, A., Durán, F., Zschaler, S., Troya, J.: "Modular DSLs for flexible analysis: An e-Motions reimplementation of Palladio." *10th European Conference on Modelling Foundations and Applications, ECMFA 2014.* Lecture Notes in Computer Science, vol. 8569. Springer (2014).

– Moreno-Delgado, A., Troya, J., Durán, F., Vallecillo, A.: "On the Modular Specification of NFPs: A Case Study." *XVIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD).* pp. 302–316 (2013), available at `http://www.sistedes.es/ficheros/actas-conferencias/JISBD/2013.pdf`.

Outline. The remaining of this Master's thesis is structured as follows. Section 2 presents the e-Motions system, showing its main features with the production line system case study. Section 3 presents how non-functional properties of systems defined in e-Motions can be analyzed. It also focuses in the modular definition of such properties and illustrates its use with two examples. Section 4 presents some formal definitions on which the modular composition of observers here presented is based. Section 5 describes an implementation of such composition operations. Section 6 presents the Palladio Architecture Simulator, and shows its main characteristics, with two basic examples. Section 7 explains the specification of Palladio in e-Motions, and Section 8 presents how the specification of Palladio is enriched with modular observers. Section 9 surveys the most important works which connect with our work. Finally, we conclude and set up some future works in Section 10.

2 e-Motions

e-Motions [37, 38] is an Eclipse-based tool for the specification, formal analysis and simulation of real time systems. e-Motions provides a way to graphically specify the abstract and concrete syntaxes, and the dynamic behavior of DSLs.

The abstract syntax of a DSL is specified in e-Motions as an Ecore metamodel, which defines the main concepts of the language—and the relations among them. The structure of all possible states is also constrained by the abstract syntax. The concrete syntax of a DLS is given by a GCS (Graphical Concrete Syntax) model, which attaches an image to each concept previously defined in the abstract syntax.

Finally, the dynamic behavior of a DSL is specified by a set of graphical rewrite rules, using its concrete syntax, and thus making this task very intuitive. Each rewrite rule in e-Motions represents an *action* in the system and is of the form: $[\mathrm{NAC}]^* \times \mathrm{LHS} \rightarrow \mathrm{RHS}$, where LHS (Left Hand-Side), RHS (Right Hand-Side) and NAC (Negative Application Condition) are model patterns which represent certain (sub-) states of the system being defined. The LHS and the—possibly empty—set of NACs stand for the precondition of the rule, whereas the RHS represents the post-condition or the effect of the action. A LHS may also have positive conditions, expressed in OCL, as any attribute assignment or modification in the RHS. A rule can be triggered if a match of the LHS is found in the state, holding its positive conditions (if any), and none of its NACs occur. If several matches are found, one of them is non-deterministically chosen and applied, given a new state where (i) those objects in the LHS that are not found in the RHS are deleted, (ii) those objects in the RHS that are not found in the LHS are created, and (iii) those objects which are present in the LHS and the RHS are updated—or remain changeless.

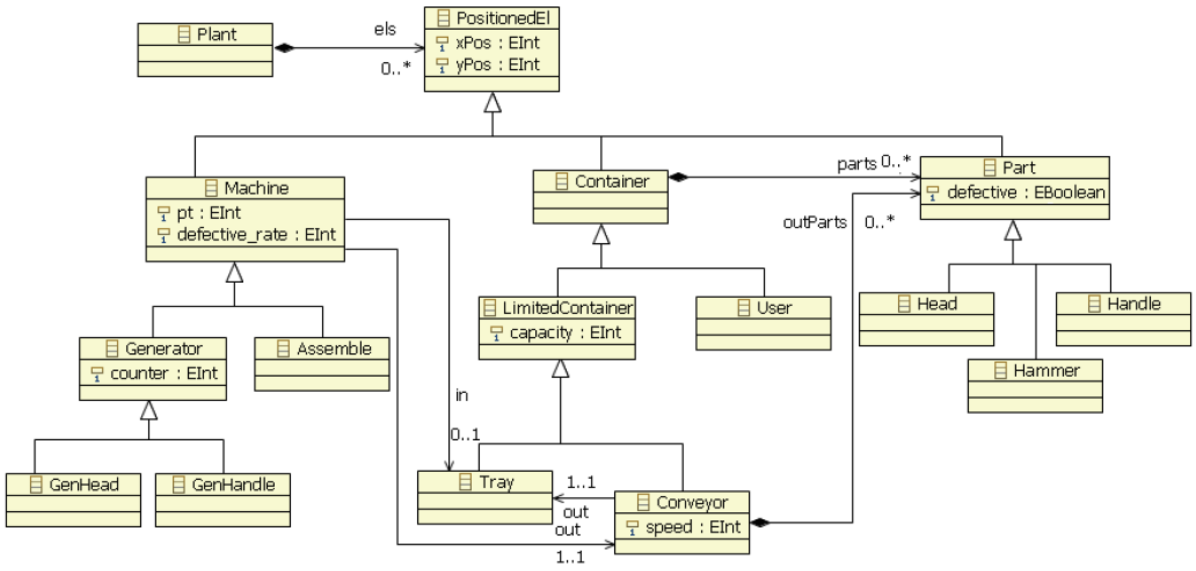
e-Motions supports a model of time with features like duration, periodicity, etc. From a DSL definition, e-Motions generates an executable Maude [7, 8] specification which can be used for simulation and analysis. Moreover, the Maude formal environment may also be used, as its model checker or its reachability analysis tool.

Atomic and *ongoing* rules can be used in e-Motions to define time-dependant behaviors. Atomic rules represent atomic actions with a duration specified by an interval of time. Atomic rules with duration zero (specified with an interval `[0, 0]`) are called *instantaneous*. Ongoing rules represent actions that progress over time while the rule's preconditions hold. Both atomic and ongoing rules can be scheduled.

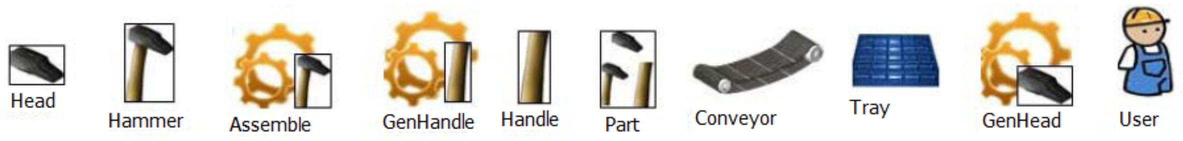
2.1 The Product Line System DSL

We present a motivating example to show how a system is defined in e-Motions. Specifically, we present the Product Line system (PLS) DSL, which is borrowed from [14, 45]. This DSL models production line systems intended for the production of hammers out of hammer heads and handles, which are generated in respective machines, and transported along the production line via conveyors, and temporarily stored in trays.

Fig. 1 (a) shows the metamodel of a DSL for specifying such PLSs. As usual in MDE-based DSMLs, this metamodel defines all the concepts of the language and their interconnections. This metamodel serves as abstract syntax for our DSL. Concrete syntax is specified as a GCS model, in which an image is associated to each concept. Fig. 1 (b) shows the GCS model for the PLS DSL (image names match with concept names to be attached to).



(a) PLS metamodel.



(b) PLS concrete syntax.

Fig. 1: PLS syntax specification.

The behavior of the PLS DSL is specified by a number of transformation rules, one for each of the actions that may occur in the system. Fig. 2 shows the e-Motions atomic rule specifying the assemble process of a hammer.[1] Its LHS represents the sub-states on which the action can be performed, i.e. an `Assemble` object with a `Conveyor` as input and a `Tray` as output, and a `Head` and a `Handle` in the `Conveyor`. If a matching is found in the system, the `NAC1` is checked. `NAC1` will hold if there is no action `Assemble` with the object `a` of type `Assemble` involved and *unfinished*, i.e. not concurrent assembles are allowed. If such a matching is found, the `Assemble` rule is applied, removing the objects `he` and `ha`, and creating the object `ham` with its attributes properly set. The rule is applied in the time specified by the local variable `prodTime`. Notice how rules stand for actions in the system, but also actions may be represented as objects within the state. Thus, for example, exceptions may be modeled by removing actions currently being executed in the system [37].

Finally, a possible initial state of the system is the one presented in Fig. 3. Basically, it is a model which respects the structure defined in its metamodel, thus holding the conformance relation. Moreover, all attributes are correctly set in each object.

[1]The full example, tutorials and the e-Motions system are available at `http://atenea.lcc.uma.es/e-Motions`.

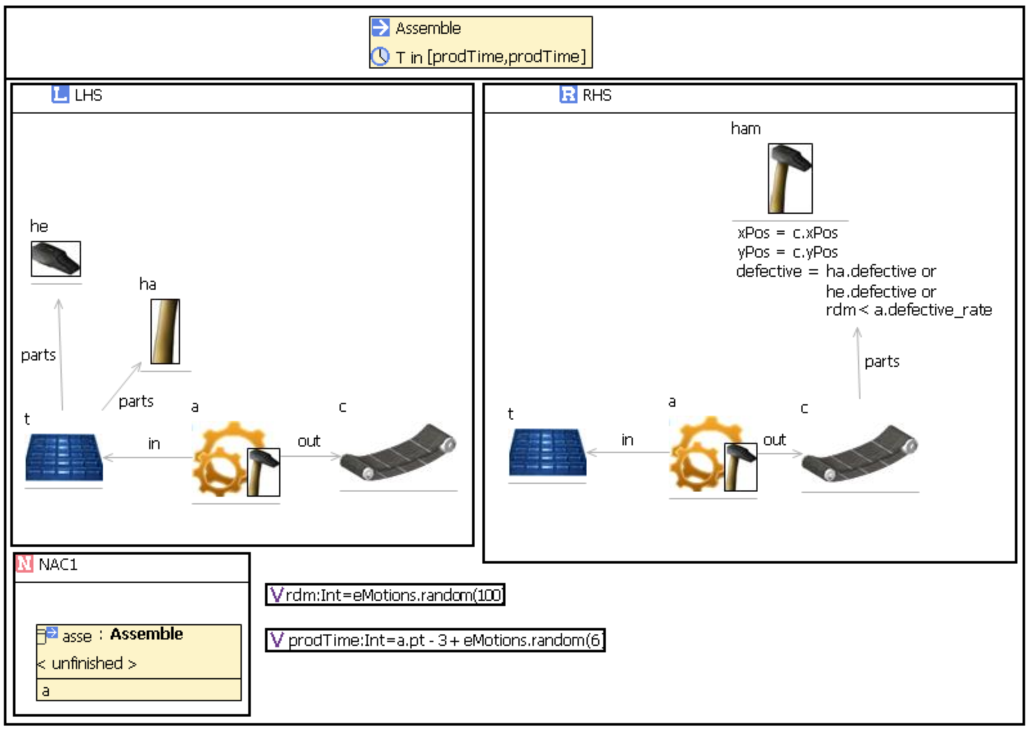


Fig. 2: e-Motions rewrite rule: `Assemble`.

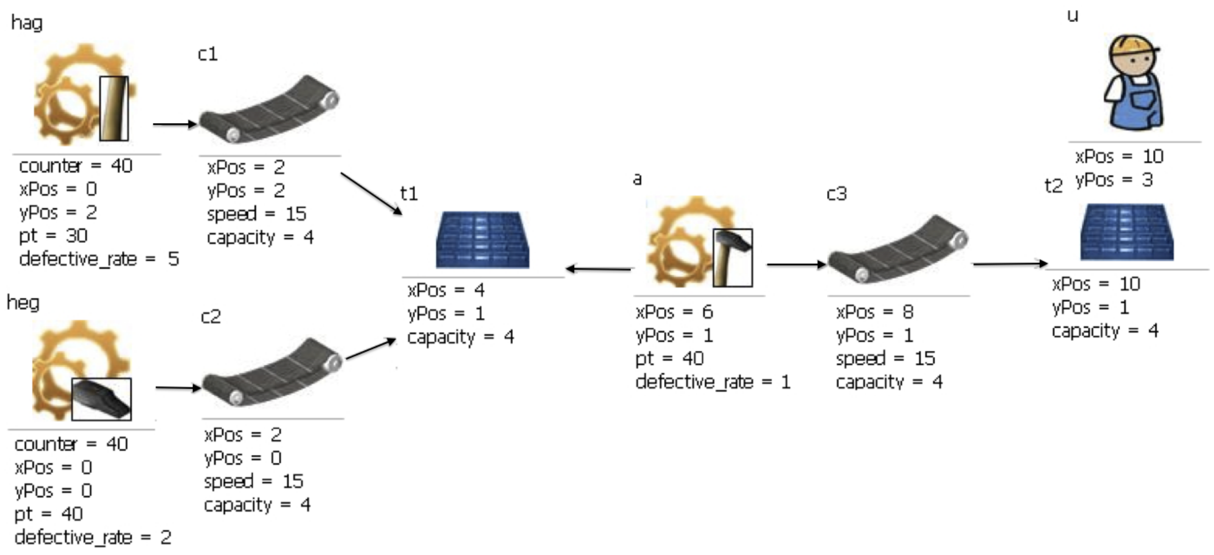


Fig. 3: An initial state for the PLS DSL.

3 Non-Functional Properties by Observation

Troya et al. proposed in [45] an approach for the specification and monitoring of non-functional properties (NFPs) using *observers*. Observers are objects with which we extend e-Motions definitions of DSLs. These observer objects are defined as DSLs, with their abstract and concrete syntaxes and their behavior. In [45, 46], observers are then added to those rewrite rules in which some NFP has to be monitored. During the simulation of any system, each observer object collects the information needed to perform the expected analysis. With the flexibility that DSLs offer, any NFP we wish may be defined, as mean and maximum cycle time, response time, busy and idle cycles, throughput, mean operation number per time units, etc. Additionally, as behavior of the NFP is explicitly defined by visual rewrite rules, domain experts may reason about such properties.

In [15, 47], it has been explored how to define modular and generic observer DSLs. This way, observers do not have to be tailored to every system over and over. Instead, observers are defined as parametric DSLs (i.e. a parametric syntax and a parametric behavior) which can afterwards be woven and merged with different systems. Given a library of generically defined observers, one could choose the one needed at each case, and add it to the system to be analyzed with the appropriate instantiation. A case study following this approach was presented in [34].

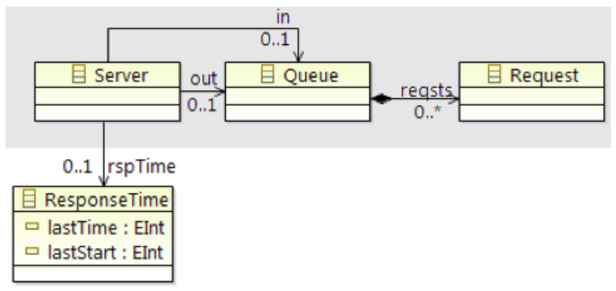
Given a graph grammar formalization of DSLs, Durán et al. [14] have built on graph transformation system (GTS) morphisms to define composition operations on DSLs. Specifically, they have defined parameterized GTSs, that is, GTSs which have other GTSs as parameters. The instantiation of such parameterized GTSs is then provided by an amalgamation construction. Specifically, they have studied 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. They have shown that under some reasonable conditions, the observers weaving and merging into a system is *conservative*, meaning that the observers added *do not change the behavior of the system.*

3.1 The *Response Time* generic observer

Let us consider a generic DSL for monitoring the *response time*. 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 [34]. 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 of a request.

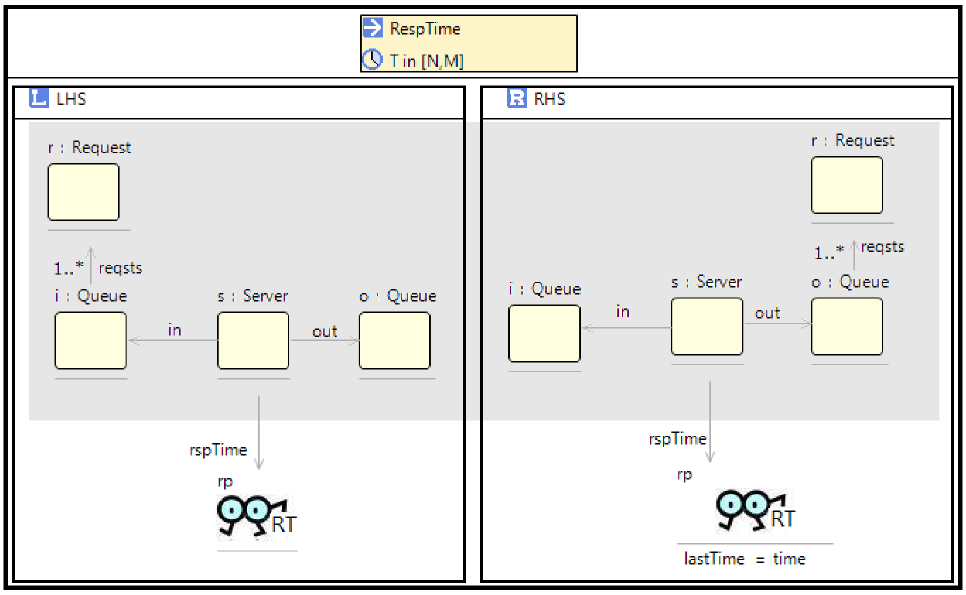
A generic DSL achieving this is shown in Fig. 4. Its abstract syntax (the metamodel in Fig. 4 (a)) contains three generic and one concrete classes—generic classes are shown with a shaded background. `System`, `Server` and `Request` are parameter classes to be instantiated by specific classes, as explained below. `Server`s may have a `Queue` as input or output (references `in` and `out`). These `Queue`s, in turn, contains `Request`s. The concrete class `ResponseTime` (whose concrete syntax is depicted in Fig. 4 (b)) represents the observer for measuring the response time.

Behavior is defined by rewrite rules, as the one presented in Fig. 4 (c). Square objects stand for classes without concrete syntax, as they are parametric objects. This rule is defined as general as possible, notice that the cardinality of the `reqsts` link is `1..*`. In non-parametric rules, cardinality of links is `1`, and for multiple cardinality, several links



(a) Generic Response Time metamodel.

(b) Concrete syntax.



(c) Response Time rule specification.

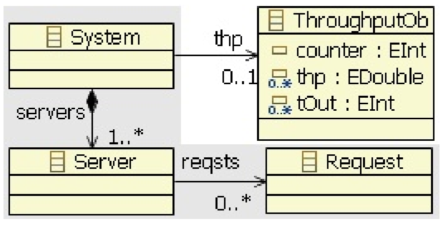
Fig. 4: Generic Response Time specification.

are defined. The rule heading (box on the rule) shows its identifier, `RespTime`, and the time `T` that the action takes, in this case some non-deterministic value in the range `[N, M]`, for some values `N` and `M` that will become later instantiated. As this rule, other rules has to be considered, as the one adding a `ResponseTime` object to each `Server`.

3.2 The *Throughput* generic observer

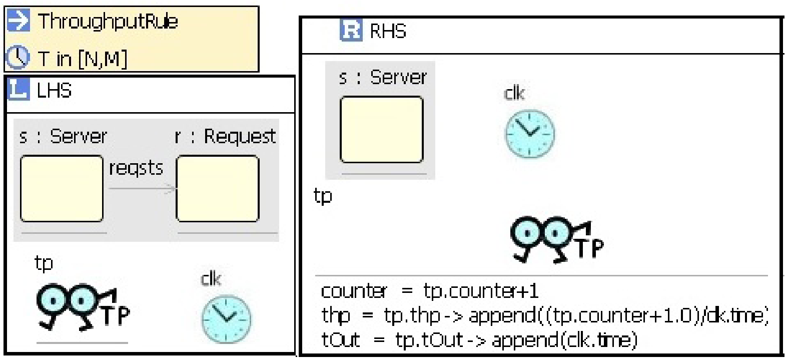
In communication networks, *throughput* is defined as the average rate of message delivery over a communication channel. However, the notion has also been used in other disciplines, acquiring a more general meaning. We can define throughput as the average rate of *requests* flowing through a system. Thus, the same generic notion allows us to measure the number of information packages being delivered through a network or the number of hammers being assembled in a production line.

Given this more general definition, and given the description of a system, to measure throughput we basically need to be able to count the number of items delivered or produced, and calculate its quotient with time. We define a `ThroughputOb` observer class



(a) Abstract syntax.

(b) Concrete syntax.



(c) Behavior.

Fig. 5: Throughput observer DSL definition.

with attributes `counter` and `thp` keeping these values. The metamodel for the DSL of `ThroughputOb` observers may be the one depicted in Fig. 5 (a). `ThroughputOb` observer objects will basically count instances of some *generic* class `Request`, which will later be instantiated to hammers or in the case of Palladio, to tokens.

We describe its behavior by the single rule in Fig. 5 (c). This rule represents the way in which the values of its `counter` and `thp` attributes are to be updated. This rule is intended to be interpreted as generic: when a request disappears, the `ThroughputOb` observer gets updated. Given a model in which there is a `Server` object `s` with a pending `Request` object `r`, plus a `ThroughputOb` object `tp`, and the system clock `clk`—its LHS—it can evolve to a system in which the request vanishes, and the observer object gets updated—its RHS. The `clk` object is an instance of the predefined `Clock` class, which represents the pass of time in the system, and whose `time` attribute keeps the time of the system since its start-up (see [38] for a detailed presentation of the modeling of time in *e-Motions*). Note that this definition of throughput is rather naive, since the `thp` attribute only changes when requests are consumed. We have used this definition here to simplify presentation.

4 Formal Framework

Once a system has been specified with a DSL, as it is the case of the PLS, properties to be monitored and analyzed may be chosen from a library containing parametric DSLs. In order to introduce observers in our specifications in e-Motions, we need to weave both the metamodel and the behaviour specifications of a specific system and the generic observer DSL. In other words, the parametric components of the observers DSLs get instantiated with specific components. To perform the weaving, bindings of classes, references, rules and object must be given.

As it has been mentioned above, the binding and weaving process has been formalized by Durán et al. in [14]. We present here some basic definitions and the intuition behind them.

Graph transformation [40] is a formal way of expressing graph manipulation based on rules. In graph-based modeling (and metamodeling), graphs are used to define the static structures, such as class and object ones, which represent visual alphabets and sentences over them. Durán et al. have formalized parameterized DSLs and their instantiation using the typed graph transformation approach, specifically the Double Pushout (DPO) algebraic approach, with positive and negative application conditions [17]. The formalization is carried out for weak adhesive high-level replacement (HLR) categories (see [16]).

Adhesive and (weak) adhesive HLR categories comes together with a collection of general semantic techniques, as general results of the Local Church-Rosser, or the Parallelism and Concurrency Theorem. Any category which is proved an adhesive HLR category, comes automatically with that results, as it is the case of category typed attributed graphs, which was proven to be an adhesive HLR category in [17].

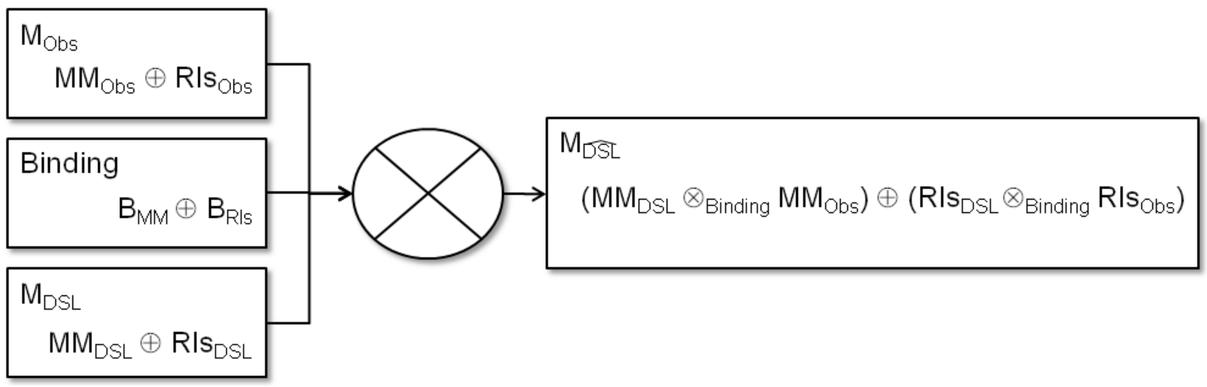

M_{Obs}
$MM_{Obs} \oplus Rls_{Obs}$
Binding
$B_{MM} \oplus B_{Rls}$
M_{DSL}
$MM_{DSL} \oplus Rls_{DSL}$
$M_{\widehat{DSL}}$
$(MM_{DSL} \otimes_{Binding} MM_{Obs}) \oplus (Rls_{DSL} \otimes_{Binding} Rls_{Obs})$

Fig. 6: Architecture of the formal framework.

First, we abstract away from the concrete representation of languages in e-Motions. Instead, we formally represent the key elements of which such languages and models consist and the functions which are used to manipulate them. Fig. 6 provides a graphical overview of the formal framework we are proposing, which consists of five parts:

1. M_{DSL}: The specification of a DSL without any notion of non-functional properties.

2. M_{Obs}: The specification of a generic DSL modeling a non-functional property.

3. *Binding*: An artifact expressing how the parameters of M_{Obs} should be instantiated with concepts of M_{DSL}.

4. $\otimes$: A function that performs the weaving.

5. $M_{\widehat{DSL}}$: A DSL resulting from the combination of the specification of a system and the non-functional properties observers defined in M_{Obs}.

4.1 Models involved and their relationships

Following the algebraic graph transformation approach, a DSL can be seen as a typed graph grammar. A *typed graph transformation system* $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P. A *typed graph grammar* $GG = (GTS, S)$ consists of a typed graph transformation system GTS and a typed start graph S. A language is then defined by the set of graphs reachable from S using the transformation rules P.

Definition 1. (**From [15], DSL**) The specification M_X of a DSL X is given by a metamodel MM_X, representing the structural concepts of the language, and a set of transformation rules Rls_X, defining its behavioral semantics.

The languages M_{DSL} and $M_{\widehat{DSL}}$ are DSL specifications. M_{Obs} is, essentially, also a normal DSL specification. Notice that we assume a single observer model M_{Obs} for each non-functional property. If we needed several of these properties, we could consider M_{Obs} to be the combination of the specifications of these non-functional properties, or we could iterate the process by instantiating $M_{\widehat{DSL}}$ once obtained with a second observers model $M_{Obs'}$ producing a resulting specification $M_{\widehat{\widehat{DSL}}}$, which could again be instantiated by another observers model $M_{Obs''}$ and so on.

Although M_{Obs} is, essentially, a normal DSL specification, it is however parameterized and we distinguish a parameter sub-model M_{Par} which specifies just enough information about real systems to define the semantics of the non-funtional property. This parameter sub-model M_{Par} is a sub model of M_{Obs} in the sense that M_{Par} is a subgraph of M_{Obs}, and each transformation rule in Rls_{Par} is a sub-rule of a rule in Rls_{Obs}.

This notion of sub-model and that of binding are captured by the general notion of DSL morphism, which can be defined as follows.

Definition 2. (**From [15], DSL Morphism**) Given DSL specifications M_A and M_B, a *DSL morphism* $M_A \to M_B$ is a pair (δ, ω) where δ is a metamodel morphism $MM_A \to MM_B$, that is, a mapping (or function) in which each class, attribute and association in the metamodel MM_A is mapped, respectively, to a class, attribute and association in MM_B such that

1. class maps in δ must be compatible with the inheritance relation, that is, if class C inherits from class D in MM_A, then $\delta(C)$ must inherit from class $\delta(D)$ in MM_B;
2. class maps and association maps in δ must be consistent, that is, the images of the extremes of an association K in MM_A must lead to classes associated by the association $\delta(K)$;
3. attribute maps and class maps in δ must be consistent, that is, given an attribute a of a class C, its image $\delta(a)$ must be an attribute of the class $\delta(C)$;

and where ω is a set of transformation rules such that for each rule $r_1 \in Rls_A$ there is a transformation rule $\sigma : r_1 \to r_2$ for some $r_2 \in Rls_B$.

The role of the parameter model M_{Par} is useful for establishing the way in which the DSL's metamodel and rules are to be modified. The binding DSL morphism is key for it, since it says how the transformations are to be applied. An inclusion morphism $(\iota, \omega) : M_{Par} \hookrightarrow M_{Obs}$ can be seen as a transformation rule, with the binding indicating how such rule must be applied to modify the DSL system being instrumentalised. Specifically, the metamodel morphism $\iota : MM_{Par} \hookrightarrow MM_{Obs}$ indicates how the metamodel MM_{DSL} must be extended, and the family of higher-order transformations (HOT) rules $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in ω indicate how the rules in Rls_{DSL} must be modified.

A transformation rule like the one in Fig. 4 (c) is interpreted as a rule transformation where the parameter part (the shadowed sub-rule) is its left-hand side, and the entire rule its right-hand side. Since M_{Par} is not intended for DSL specification, but only as constraints in the different transformations involved, i.e., for matching, we can enrich its expressiveness, for example, by allowing associations with multiplicity $1..n$ as in Fig. 4 (c). The left-hand side of this HOT rule can in this way match rules whose queues have 1, 2, or any number of request objects associated. Notice that this rule is woven with the `Assemble` rule in Fig. 2, which has a head and a handle associated to its `in` tray. It could be seen as the inclusion and the binding happening on specific submodels, those defined by the concrete match. We do not consider this additional flexibility in the formalization below. Notice however that the matches induce corresponding rules for which the formalization does work.

4.2 Model Weaving

M_{Obs} and M_{DSL} are woven to produce a combined DSL, $M_{\widehat{DSL}}$. This weaving is encoded as a function $\otimes : M_{Obs} \times M_{DSL} \times Binding \rightarrow M_{\widehat{DSL}}$, which is graphically depicted in Fig. 6. As indicated above, *Binding* is a DSL morphism, which expresses how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages. Intuitively, $\otimes$ works in two stages:

1. *Binding stage.* In this stage, *Binding* is used to produce an instantiated version of M_{Obs} (and its parameter sub-model M_{Par}), $M_{Obs'} = (MM_{Obs'}, Rls_{Obs'})$, which is the result of replacing each parameter element $p \in MM_{Par}$ by a corresponding element from MM_{DSL} in M_{Obs} in accordance with *Binding*. The resulting $MM_{Obs'}$ is used to construct the output metamodel $MM_{\widehat{DSL}} = MM_{DSL} \uplus_{Binding} MM_{Obs'}$. The operator $\uplus_{Binding}$ stands for disjoint union, where elements related by *Binding* are identified and the rest are distinguished. Each rule $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$ in $Rls_{Obs'}$ is the result of a similar replacement of a rule $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in Rls_{Obs}.

2. *Transformation stage.* In this stage, $Rls_{Obs'}$ is used to transform Rls_{DSL}. For each inclusion morphism $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$, the corresponding rule $r \in Rls_{DSL}$ is identified and transformed according to σ'_i. This step produces $Rls_{\widehat{DSL}}$.

Note that the rules and appropriate matches to apply the HOT rules should be guided by *Binding*. Although there might be cases in which we can systematically apply the HOT rules on the rules in Rls_{DSL}, in general this is not the case. Note that each HOT rule defined by a rule in Rls_{Obs} may be applicable to different rules in Rls_{DSL}, and for each of them there might be more than one match. Although there might be many cases in which a partial binding might be enough, we however assume that the binding is complete.

$$\begin{tikzcd}
\boxed{\begin{array}{l} M_{Par} \\ MM_{Par} \oplus Rls_{Par} \end{array}} \arrow[r, Rightarrow, "\substack{\mathit{Binding} \\ B_{MM} \oplus B_{Rls}}"] \arrow[d, Rightarrow] & \boxed{\begin{array}{l} M_{DSL} \\ MM_{DSL} \oplus Rls_{DSL} \end{array}} \arrow[d, dotted] \\
\boxed{\begin{array}{l} M_{Obs} \\ MM_{Obs} \oplus Rls_{Obs} \end{array}} \arrow[r, dotted] & \boxed{\begin{array}{l} M_{\widehat{DSL}} \\ (MM_{DSL} \otimes MM_{Obs}) \oplus (Rls_{DSL} \otimes Rls_{Obs}) \end{array}}
\end{tikzcd}$$

Fig. 7: Amalgamation in the category of DSLs and DSL morphisms.

The semantics of the weaving operationis provided by an amalgamation in the category of DSLs and DSLs morphisms [14]. Fig. 7 shows the amalgamation of an inclusion morphism between the model of an observer DSL, M_{Obs}, and its parameter sub-model M_{Par}, and the binding morphism from M_{Par} to the DSL of the system at hand, M_{DSL} (e.g. the prouction line system DSL in our example). The amalgamation object $M_{\widehat{DSL}}$ is obtained by the construction of the amalgamation of the corresponding metamodel morphisms and the amalgamation of the rules describing the behavior of the different DSLs. In [14], Durán et al. have shown that under very reasonable conditions, this amalgamation is conservative, that is, the behavior of the system enriched with observers can be protected.

5 Binding and weaving of Observers

The formalization presented in Section 4 has been implemented using ATL [26]. Following the proposal presented in 4.2, the weaving process has been split into two ATL transformations: one for weaving the metamodels, MM_{DSL} and MM_{Obs}, and another one for weaving the behavioral rules, Rls_{DSL} and Rls_{Obs}. Fig. 9 illustrates the architecture and flow of such two ATL transformations.

For the remainder of this section, let us clarify that by *binding* we mean the relations established between two models. As for *correspondence(s)* and *matching(s)*, we use them indistinctly when we refer to one or more specific relations among the concepts in both models (either DSL and observer metamodels or DSL and observer behavioural rules).

The binding between M_{DSL} and M_{Obs} is given by a model that conforms to the correspondences metamodel shown in Fig 8. Thus, both bindings, between metamodels and between behavioural rules, are given in the same model. For the binding between metamodels (Fig. 1 (a) and 4 (a) in our example), we have the classes `MMMatching`, `ClassMatching` and `RefMatching` that specify it. We will have one object of type `MMMatching` for each pair of metamodels that we want to weave. In our example, we have one object of this type, and its attributes contain the names of the metamodels to weave. Objects of type `MMMatching` contain as many `classes` (objects of type `ClassMatching`) as there are correspondences between classes in both metamodels. Each object of type `ClassMatching` stores the names of the classes in both metamodels that correspond. Regarding the objects of type `RefMatching`, contained in the `refs` reference from `MMMatching`, they store the matchings between references in both metamodels. Attributes `obClassName` and `DSLClassName` keep the names of the source classes, while `obRefName` and `DSLRefName` contain the names of the references.

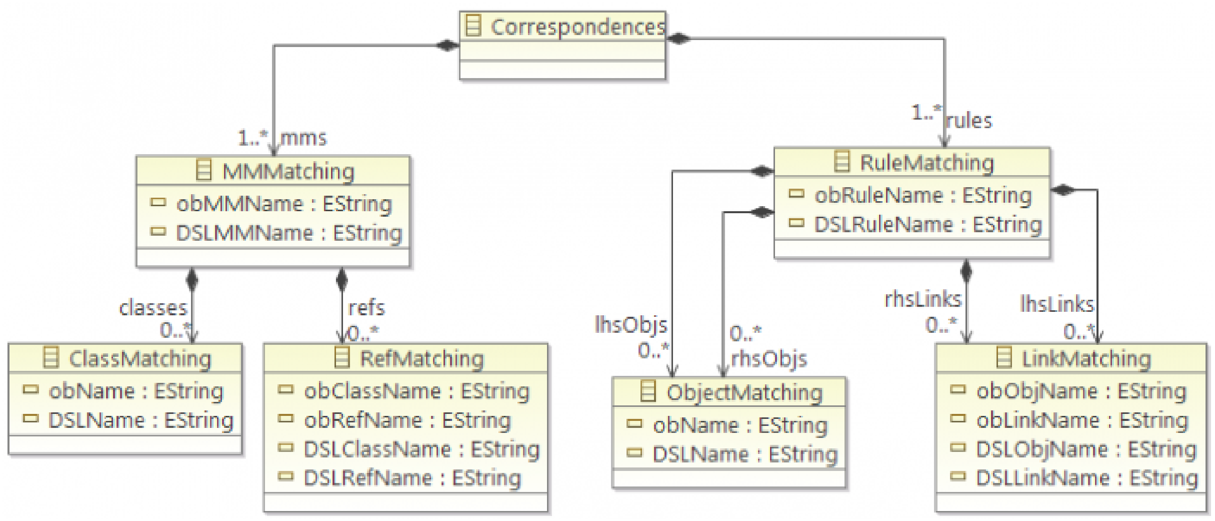


Fig. 8: Correspondences metamodel.

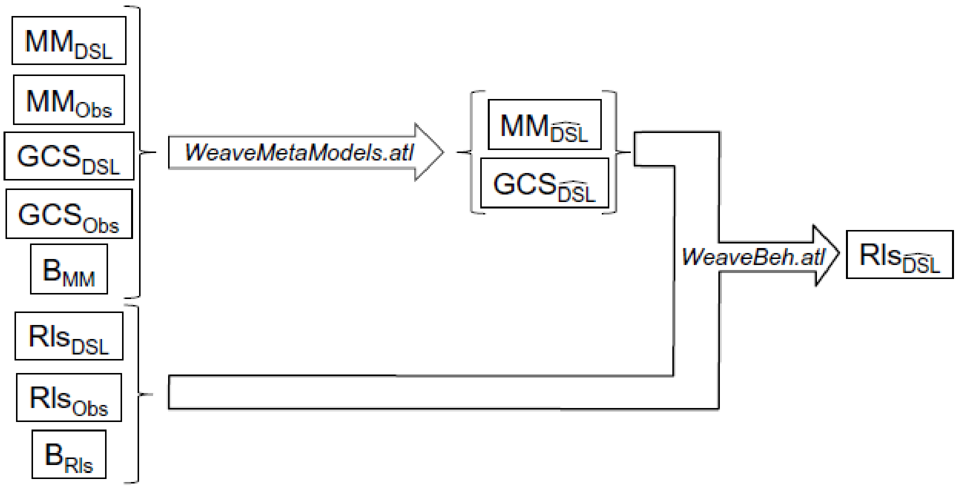


Fig. 9: Architecture of the ATL transformations to weave observers.

Regarding the binding between rules (Rls_{DSL} and Rls_{Obs}), there is an object of type `RuleMatching` for each pair of rules to weave, so in our example there is only one. It contains the names of both rules (`Assemble` and `RespTime`). Objects of types `ObjectMatching` and `LinkMatching` contain the correspondences between objects and links, respectively, in the rules. Specifically, our correspondence models differentiate between the bindings established between left- and right-hand side in rules, as we describe later. In our behavioral rules described within e-Motions, which conform to the `Behavior` metamodel (presented in [37]), the objects representing instances of classes are of type `Object` and they are identified by their `id` attribute, and the links between them are of type `Link`, identified by their name, input and output objects. Similar to the binding between metamodels, objects of type `ObjectMatching` contain the identifier of the objects matching, and instances of `LinkMatching` store information about matchings between links (they store the identifier of the source classes of the links as well as the name of the links).

5.1 Weaving of the *Response time* generic observer

In this section, we present an example of binding and weaving of an observer with a specific system. Specifically, we enrich the product line system DSL presented in Section 2.1 with the response time generic observer presented in Section 3.1.

As it has been described during this section,we firstly weave both metamodels MM_{DSL} and MM_{Obs} binding the parameter MM_{Par} with the system. Bindings are given by means of a correspondences model (see Section 5). Graphically, such binding is illustrated in the top part of the Fig. 10 in which:

– `Server` is bound to `Assemble`, as we are interested in measuring response time of this particular machine;

– `Queue` is bound to `LimitedContainer`, as the `Assemble` machine is to be connected to an arbitrary `LimitedContainer` for queuing incoming and outgoing parts;

– `Request` is bound to `Part`.

Although associations bindings are not depicted in Fig. 10, they also have to be matched:

– `in` and `out` associations from `Server` to `Queue` are bound to the corresponding `in` and `out` associations from `Machine` to `Tray` and `Conveyor`, respectively;

– `reqsts` association from `Queue` to `Request` is bound to `parts` association from `Container` to `Part`.

Parametric part of response time definition MM_{Par} stands for the precondition of the weaving.

Note how the formal definition of the amalgamation depicted in Fig. 7 looks very similar to Fig. 10, which shows a practical example. The resulting metamodel $MM_{\widehat{DSL}}$ has a `ResponseTime` observer associated to `Assemble`, stating that `Assemble` machine will be monitored by this observer. However, weaving the metamodels is just part of the story, and the behavior of DSLs Rls_{DLS} also have to be enriched with observer rules Rls_{Obs}. Each of the rules must be appropriately bound. For instance, rule `Assemble` (Fig. 2) is enriched with a response time observer by weaving rule `RespTim` (Fig. 4 (c)).

Fig. 11 shows graphically the bindings for the weaving rules `Assemble` and `RespTime`. The pattern highlighted in rule `RespTime` is matched with rule `Assemble` given this binding: In the left-hand side, there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) that holds two `Requests` (`Handle` and `Head`) and an out-`Queue` (`Conveyor`). In the right-hand side,

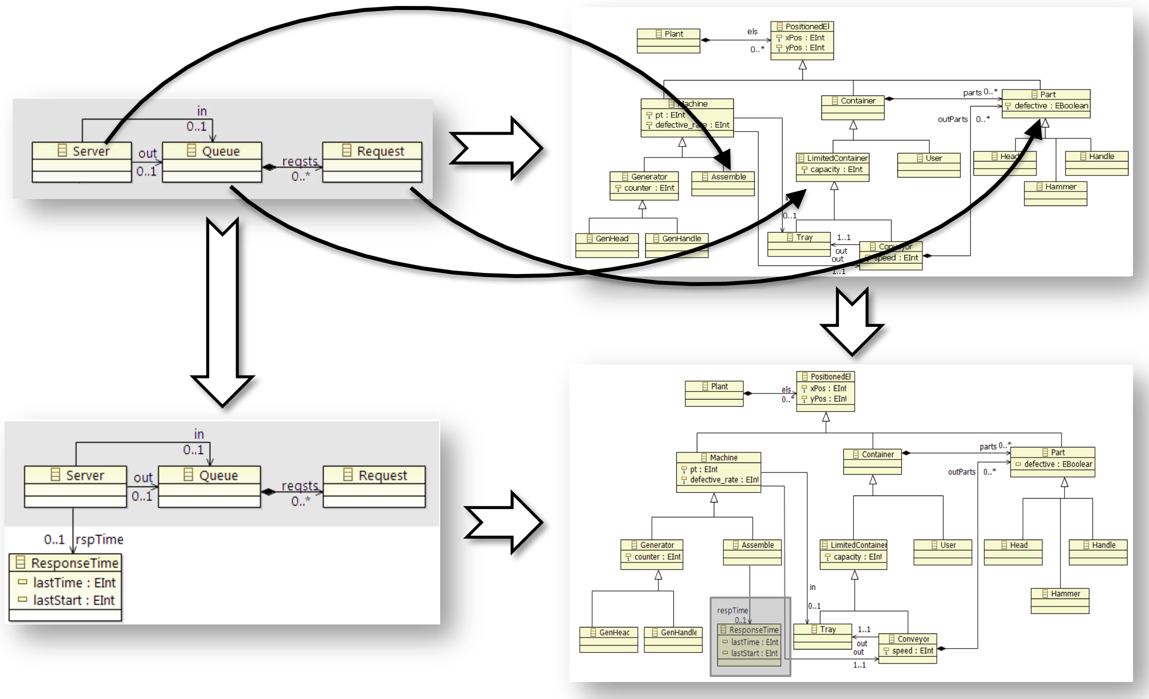


Fig. 10: Weaving of metamodels.

there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) and an out-`Queue` (`Conveyor`) that holds one `Request` (`Hammer`). Rule `Assemble'` depicted in Fig. 11 results of the second ATL transformation application (`WeaveBeh.atl` in Fig. 9). Rule `Assemble'` together the rest of rules specifying the product line system DSL behavior conforms the model $Rls_{\widehat{DSL}}$.

5.2 Weaving of the *Throughput* generic observer

As for the response time, we systematically bound the *Throughput* generic observer and weave with the PLS DSL. We firstly bound the throughput metamodel (Fig. 5 (a)) with the PLS DSL metamodel (Fig. 1 (a)) in which:

- `System` is bound to `Plant`;
- `Server` is bound to `Conveyor`;
- and `Request` is bound to `Hammer`.

Note we will measure the number of hummers generated in a plant per time unit. The concept which will consume "requests", in this case, is the `Conveyor` class, whose instances will store hammers until they will be consumed. Following the throughput metamodel, the `ThroughputOb` class will be associated to the class `Plant` after the weaving, thus meaning the throughput is a global measurement of all the system.

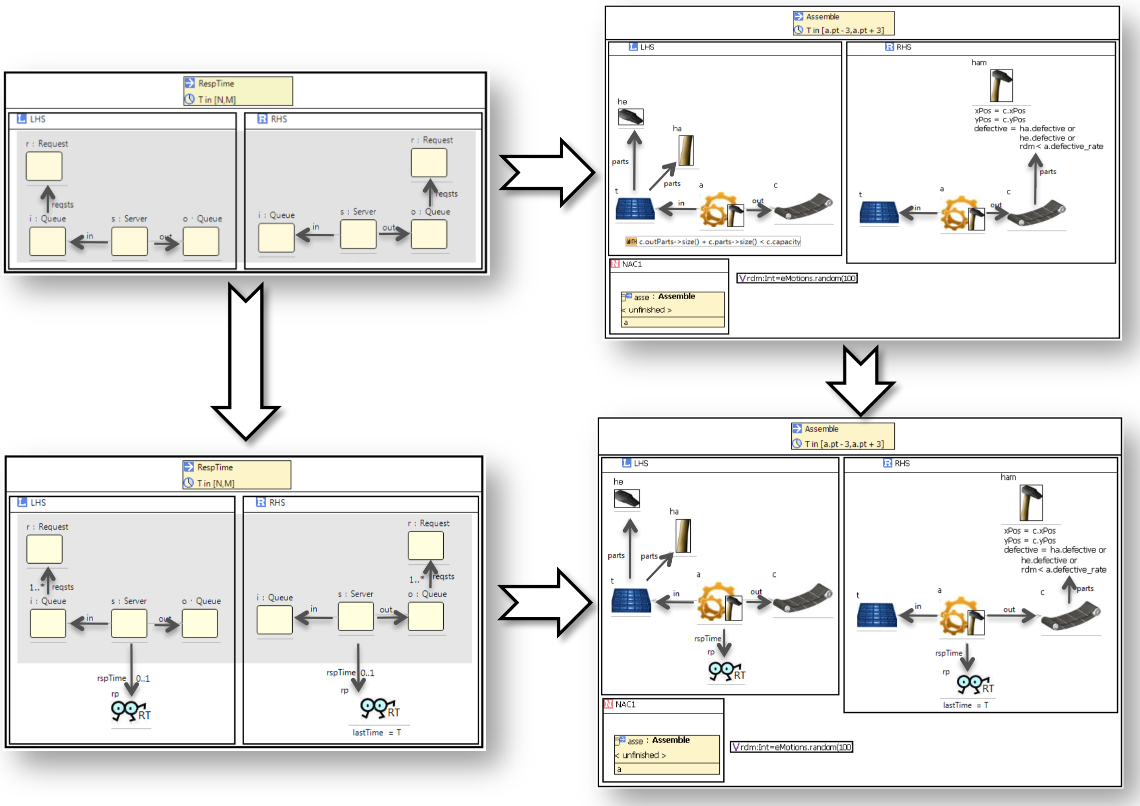


Fig. 11: Weaving of the `Assemble` rule.

Finally, the rule specifying the throughput behavior (Fig. 5 (c)) is matched with rule `Collect`, which amalgamation is shown in Fig. 12. Server `s` is matched with conveyor `c`, whereas request `r` is matched with hammer `h`. Links are also bound.

6 Palladio Component Model

The Palladio Component Model (PCM) [2] is a set of metamodels for the specification of component-based systems. It focuses on the description of performance-relevant aspects of a system architecture, as workloads or hardware utilization, with the aiming of the prediction of Quality-of-Service (QoS) attributes.

The Palladio Architecture Simulator [23] is an Eclipse-based tool which provides graphical editors to specify the different views and models of a system.[2] Following a component-based development process [27] one may specify the different views of a system, as it might be: usage, system and resource models and component repositories.

Instances of the PCM metamodel, which fully specify a system, are then transformed into other models more suitable to analysis. Prediction analysis are performed (i) by

[2]Palladio Modeling Workbench is available at `https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model`.

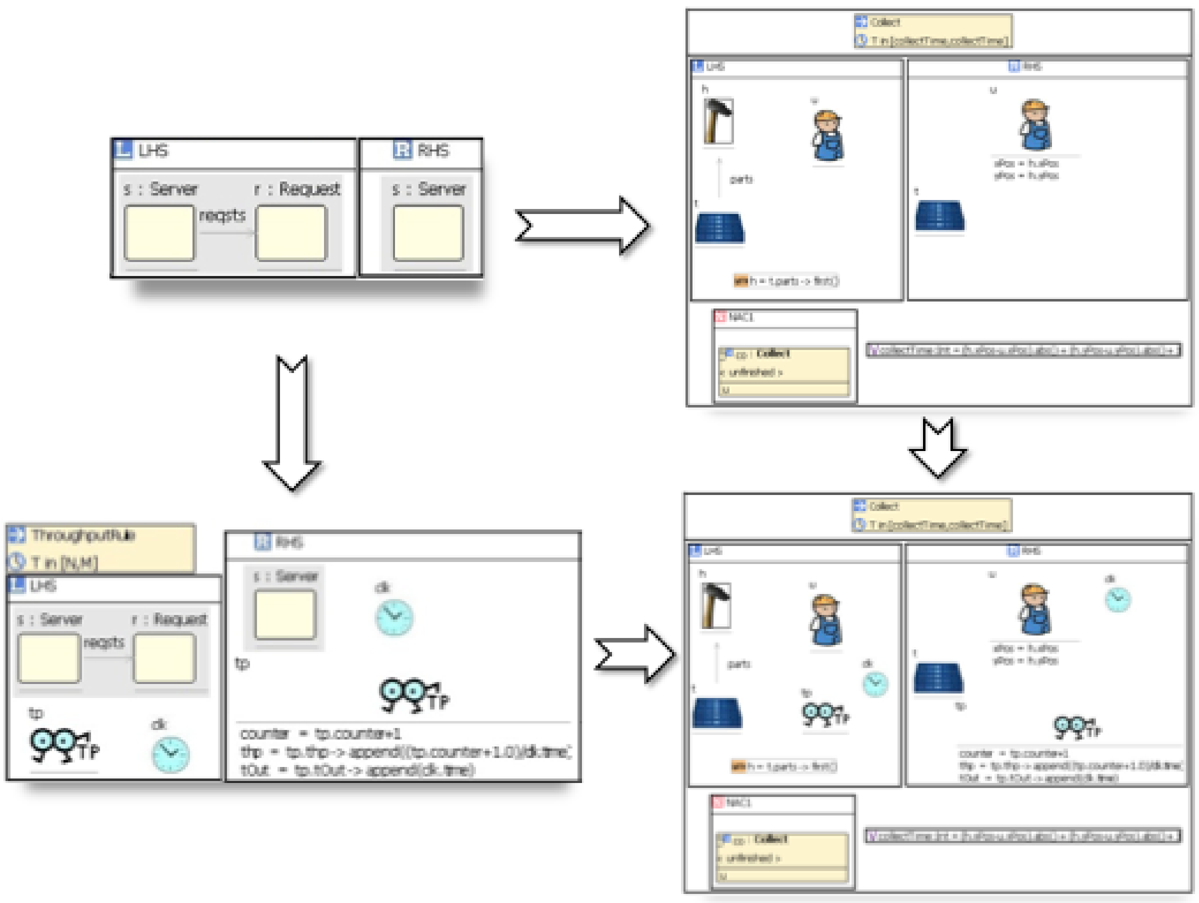


Fig. 12: Weaving of the `Collect` rule.

discrete-event simulation or (ii) by analyzing the models in other formalism, for example Layered Queuing Petri Nets [43, 28]. As for the measurements gathered, both kinds of analysis return the response time of each component and the system and the utilization of each resource, i.e. CPU and HDD. Analysis results can answer questions like "*How will the response time of the system be if the number of users is multiplied by 10?*".

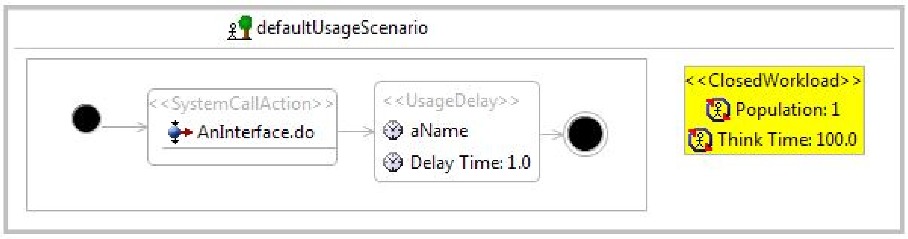
The semantics of the PCM is encapsulated in the transformations into other models or formalism. This makes very difficult both (i) the understanding and validation of analyzed non-functional properties and (ii) extending Palladio with new properties to be analyzed. Palladio consists of over 4 million lines of code written in 12 languages.[3]

6.1 Minimum Example

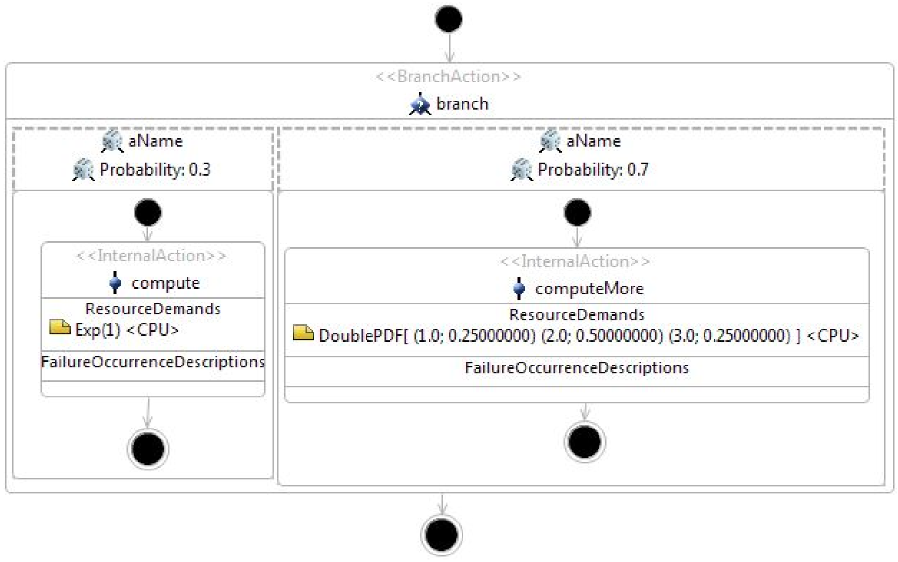
The *Minimum Example* project is provided by the Palladio team within the example workspace.[4] It shows the main features of the PCM, as it are the flow of the systems, resource utilization, etc. Fig. 13 shows the main models involved in the definition of this case study.

[3]Based on data obtained from `http://www.ohloh.net/p/palladio`, June 2014.

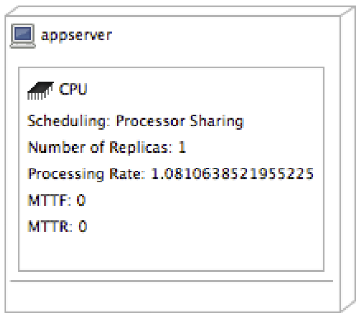
[4]Minimum Example is available online at `https://sdqweb.ipd.kit.edu/wiki/Palladio_Examples`.

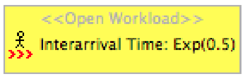


(a) Usage Model.



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


appserver
CPU
Scheduling: Processor Sharing
Number of Replicas: 1
Processing Rate: 1.0810638521955225
MTTF: 0
MTTR: 0



(c) Resource Environment specification. **(d)** `OpenedWorkload` spec.

Fig. 13: *Minimum Example*: Specification in Palladio.

Fig. 13 (a) specifies how is the flow of the tasks or request within the system. The flow of a request when it arrives to the system is modeled by a set of actions connected by links. Start, stop, branches, calls to components and delays are examples of actions. For this example, the first action a request performs is a calling to a component with signature `do` implementing the `anInterface` interface. Then, a delay with 1.0 time unit is performed before the request ends, i.e. it leaves the system leaves the system. The load of the system is modeled by a `Workload` object. The `ClosedWorkload` and `OpenedWorkload` objects model a fixed number or a infinite stream of requests, respectively. For this example, the load of the system is modeled by a `ClosedWorkload` with 1 task, specified with the `population` attribute; and an inter-arrival rate of 100.0 time units, the `think time` attribute.

Fig. 13 (b) shows the specification of the component which implements `anInterface`, specifically the operation `do`. A component is specified by a Resource-Demanding Service-Effect (RDSEFF) model. The flow is branched into one of the two branches, which is probabilistically chosen depending on the `probability` attribute. Both branches perform internal actions, which demand resources. The exponential distribution $exp(1)$ or the distribution given by the density function $[(1.0; 0.25)(2.0; 0.5)(3.0; 0.25)]$ specify the cycle units of CPU required by such a task.

Fig. 13 (c) specifies the available resources in the system being modeled, as the CPU. Number of instances, processing rate and scheduling policy are the type of information needed to perform a prediction analysis. Each of these models correspond to a developer role or view of the system being modeled.

From the models above presented for the Minimum Example, one may perform—among others analyses provided by Palladio—(i) the code generation of Java code skeletons or (ii) a prediction-analysis of the non-functional properties, namely performance and reliability. For this example, QoS analysis makes no sense, because it models just an user in the system. To show how Palladio presents results, we have changed the `ClosedWorkload` object by an `OpenedWorkload` with inter-arrival rate of $exp(0.5)$ time units, as Fig. 13 (d) shows. With this modification, Palladio gives the mean response time and confidence intervals in Table 2. The chart in Fig. 22 (a) represents the cumulative distribution function of the system's response time. Since the CPU resource gets saturated, the response time keeps increasing along time. For 1,000 runs, tasks take up to 24 time units.

Table 1: Palladio: results of Plain Batch Means Algorithm.

Mean value:	4.9647
Confidence value alpha:	0.9
Upper bound:	5.2656
Lower bound:	4.4665

6.2 Another Palladio case study

In this section we present another very simple case study. Fig. 15 shows some of the models involved in the case study specification.

Specifically, Fig. 15 (a) define the global behavior of requests within the system. Requests arrive to the system each 2 time units, as the `Open Workload` object states. As for the component `do` called, Fig. 15 (b) shows the resource-demanding service-effect specification (RDSEFF) showing that the control flow in our component may branch into either of three flows, with different CPU demands for each flow. As in the Minimum Example, each branch is associated with a particular branch probability to indicate the likelihood of a particular branch being taken.

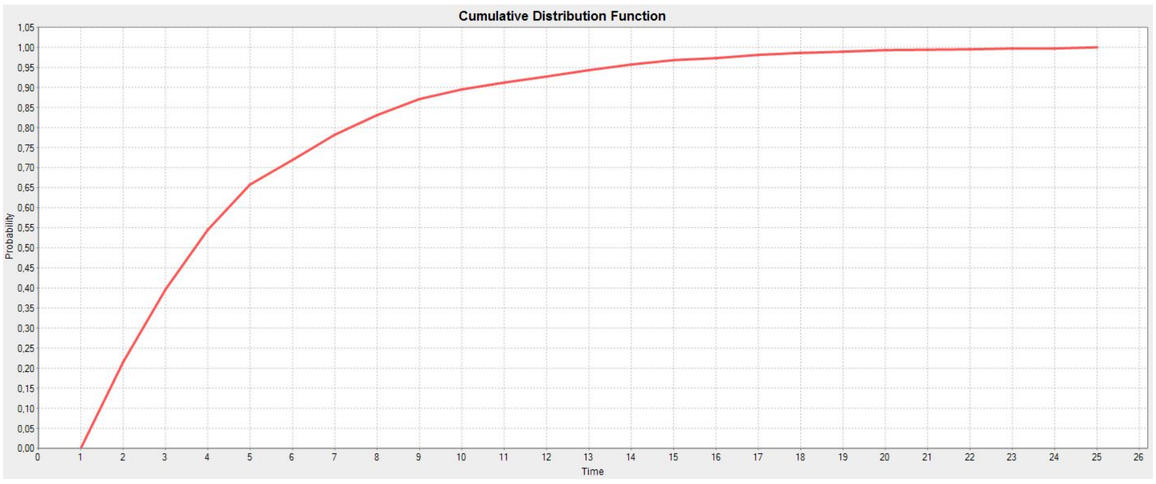
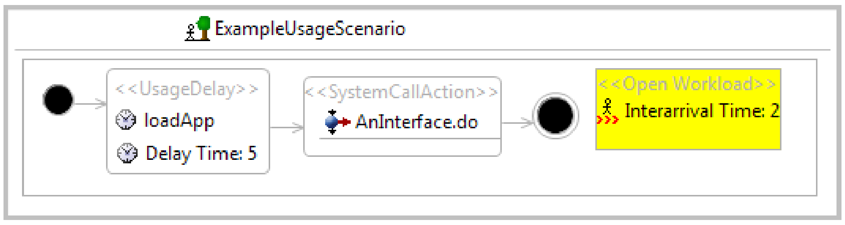
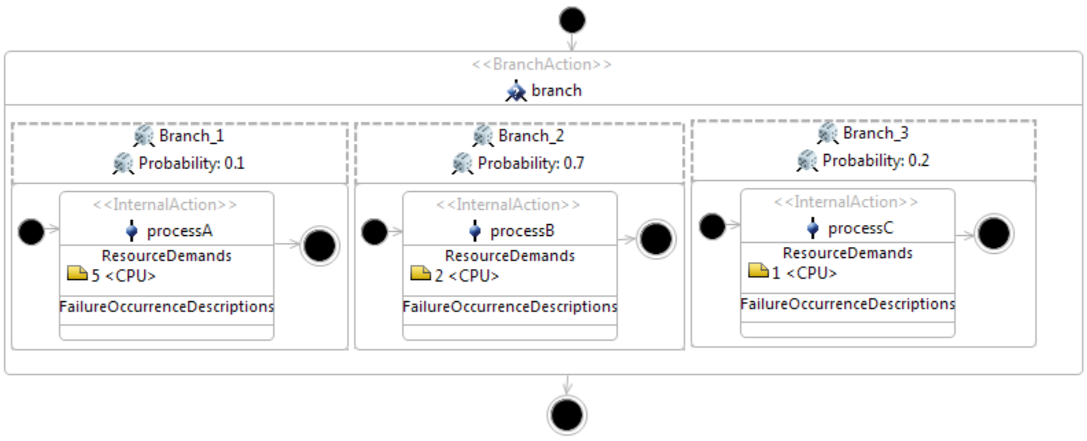


Fig. 14: Cumulative distribution function of the system's response time.



(a) Usage model.



(b) Component model.

Fig. 15: Case study specification in Palladio.

7 An e-Motions definition of Palladio

The Palladio Component Model is a DSL [2], and therefore we may define it in e-Motions. As for any DSL, the definition of PCM includes its abstract syntax, its concrete syntax and its behavior.

Since Palladio Architecture Simulator has been developed following MDE principles, and specifically it has been implemented using the Eclipse Modeling Framework (EMF), its metamodels can be directly used as abstract syntax in e-Motions. PCM models consist of several views of a system, corresponding to the different roles in the cycle life of a system development. These models are conformant to metamodels `PCM Core`, `Stoex`, `Identifier` and `Units`, conforming PCM and used by the Palladio Architecture Simulator.

The concrete syntax of the Palladio definition is defined by a GCS model, in which an image is attached to every concept specified in the PCM metamodels. To maintain the PCM's look, we have used in the e-Motions definition the same images used by the Palladio Architecture Simulator.

PCM models define the architecture of a system and all the information needed to perform a prediction analysis. Thus, PCM models are static, in the sense of they cannot be run by themselves, but they are transformed into Layered Queuing Petri Nets [28, 43] or stochastic process algebra to be analyzed, or java code to be simulated.

In e-Motions, we describe how systems evolve by describing all possible changes in the model—or state—by corresponding visual rewrite rules, that is, time-aware *in-place* transformation rules. Since the PCM metamodel only specifies those relevant concepts needed to specify and to analyze systems, we have added the *Token* concept to handle the control flow of an execution.

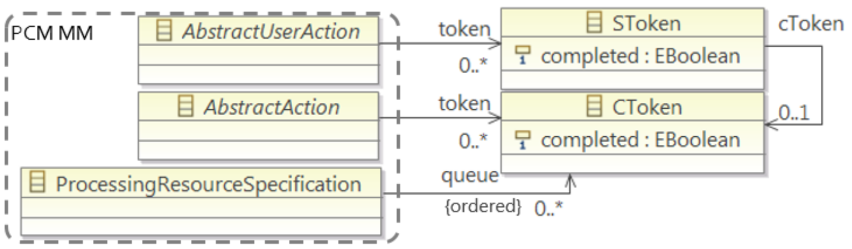
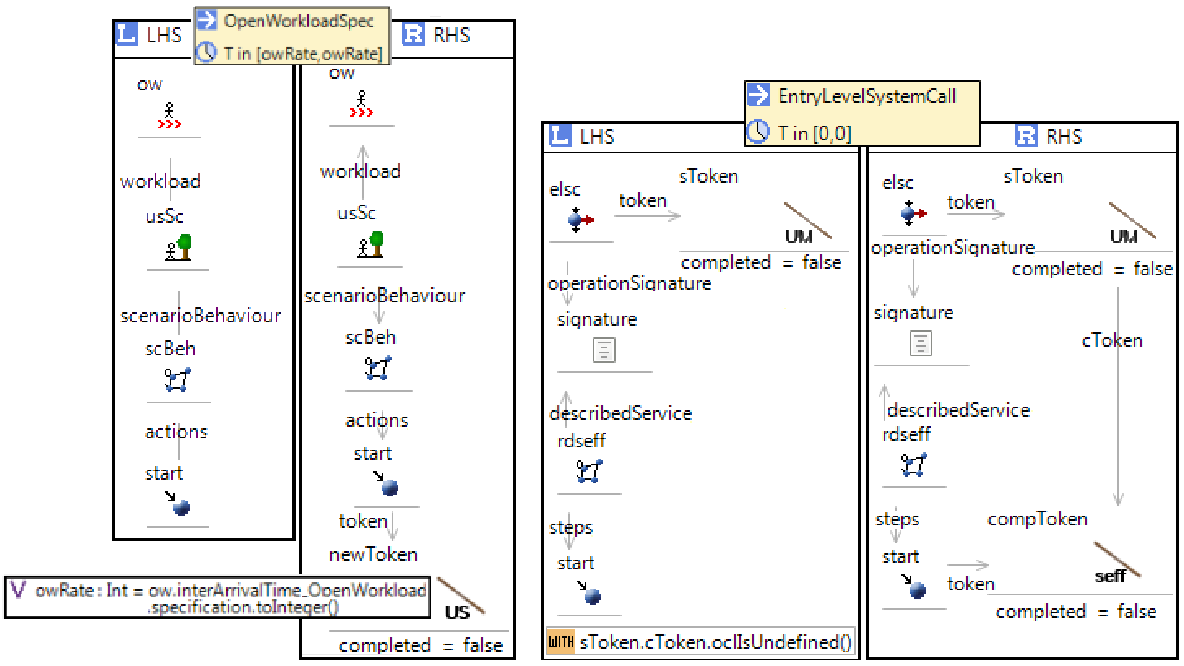


Fig. 16: Token metamodel.

The Palladio metamodel has been conservatively enriched with the `Token` metamodel, thus obtaining the so-called *Palladio* metamodel*. Since this extension of the Palladio metamodel has been done in a conservative way, each model which conforms to Palladio metamodel, also conforms to Palladio* metamodel. Fig. 16 shows the `Token` metamodel. A token represents a task within the system, and its *Bool* attribute `completed` states whether the action containing the token has been performed. The `Token` metamodel consists of two classes `SToken` and `CToken`, being the former specified at the system level (`UsageModel`) and the latter at the component level (`RDSEFF`). References—with cardinality `*`—has been added to the classes `AbstracUserAction` and `AbstractAction`, which are the common parent abstract classes to all actions in the `UsageModel` and `RDSEFF` metamodels, respectively. Ordered reference `queue` is used to model a queue where tokens wait until any instance of the resource—modeled with `ProcessingResourceSpecification` objects—will be available.

We may visualize that the execution of a Palladio model has a token "moving around" such model. Each visual rewrite rule specifying all possible actions of a Palladio model follows a similar pattern: the LHS of the rule specifying an action with a `SToken`/`CToken`, with `completed` set to `false`, and the RHS with the same action and token with its `completed` attribute set to `true`. Moreover, two rules move *completed tokens* from an action to its succeeding action (changing the `completed` attribute to `false`).



(a) `OpenWorkloadSpec` rule. (b) Component call.

Fig. 17: New request rule specification.

In Palladio, the load of a system is modeled by `Workload` objects. Specifically, the `OpenedWorkload` object models an infinite stream of requests, with an inter-arrival rate specified by a `RandomVariable` object which may be a stochastic expression. Fig. 17 (a) shows the `OpenWorkloadSpec` rule, which specifies a `UsageScenario` with an `OpenedWorkload` associated to it. Whenever a matching with the LHS of the rule is found, i.e. there is a state with an `UsageScenario` containing a `ScenarioBehavior` with a `Start` action, a new `SToken` is created and associated to action `Start` of the `UsageModel` after `owRate` time units, being `owRate` a local variable containing the value of the `OpenedWorkload` stochastic expression.

On the other hand, Fig. 17 (b) shows how a component call or an `EntryLevelSystemCall` action is modeled. If there is an `EntryLevelSystemCall` (object `elsc` in Fig. 17 (b)) containing a `SToken` *not* completed and it does not have a `CToken` yet (stated by the positive OCL condition in the LHS), then a new `CToken` is created and linked to the `Start` action of the component specification (`RDSEFF`). Token `sToken` wait associated to `elsc` component call until the token `compToken` reaches the `Stop` action of the component and it is completed. Note that this rule is performed in zero time (it is instantaneous).

Fig. 18 shows an example of action specification, specifically the `Delay` action. If a `Delay` object contains a `Token` whose `completed` value is `false`, rule `DelayUsageModel` will be triggered. After `spec` time units (notice that `spec` is an *Integer* value which stores the stochastic value specified in the delay time specification), the `completed` attribute will be set to `true`.

Fig. 19 shows how `SToken` objects are passed from one action to its successor action.

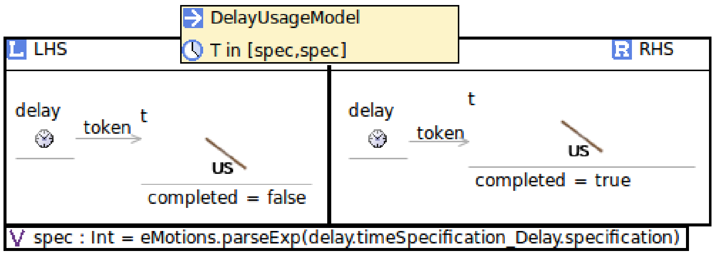


Fig. 18: Delay specification.

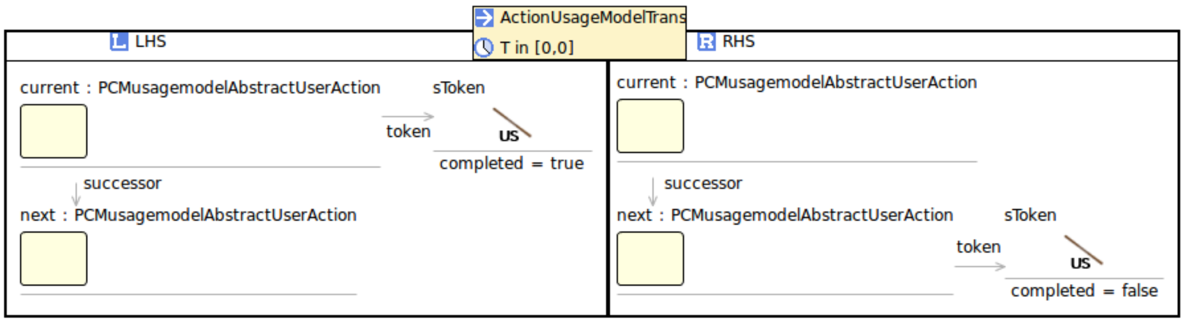


Fig. 19: Token passing specification.

The `ActionUsageModel` instantaneous rule passes a token of an action `current` to its successor action `next` if the action has been completed, that is, if the `completed` attribute is set to `true`.

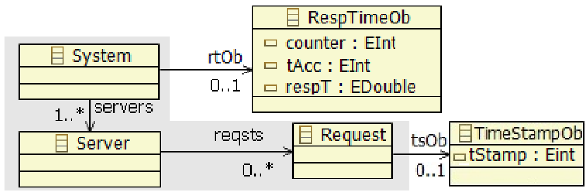
8 The Palladio Specification with Observers

Once a system has been specified with a DSL, as it is the case of Palladio (see Section 6), properties to be monitored and analyzed may be chosen from a library containing parametric DSLs. As explained in Section 3, in order to introduce observers in our specifications in e-Motions, we need to weave both the metamodel and the behaviour specifications of a specific system and the generic observer DSL. In other words, the parametric components of the observers DSLs get instantiated with specific components.

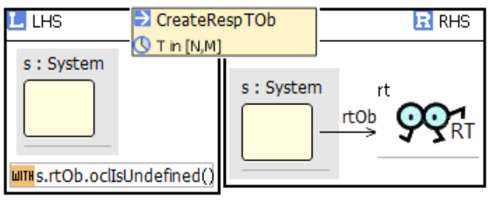
Modular and generic observers are systematically instantiated with concepts and rules of the Palladio specification. For example, for weaving the metamodel of response time (Fig. 20 (a)) with the metamodel of Palladio, the `System` class is mapped to the `ScenarioBehaviour` class, `Server` to `Start` and `Request` to `SToken`. References as `servers` and `reqsts` also have to be bound.

We use a different specification of the response time because the previously defined in Section 3.1 because the former is defined "at the system level", whereas the latter is specified "to a concrete server". I.e., the response time of Fig. 20 is the mean time that elapse between a request enter in the *global system* until such request leaves.

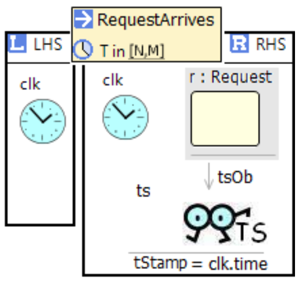
Regarding rules, we basically need to map each rule in the source DSL to a rule in the target one. Although currently correspondences are detailed for the behavior—i.e.



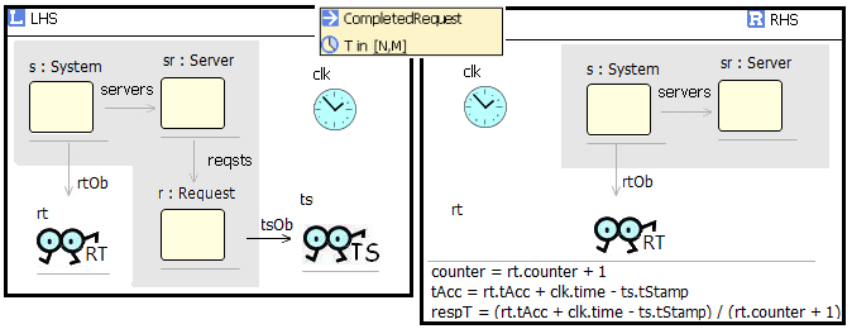
(a) Abstract syntax.



(b) Behavior: CreateRespTOb.



(c) Behavior: RequestArrives.



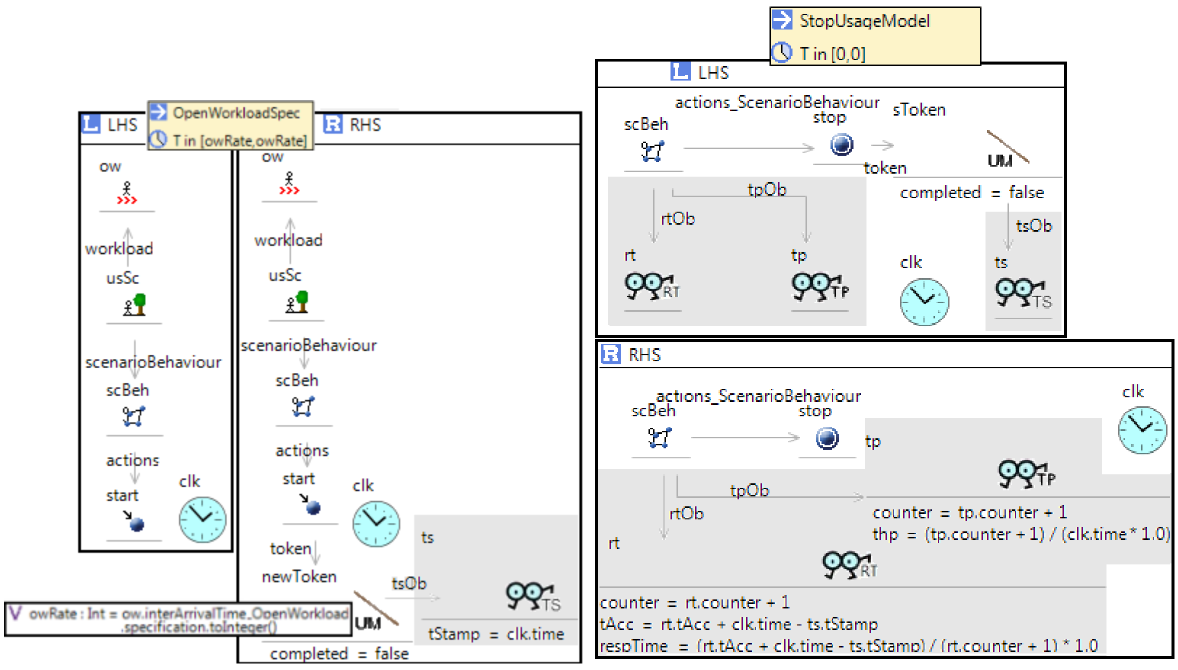
(d) Behavior: CompletedRequest.

Fig. 20: Response Time observer DSL definition.

correspondences among rules, objects and links—, note that most of them can be inferred from the metamodel correspondences. For example, if class `System` is instantiated by `ScenarioBehaviour`, all object of type `System` within the LHS and RHS of any rule also are going to be instantiated by `ScenarioBehaviour` objects. As for the mapping with the response time, rule `RequestArrives` (Fig. 20 (c)) is woven with the `OpenWorkloadSpec` rule of our Palladio system (Fig. 17 (a)), that represents the arrival of a new `SToken` into 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. 20 (d)) is woven with the `StopUsageModel` rule, which models the elimination of a token upon its arrival to a `stop` action.

A similar mapping is provided for the throughput observer: rules `CreateThpOb` and `UpdateTHP` are woven to the identity rule, as `CreateRespTOb`, and rule `UpdateCounter` is mapped to `StopUsageModel`.

The result of weaving the response time and throughput observer DSLs and the Palladio* DSL results in a DSL whose 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. Figs. 21 (a) and 21 (b) show the rules `OpenWorkLoad`


LHS
OpenWorkloadSpec
T in [owRate,owRate]
RHS
ow
workload
usSc
scenarioBehaviour
scBeh
actions
start
clk
token
newToken
ts
tsOb
tStamp = clk.time
completed = false
UM
owRate : Int = ow.interArrivalTime.OpenWorkload.specification.toInteger()
StopUsageModel
T in [0,0]
actions_ScenarioBehaviour
stop
sToken
rtOb
tpOb
rt
tp
counter = tp.counter + 1
thp = (tp.counter + 1) / (clk.time * 1.0)
counter = rt.counter + 1
tAcc = rt.tAcc + clk.time - ts.tStamp
respTime = (rt.tAcc + clk.time - ts.tStamp) / (rt.counter + 1) * 1.0

(a) Enriched `OpenWorkloadSpec` rule. **(b)** Ennriched `StopUsageModel` rule.

Fig. 21: Woven rules.

(Fig. 17 (a)) and `stop` as resulting from the weaving process.

Using the same mechanisms these observers may be attached to other elements of the model. For instance, we can in this way measure the response time of each of the components in the system. Additional observers for other NFPs may be considered similarly.

In the following, we present some analysis of the case study presented in Section 6.2 performed by simulation in e-Motions.

e-Motions specification of Palladio can be directly fed with such Palladio models defining a system, just with minor changes, e.g. an ATL transformation grouping all classes within all models in an unique package, since e-Motions currently is not able to handle several packages. This resulting model can be simulated in e-Motions, using those rewrite rules specifying Palladio (see Section 7). Moreover, as such Palladio specification has been enriched with response time and throughput observers, those information relevant to analyze both response time and throughput are collected during the simulation.

Table 2: Palladio: results of Plain Batch Means Algorithm.

Mean value:	41.9713
Confidence value alpha:	0.9
Upper bound:	52.1718
Lower bound:	31.7709

Table 2 shows the results obtained after a 1,000-request simulation, measuring the mean response time and providing a confidence interval. On the other hand, Table 3

Table 3: Case study's e-Motions results.

Mean System Response Time	43.6626 seconds
Throughput	0.4804 seconds

shows results from e-Motions, specifically from response time and throughput observers. Note how after in addition of the response time, our e-Motions specification also is able to measure the throughput of a system. Besides response time or throughput, any NFP may be easily defined and weaved in the e-Motions specification of Palladio, thus analyzing such NFP by simulation.

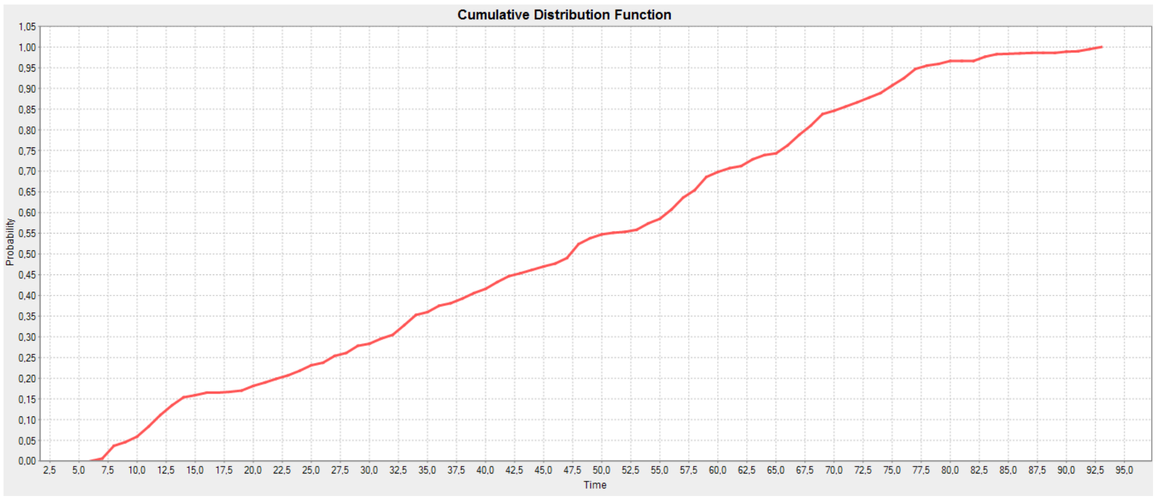
In addition to numbers presented in Table 2 and 3, we may also graphically present the information gathered by response time observers. Fig. 22 shows charts representing the cumulative distrubution functions in Palladio and e-Motions. Fig. 22 (a) illustrates the chart directly obtained from the Palladio Simulator, whereas Fig. 22 (b) shows the chart realized from response time's information.

We may define observers to collect any raw information. Specifically, response time observer may collect the response time and the entrance time of each request. After gathering such information, we may analyze it and generate a chart, as the showed in Fig. 23. One may observer the response time increase through the entrance time. This occurs because the queue associated to CPU resources gets saturated, and request remains more and more time waiting to be processed. With this simulation we visualize that the system architecture have to be modified. One posible solution is to increase the number of CPU resources, or the processing rate, both solution aiming to keep queues without saturation.

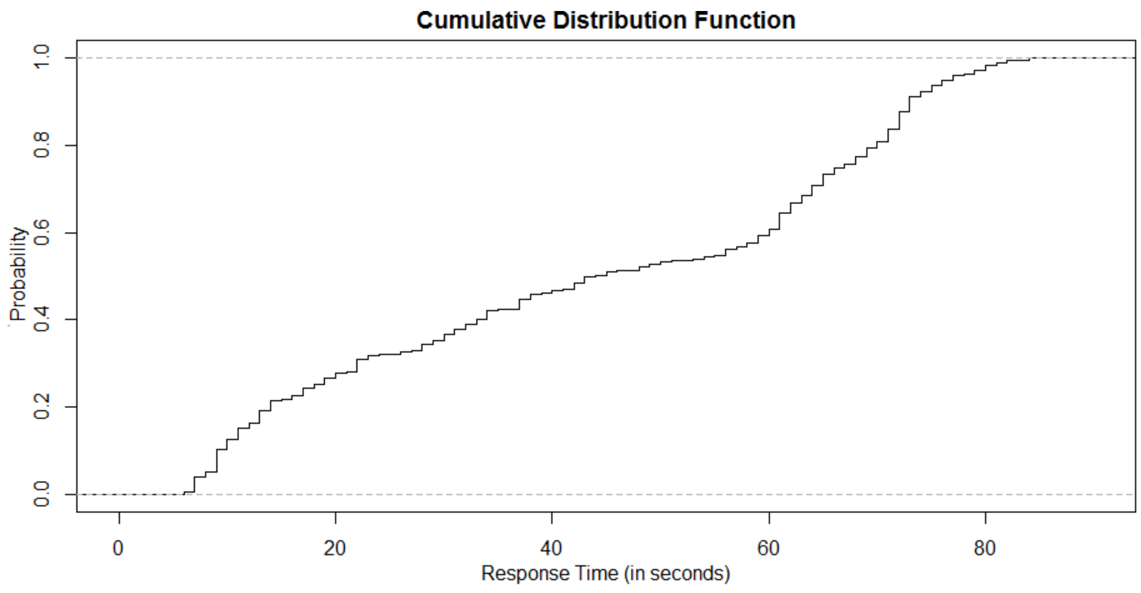
9 Related Work

While syntax is reasonably well understood, behavior remains yet as an unsolved problem. Over the last years, many authors have proposed and surveyed different techniques to compose both syntax and behavior. Most of the works in the literature of the MDE community are targeted to *model management*, that is, model to model transformation, model merging, model comparison, model matching, model synchronization or traceability relations. In the following, we systematically introduce some of them.

De Lara et al. [30] have developed the research most closely related to ours. They propose the use of *concepts*, *hybrid concepts* and *mixin layers*. Concepts have been inspired by generic programming in C++, in fact, they can be seen as templates. With concepts, behaviors of any system may be defined at the level of concepts instead of at the level of a concrete metamodel. Metamodels specify a type requeriment. This is specially useful because concepts allow the reuse of the behavior or code with any metamodel which holds the concept type requirement. Bindings among concepts and metamodels that realize a concept are given explicitly by a function $bind : C \to M$. Hybrid concepts appears as a relaxation of the type requirements imposed by concepts, since concepts require an embedding of this in the metamodel. The power of hybrid concepts comes from being able to hide some details of the types to be required and requires them by realization of



(a) Cumulative distribution function in Palladio.



(b) Cumulative distribution function for the simulation in e-Motions.

Fig. 22: Cumulative distribution functions of system's response time in Palladio and e-Motions.

operations. In this sense, hybrid concepts may be seen as *interfaces* (e.g. Java interfaces) for metamodels. Finally, a *mixin layer* is a metamodel containing a set of auxiliary elements, which provide an extensible way of defining languages. Mixin layers are used to extend DSL with common functionality. For example, the elements needed to simulate a system do not correspond to the system but are indispensable. In [31] is presented a plethora of possible metamodel abstractions using this techniques. Rose et al. [39] shows an implementation of such concepts in the *Epsilon*[5] language.

Wimmer et al. [51] extend the previous work of Cuadrado et al. [9] defining and imple-

[5]More information about the Epsilon project is at `https://www.eclipse.org/epsilon/`.

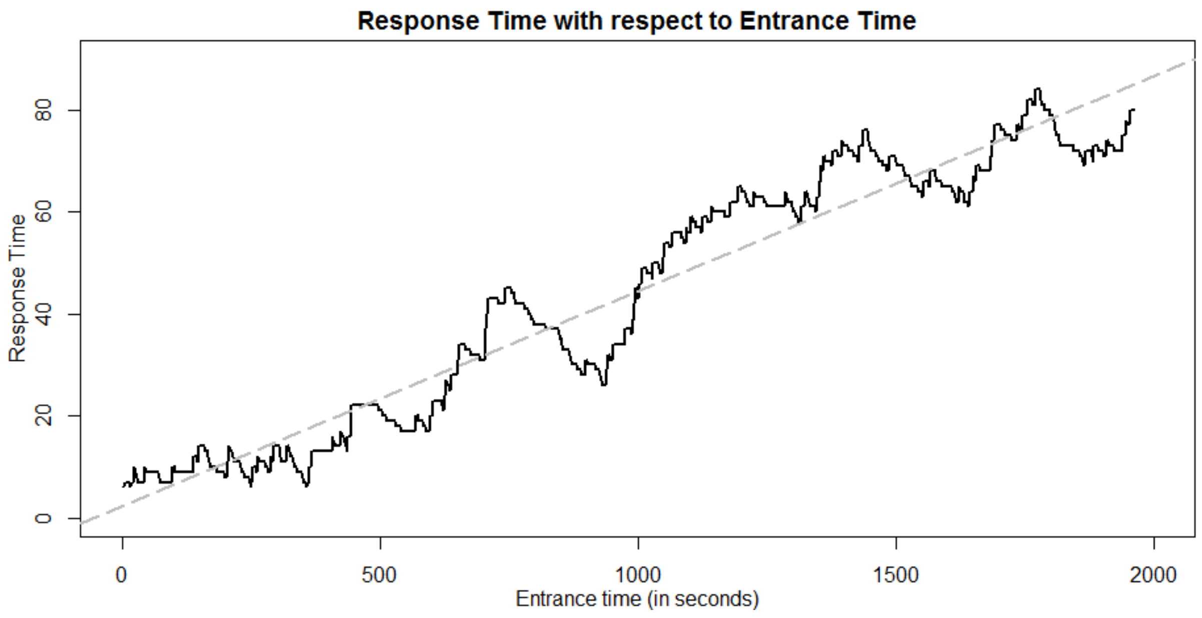


Fig. 23: Response Time obtained from e-Motions output.

menting model to model transformations (MT) which are defined with *generic metamodels*, although, following previous work presented in [30] they call them *concept metamodels*. Once a MT is defined using concepts as source and target metamodels, given two metamodels realizing that concepts a new transformation is generated tailored to such concrete metamodels. In fact, this work may be seen as an implementation of [30] using high order transformations on top of ATL [26].

In [50] Wimmer et al. have classified the different approaches of model to model transformations (MT) focusing on their reuse. Reuse can be achieved in different ways, such as unifying functionality in the same MT, or more challenging, reusing MT across different source and target metamodels the MT has been originally developed for. The former is achieved by using helpers or rule inheritance whereas the latter is achieved by parameterized MTs. Cuadrado et al. [41] shows an implementation of parameterized MTs in RubyTL [11].

In the scope of Aspect-Oriented Programming (AOP), in which this work is strongly inspired, Clark et al. [6] have proposed the so-called Aspect-Oriented Metamodelling (AOM). The basic idea is to use package templates and package extensions that may be composed afterwards to obtain accurate metamodels. Similarly, Berg et al. [3] propose the use of templates and inner classes to provide genericity.

Emerson et al. [19] survey some of the most used techniques in the reuse of metamodels, such as metamodel merging, metamodel interfacing and class refinement. They also propose *Template Instantiation*. All this techniques deal with syntax, but not with semantics, which remains as future work.

Vallecillo [48] discusses the most common techniques for DSL combination, specially focusing on the syntax. Basically, a system can be specified as a set of *views*, such as functionality, information about dynamic libraries, deployment information, etc. From all of such views, a new model containing all such view models (and therefore not human readable) is generated to be executed. In DSLs a similar approach may be taken into account, several views of the same system that finally conforms a global model. How to relate this views, to integrate them into a common metamodel and to analysis the resulting model are open problems. This approach is inspired by Open Distributed Systems.

Chen et al.[5] have proposed the use of reusable *"semantic units"*. Semantic units provide reference semantics for basic behavioral categories using Abstract State Machines. They also support the specification and composition of semantic units on their Model Integrated Computing (MIC) tool suite.

Guerra et al. [22] have presented an approach to model management defining visual patterns which describe allowed or forbidden relations between two modeling languages. They specify how two or more modeling languages have to be related, resulting in a model that describes the way in which such a different models can be related. This model inspires how our correspondences model maps a parametric and a specific DSL.

Zschaler has proposed in his recent work [52] a generalization of the notion of a model type. He attempts to define the general basis of a type system to compare and analyze the correctness of the different approaches proposed.

Finally, Farias et al. [21] present an empirical study about the effort that domain experts have to do to compose design models. They compared Epsilon versus IBM RSA,[6] in which subjects mentioned that they often had some difficulties to match and compose the input model elements. Besides, models generated after applying composition techniques had inconsistencies (e.g. erroneous cardinality of the references). This work shows how the model composition is still in a early stage and need more research.

10 Conclusions and Future Work

In this work we have proposed an approach to reuse and compose domain-specific languages (DSLs). We have presented a formal framework stating that, given some natural requirements, an enrichment of a DSL with a parametric DSL is *conservative*. To be conservative means that we can guarantee protection of behavior when parametric DSLs get instantiated. We have also presented an implementation of such formal framework.

To demonstrate the benefits of such composition operations, we have defined a well-known DSL for reliability and performance analysis: the Palladio DSL. Once the key features of the Palladio Architecture Simulator have been specified, we have an explicit behavior definition of it. This way, domain experts may visualize the semantics of non-functional properties being analyzed by Palladio. On the other hand, as the behavior has been specified has a GTS, we have composed it with several non-functional properties, namely response time (it is already analyzed in Palladio) and throughput.

This work serves as an initial study of the state-of-the-art of DSL composition. Although some results have been presented so far, there are much work to do, which may finalize in a dissertation. Among possible works to go on, we find particularly interesting those related with correct bindings. In this line, De Lara et al. [10] have proposed techniques to study and match references with different cardinality or related by inheritance. On the other hand, to guarantee correct bindings leads guaranteeing conservative weaving

[6]`http://www.ibm.com/developerworks/rational/products/rsa/`

of parameterized DSLs. We plan to study and identify under which circumstances the weaving of parametric DSLs is safe. We also plan to study how to infer possible bindings automatically, e.g. whether two metaclasses match, all instances of such metaclasses must match. Finally, we will implement such ideas in final tools to be used by domain experts. I.e., although we have prototypical implementations (see Section 5), bindings have to be given as a model. We plan to create tools which allow us to create correspondences models graphically, by binding elements of DSLs.

References

[1] Atenea: Atenea group (2014), `http://atenea.lcc.uma.es/`

[2] Becker, S., Koziolek, H., Reussner, R.: Model-Based Performance Prediction with the Palladio Component Model. In: Proceedings of the 6th International Workshop on Software and Performance. pp. 54–65. WOSP '07, ACM (2007)

[3] Berg, H., Møller-Pedersen, B., Krogdahl, S.: Advancing Generic Metamodels. In: Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE'11, AOOPES'11, NEAT'11, VMIL'11. pp. 19–24. SPLASH '11 Workshops (2011)

[4] Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)

[5] Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Proceedings of the First European Conference on Model Driven Architecture: Foundations and Applications. pp. 115–129. ECMDA-FA'05, Springer (2005)

[6] Clark, T., Evans, A., Kent, S.: Aspect-oriented Metamodelling. Comput. J. 46(5), 566–577 (2003)

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science 285(2), 187–243 (2002)

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)

[9] Cuadrado, J.S., Guerra, E., de Lara, J.: Generic Model Transformations: *Write Once, Reuse Everywhere*. In: Proceedings of the 4th International Conference on Theory and Practice of Model Transformations. pp. 62–77. ICMT'11, Springer (2011)

[10] Cuadrado, J.S., Guerra, E., de Lara, J.: Flexible Model-to-Model Transformation Templates: An Application to ATL. Journal of Object Technology 11(2), 4: 1–28 (2012)

[11] Cuadrado, J.S., Molina, J.G., Menarguez, M.: RubyTL: A Practical, Extensible Transformation Language. In: 2nd European Conference on Preliminary Version Preliminary Version Model Driven Architecture. pp. 158–172 (2006)

[12] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An Annotated Bibliography. SIGPLAN Not. 35(6), 26–36 (Jun 2000)

[13] Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs, rR 06.02 RR 06.02

[14] Durán, F., Orejas, F., Zschaler, S.: Behaviour Protection in Modular Rule-Based System Specifications. In: Martí-Oliet, N., Palomino, M. (eds.) Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012. Lecture Notes in Computer Science, vol. 7841, pp. 24–49. Springer (2013)

[15] Durán, F., Zschaler, S., Troya, J.: On the Reusable Specification of Non-functional Properties in DSLs. In: Proc. 5th Int'l Conf. on Software Language Engineering (SLE 2012). pp. 332–351 (2012)

[16] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer (2006)

[17] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of Constraints and Application Conditions: From Graphs to High-Level Structures. Fundam. Inf. 74(1), 135–166 (Oct 2006)

[18] Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive High-Level Replacement Categories and Systems. In: Graph Transformations, Second International Conference, ICGT 2004. pp. 144–160 (2004)

[19] Emerson, M., Sztipanovits, J.: Techniques for Metamodel Composition. In: The 6th OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2006. pp. 123–139. ACM (2006)

[20] 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: Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard. pp. 323–337. UML'00, Springer (2000)

[21] Farias, K., Garcia, A., Whittle, J., Chavez, C., Lucena, C.: Evaluating the Effort of Composing Design Models: A Controlled Experiment. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems. pp. 676–691. MODELS'12, Springer (2012)

[22] Guerra, E., Lara, J., Orejas, F.: Inter-modelling with Patterns. Software and System Modeling 12(1), 145–174 (Feb 2013)

[23] Happe, J., Koziolek, H., Reussner, R.: Facilitating performance predictions using software components. IEEE Software 28(3), 27–33 (2011)

[24] Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. A case study in transformation modularity. In: Gray, J., Pierantonio, A., Vallecillo, A. (eds.) International Conference on Model Transformation (ICMT 2008). Lecture Notes in Computer Science, vol. 5063, pp. 183–198. Springer (June 2008)

[25] Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical Assessment of MDE in Industry. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 471–480. ICSE '11, ACM (2011)

[26] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming 72(1-2), 31–39 (Jun 2008)

[27] Koziolek, H., Happe, J.: A QoS Driven Development Process Model for Component-based Software Systems. In: Proceedings of the 9th International Conference on Component-Based Software Engineering. pp. 336–343. CBSE'06, Springer (2006)

[28] Koziolek, H., Reussner, R.: A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In: Performance Evaluation: Metrics, Models and Benchmarks, pp. 58–78. Springer (2008)

[29] Lack, S., Sobociński, P.: Adhesive categories. In: Foundations of software science and computation structures, pp. 273–288. Springer (2004)

[30] de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Software and System Modeling 12(3), 453–474 (2013)

[31] de Lara, J., Guerra, E., Cuadrado, J.S.: Reusable abstractions for modeling languages. Information Systems 38(8), 1128–1149 (2013)

[32] de Lara, J., Vangheluwe, H.: Automating the Transformation-based Analysis of Visual Languages. Form. Asp. Comput. 22(3-4), 297–326 (May 2010)

[33] Moreno-Delgado, A., Durán, F., Zschaler, S., Troya, J.: Modular DSLs for flexible analysis: An e-Motions reimplementation of Palladio. In: 10th European Conference on Modelling Foundations and Applications, ECMFA 2014. Lecture Notes in Computer Science, vol. 8569. Springer (2014)

[34] Moreno-Delgado, A., Troya, J., Durán, F., Vallecillo, A.: On the Modular Specification of NFPs: A Case Study. In: XVIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD). pp. 302–316 (2013), available at `http://www.sistedes.es/ficheros/actas-conferencias/JISBD/2013.pdf`.

[35] Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-Languages. In: Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'05). Lecture Notes in Computer Science, vol. 3713, pp. 264–278. Springer (2005)

[36] Rivera, J.E.: On the Semantics of Real-Time Domain Specific Modeling Languages. Ph.D. thesis, University of Málaga (2010)

[37] Rivera, J.E., Durán, F., Vallecillo, A.: A Graphical Approach for Modeling Time-dependent Behavior of DSLs. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). VLHCC '09, Corvallis, Oregon (US) (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.) Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 174–190. Springer (2010)

[39] Rose, L.M., Guerra, E., de Lara, J., Etien, A., Kolovos, D.S., Paige, R.F.: Genericity for model management operations. Software and System Modeling 12(1), 201–219 (2013)

[40] Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)

[41] Sánchez Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: Proceedings of the 1st International Conference on Theory and Practice of Model Transformations. pp. 168–182. ICMT '08, Springer (2008)

[42] Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (Feb 2006)

[43] Spinner, S., Kounev, S., Meier, P.: Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In: Haddad, S., Pomello, L. (eds.) Proc. 33rd Int'l Conf. Application and Theory of Petri Nets and Concurrency (Petri Nets 2012). LNCS, vol. 7347, pp. 388–397. Springer (2012)

[44] Troya, J.: On the Model-Driven Performance and Reliability Analysis of Dynamic Systems. Ph.D. thesis, University of Málaga (2013)

[45] Troya, J., Rivera, J.E., Vallecillo, A.: Simulating Domain Specific Visual Models by Observation. In: Proc. of the 2010 Spring Simulation Multiconference. pp. 128:1–8. SpringSim'10, ACM, New York, NY (2010)

[46] Troya, J., Vallecillo, A.: A domain specific visual language for modeling power-aware reliability in wireless sensor networks. In: Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages. pp. 3:1–3:6. NFPinDSML '12 (2012)

[47] Troya, J., Vallecillo, A., Durán, F., Zschaler, S.: Model-driven performance analysis of rule-based domain specific visual models. Information and Software Technology 55(1), 88 – 110 (2013)

[48] Vallecillo, A.: On the Combination of Domain Specific Modeling Languages. In: Proceedings of the 6th European Conference on Modelling Foundations and Applications. pp. 305–320. ECMFA'10, Springer (2010)

[49] Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Software 31(3), 79–85 (2014)

[50] Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Fact or fiction — reuse in rule-based model-to-model transformation languages. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations. pp. 280–295. ICMT'12, Springer (2012)

[51] Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Sánchez Cuadrado, J., Guerra, E., De Lara, J.: Reusing model transformations across heterogeneous metamodels. Electronic Communications of the EASST 50 (2012)

[52] Zschaler, S.: Towards Constraint-Based Model Types: A Generalised Formal Foundation for Model Genericity. In: 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, 2014 (workshop co-located with STAF 2014)