Modular DSLs for flexible analysis: An e-Motions reimplementation of Palladio

Antonio Moreno-Delgado[1], Francisco Durán[1], Steffen Zschaler[2], and Javier Troya[3]

[1] University of Málaga
{amoreno,duran}@lcc.uma.es
[2] King's College London
szschaler@acm.org
[3] Vienna University of Technology
troya@big.tuwien.ac.at

Abstract. We address some of the limitations for extending and validating MDE-based implementations of NFP analysis tools by presenting a modular, model-based partial reimplementation of one well-known analysis framework, namely the Palladio Architecture Simulator. We specify the key DSLs from Palladio in the e-Motions system, describing the basic simulation semantics as a set of graph transformation rules. Different properties to be analysed are then encoded as separate, parametrised DSLs, independent of the definition of Palladio. These can then be composed with the base Palladio DSL to generate specific simulation environments. Models created in the Palladio IDE can be fed directly into this simulation environment for analysis. We demonstrate two main benefits of our approach: 1) The semantics of the simulation and the non-functional properties to be analysed are made explicit in the respective DSL specifications, and 2) because of the compositional definition, we can add definitions of new non-functional properties and their analyses.

1 Introduction

It has been generally recognised that the non-functional properties (NFPs)—for example, performance or reliability—of a system are central to the success of a software development project. The later in the process an error in NFPs is discovered, the more costly will it be to repair. There is, therefore, a need for early predictive analysis of NFPs.

Model-driven engineering (MDE) advocates the use of models as the primary artefacts in software development. It has been recognised that this provides opportunities for very early analysis of NFPs based on early design models. These models can often be transformed into analysis models (e.g., in the form of Petri nets or queuing networks) that can be analysed or simulated by standard tooling [1–3, 7–9].

Typically, in these approaches a design model is translated into an analysis model which is then evaluated by a dedicated analysis tool. Alternatively, the

design model is translated into a simulation of the system to be built. In both cases, however, the semantics of the non-functional property to be analysed and of the analysis technique are only represented implicitly as encoded in the transformations or analysis tools. This causes two problems:

1. *Validation of analysis.* As there is no explicit specification of the analysis nor a high-level representation of the NFPs to be analysed, it is difficult for users to be sure that they are analysing the correct property of their system (see, e.g., [12] for a discussion of some of the subtleties that might need to be considered). Conversely, it is also very difficult for tool providers to validate the correctness of their tooling, which has a direct impact on the correctness of their predictions.
2. *Maintainability and extensibility of analyses.* The tool implementations, especially in the transformations producing simulations, often tangle code concerned with different NFPs. For example, the transformations used in the Palladio Architecture Simulator [9] tangle code for performance and reliability simulations. This makes the code very difficult to maintain and, in particular, extend to support new NFPs.

In previous work [6, 10, 17], we have explored the modular definition of non-functional properties as parametrised domain specific languages (DSLs) in the e-Motions framework [11]. In the present paper, we demonstrate how these ideas can be integrated with predictive analysis of architectural software models by providing a modular reimplementation of a substantive part of the Palladio Architecture Simulator [9]. In particular, we have re-implemented the Palladio Component Model [3], its workload model, and parts of its stochastic expressions model. However, instead of implementing transformations to analysis models or simulators as done in Palladio, we have explicitly modelled the simulations as graph transformations in the e-Motions framework. Each NFP to be analysed is then modelled as an independent, parametrised DSL ready to be composed with the base Palladio model. This addresses the above two problems in the following ways:

1. There is an explicit specification of both the simulation mechanism and the NFPs to be analysed. These models can be inspected and reasoned about separately giving more assurance of correctness of the simulation results.
2. Modular definition of NFPs as separate, parametrized DSLs allows its reuse, but also makes it easy to define additional NFPs to be analysed. For a particular analysis problem, the relevant NFP DSLs can then be selected from a library and composed as required. Our previous work in [5] provides guarantees for preservation of semantics under composition, that is, the consideration of additional NFPs (satisfying certain restrictions) do not change the behaviour of the system being modeled.

While our approach may not be as performant for large models as the native Palladio implementation, its modular and model-based nature mean that new analyses can be prototyped very effectively. These might then still be translated

into native implementations tightly integrated with Palladio where efficiency of analysis is a concern over full validation of analysis. We present in this paper the specification of the NFPs response time and throughput, but new types of analysis could be easily added. One such analysis that could be easily prototyped in our approach is support for dynamic systems — this possibility has already been explored in [17]. In e-Motions, this effectively amounts to a number of additional rewrite rules for the base model.

The remainder of this paper is structured as follows. Section 2 provides some background on the two MDE frameworks our work relies on, namely Palladio and e-Motions. Section 3 explains how the Palladio DSL has been defined in the e-Motions system. Section 4 describes the way observers are defined and how they are woven with the Palladio system to enrich the definition of its behavior for the observation of NFPs. Section 5 illustrates our approach on a concrete example and compares the results obtained by Palladio and by its e-Motions counterpart. We wrap up with some conclusions and future work in Section 6.

2 Preliminaries

Our work is based on two MDE frameworks: We use Palladio [9], and in particular the Palladio Component Model (PCM) [3], to allow modelling of component-based systems and their performance-relevant properties; and we use e-Motions to implement simulations of these systems' performance properties (as well as of other non-functional properties). In this section, we provide some background on both frameworks to ground the discussion that will follow.

2.1 Palladio

The Palladio Architecture Simulator [9] is a predictive software analysis tool developed by the group around Ralf Reussner at KIT in Karlsruhe, Germany. It consists of a number of metamodels, foremost the Palladio Component Model (PCM) [3], that allow the high-level modelling of component-based architectures and their properties relevant for performance and reliability analysis. Instances of these metamodels are then transformed in preparation for analysis. Palladio supports two kinds of predictive analyses: 1) by transformation into a program that runs a simulation of the architecture's behaviour and 2) by transforming to a formalism more amenable to analysis—for example, Queuing Petri Nets [14]. In both cases, the semantics of the models, and in particular of the non-functional properties being analysed, is encapsulated in the transformations. This makes it very difficult to understand and validate these semantics. This is particularly problematic as more non-functional properties are supported: the current transformations support performance and reliability, but already are quite complex. Palladio consists of over 4 million lines of code written in 12 languages.[1]

Fig. 1 shows a very simple example of a component specification in Palladio. It shows a so-called resource-demanding service-effect specification (RDSEFF)

[1] Based on data obtained from `http://www.ohloh.net/p/palladio` on Feb. 4, 2014.

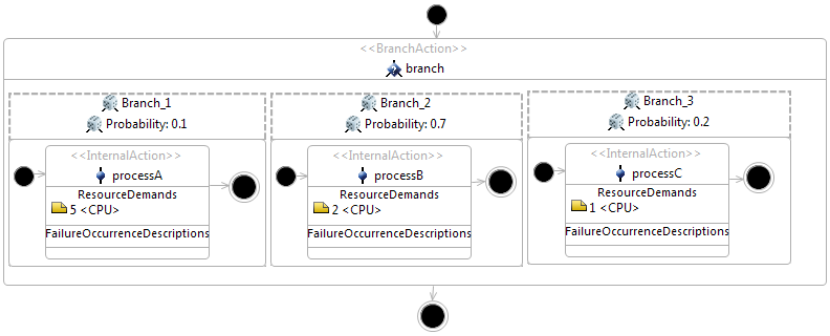


Fig. 1. Component model.

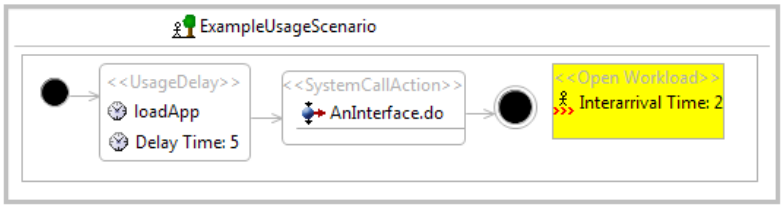


Fig. 2. Usage model.

describing the key performance-relevant elements of a component's behaviour. In particular, Fig. 1 shows that the control flow in our component may branch into either of three flows, with different CPU demands for each flow. Each branch is associated with a particular branch probability to indicate the likelihood of a particular branch being taken. This is the kind of information required to perform execution-time analysis on the component's behaviour as is standard in software performance engineering (see, e.g., [13]). In addition, we could model failure information to support reliability analysis.

Fig. 1 is only half the story. We also need to provide information about how the component is used to be able to provide useful predictions of performance. In Fig. 2, we see an example usage model specifying a particular workload for our component. This part of the model uses standard workload terminology to specify an open workload with an inter-arrival time of 2 time units. When a request arrives, there is a delay of 5 units loading the application, after which a call to our component is executed. With these models we now have enough information to run a first basic simulation of our system.

The Palladio Simulator offers the results of the analysis of performance and reliability of the system being analysed in different formats. For example, for the above model, it gives the mean response time and confidence intervals in Table 1. The chart in Fig. 3 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 90 time units.

Table 1. Palladio: results of Plain Batch Means Algorithm

Mean value:	41.97139713971397
Confidence value alpha:	0.9
Upper bound:	52.17187782288832
Lower bound:	31.770916456539624

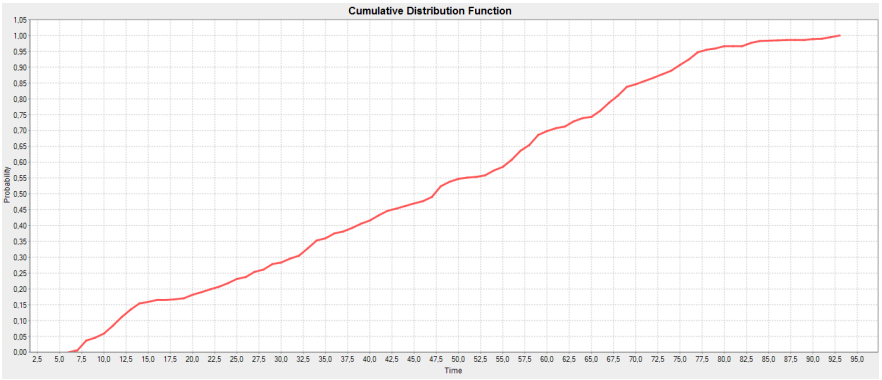


Fig. 3. Cumulative distribution function of the system's response time.

2.2 The e-Motions System

e-Motions [11] 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 behaviour 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 behaviour is specified with (graphical) in-place model transformations.

e-Motions provides a model of time, supporting features like duration, periodicity, etc., and mechanisms to state action properties. From a DSL definition e-Motions generates an executable Maude [4] specification which can be used for simulation and analysis. Other tools in the Maude formal environment, as its model checker or its reachability analysis tool, can also be used on this specification.

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. 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.

In e-Motions, there are two types of rules to specify time-dependent behaviour, namely, *atomic* and *ongoing* rules. Atomic rules represent atomic actions with a duration, which is specified by an interval of time. 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.

3 Palladio into e-Motions

The PCM is a DSL [3], and therefore we may define it in e-Motions. As for any DSL, the definition of the PCM includes its abstract syntax, its concrete syntax and its behavior.

Since the Palladio system has been developed following MDE principles, and specifically it is implemented using the Eclipse Modeling Framework (EMF), its metamodel may be directly used as abstract syntax definition of Palladio in e-Motions. Palladio models consist of several views, namely `UsageModel`, `System`, etc., corresponding to the different developer roles. These models are conformant to metamodels `Core PCM`, `StoEx`, `Units`, ... used by the different Eclipse plug-ins in the PCM Bench.[2]

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 PCM Bench uses to represent these concepts. This way, we maintain the PCM's look in the e-Motions definition.

The PCM Bench supports the design of the models corresponding to the different views that each developer role has to fill. However, these models define the architecture of a system. Transformations of PCM models into queueing network models or stochastic process algebra provide the necessary predictive analysis for the PCM models. Thus, the semantics of the properties to be analysed as well as of the analysis methods themselves are implicitly encoded in the transformations and support tooling.

In e-Motions, we describe how systems evolve by describing all possible changes of the models by corresponding visual rewrite rules, that is, time-aware in-place transformation rules. Since the PCM metamodel only specifies those concepts relevant for the PCM language and the models obtained from

[2] The metamodel provided to e-Motions must have a single package in a single file. Since the PCM metamodel is defined in several packages in several files, we have developed a higher-order transformation to prepare the input models.

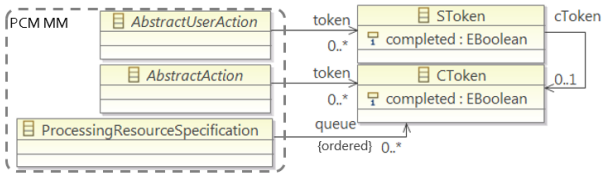


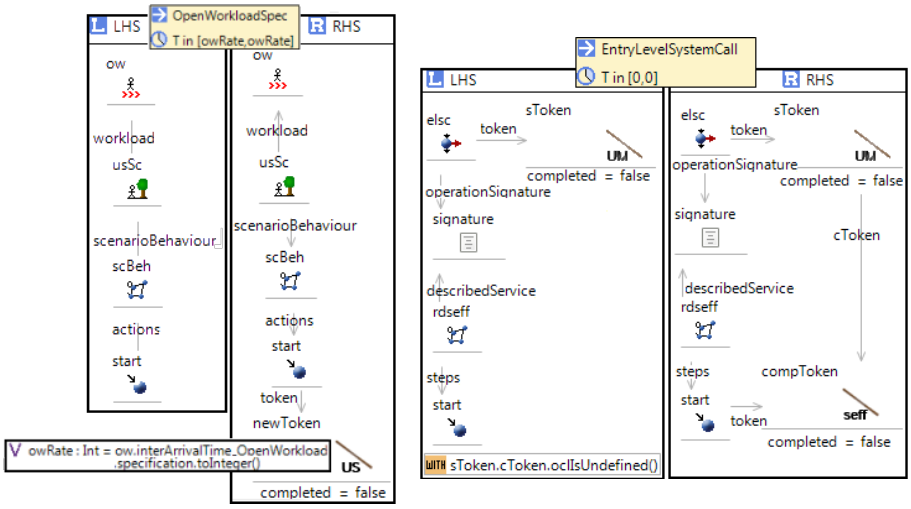
Fig. 4. Token metamodel.

the PCM Bench cannot be directly simulated or analyzed, we have conservatively enriched the PCM metamodel with new concepts to handle the control flow. We call this new metamodel Palladio*. Specifically, Palladio* has an additional metamodel `Token`, which includes two classes `SToken` and `CToken`. The former is specified at the system model (`UsageModel`) level, and the latter at the component model (`RDSEFF`) level. Both `SToken` and `CToken` classes have a *Bool* attribute `completed`, which states whether an action with this token is accomplished. References—with cardinality `*`— to classes `SToken` and `CToken` have been added to `AbstractUserAction` and `AbstractAction`, respectively. An ordered reference `queue` from `ProcessingResourceSpecification` to `CToken`, with multiplicity `*`, is used as a queue in which actions wait until resources of the corresponding type are available. Fig. 4 shows the `Token` metamodel and the references from classes of PCM to `SToken` and `CToken`.

We may visualize that the execution of a Palladio model has a token "moving around" such model. An action with a token has the control of execution — the `completed` attribute of a `Token` object becomes `true` once the action is completed, then it can be moved to its successor action. 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 — modelled by the rule in Fig. 6.

Since the extension of the metamodel has been done in a conservative way, every model conforming to the Palladio metamodel is also conforming to the Palladio* metamodel. As we will see in Section 5, this will allow us to take models generated in the PCM bench directly into e-Motions, and use them to perform simulations in the e-Motions definition of Palladio.

In Palladio, an open workload specifies system usage intensity with an inter-arrival time, i.e., the time between two user arrivals at the system, as a random variable with some probability distribution. It models an infinite stream of users arriving at a system, which execute their scenario, and then leave the system. Fig. 5(a) shows the `OpenWorkloadSpec` rule, which specifies the behaviour of a `UsageScenario usSc` with an `OpenWorkload ow`. When the rule is triggered, a new system token is added to the first action of the system, i.e., the `start` action. Moreover, the rule is fired every `owRate`, which is a local variable whose value is given by `ow`'s random variable.



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

Fig. 5. New request rule specification.

A `ScenarioBehaviour`, which is included in a `UsageScenario`, is composed of a set of actions, which can be `Start`, `Stop`, `EntryLevelSystemCall`, `Branch`, and `Loop`. These actions are modelled in e-Motions, since they are used to describe the behaviour of system components. Components are independently specified in Palladio, and can be instantiated from a `ScenarioBehaviour` by `Signatures`. The `EntryLevelSystemCall` action represents the invocation of a component.

The rule in Fig. 5(b) shows our definition of an `EntryLevelSystemCall` in e-Motions. If a (sub)-state matches its LHS, 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 rule in Fig. 6 shows the behaviour of an `InternalAction`, which represents the execution of an internal activity by a component service, possibly using some resources, like HDD or CPU. In Palladio, these executions present a high-level abstraction, and the resource demands are expressed as a single stochastic expression. The duration of the action depends on the parameters of the demanded resources. Resources are limited by the available number of resources of that type (`PRS.numberOfReplicas`). Tokens are served following an FCFS strategy by using a queue associated to each resource type. Only the first `PRS.numberOfReplicas` tokens in the queue `PRS.queue` get to be executed. Once an internal action is executed, its token is removed from the queue (`PRS.queue->excluding(t)`).

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

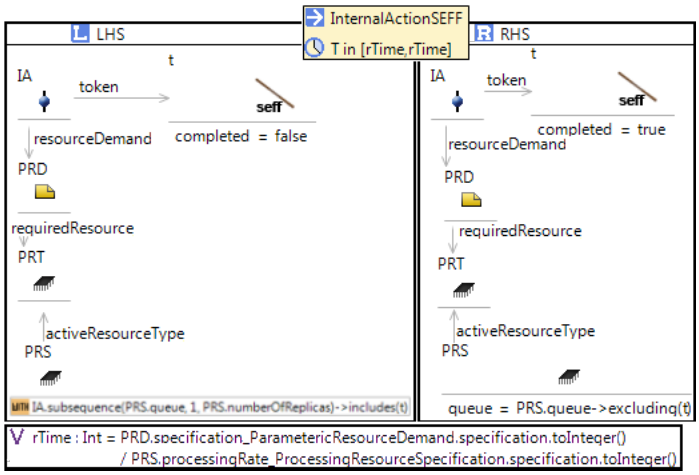


Fig. 6. Internal Action specification

Once the whole DSL has been defined, and given a model as initial state, it may be simulated by applying the rules describing its behaviour. 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.

4 NFPs by Observation

In previous work, we have proposed an approach for the specification and monitoring of non-functional properties using *observers* [15, 16]. They are objects with which we extend the e-Motions definition of systems for the analysis of NFPs by simulation, such as mean and max cycle times, busy and idle cycles of operation units, throughput, mean-time between failures, etc. We also explored in [6, 17] 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 these composition mechanisms to combine them. The result is that we can use the combined enriched system DSL to monitor NFPs of our systems.

We proved in [5] that, given very natural requirements on the observers and the instantiating mappings, the system thus obtained was a conservative enrichment of the original system, in the sense that the observers added *do not change the behaviour of the system.*

Given an e-Motions definition of Palladio as the one presented in Section 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 both those observers that monitor non-functional properties available in the Palladio Simulator as well as those that monitor other properties. The NFPs chosen can then be analysed by simulation.

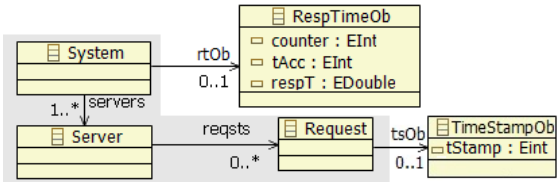
4.1 Generic Observers

We present in the first place a generic DSL for monitoring the *response time*, which is a property included in the analysis made by 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, the number of cars being manufactured in a production line, the number of passengers checking-in in an airport, etcetera. Given the description of a system, in order to measure response time, we basically need to register the time at which requests appear in the system, and the time at which they are completed. With this data and a simple calculation, we can easily get the response time.

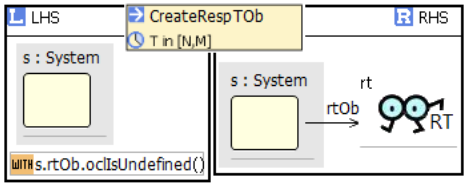
A generic DSL achieving this is shown in Fig. 7. Its abstract syntax (the metamodel in Fig. 7(a)) contains three generic and two concrete classes – generic classes are shown with a shaded background. `System`, `Serve` and `Request` are parameter classes to be instantiated by specific classes, as explained in Section 4.2. The `System` class represents the whole system, which is composed of a set of `Server`s. These, in turn, can have `Request`s to be processed. The class `RespTimeOb` represents the observer for measuring the response time mean. Its three attributes represent the number of requests already processed (`counter`), the accumulated time by them (`tAcc`), and the current average response time (`respT`). Note that there is yet another observer in this metamodel, `TimeStampOb`, used to store the times of incoming `Request`s.

The behaviour of this DSL is defined by the in-place transformation rules in Fig. 7, in which parametric concepts have no concrete syntax, they are depicted as boxes with 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. We see in its RHS that the observer is associated to the system. 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 class `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 `Request`s, while `tAcc` contains the addition of cycle times of all `Request`s, 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`.

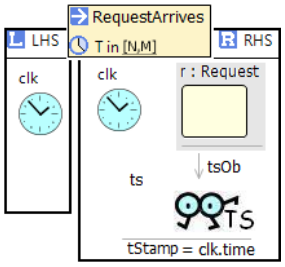
Fig. 8 shows a DSL for the *throughput* observer.Throughput can be defined as the average rate of requests processed by a system. Given the description of a system, in order to measure this property, we basically need to be able to count the number of processed requests, and calculate its quotient with time. The abstract syntax (metamodel in Fig. 8(a)) contains the same parametric



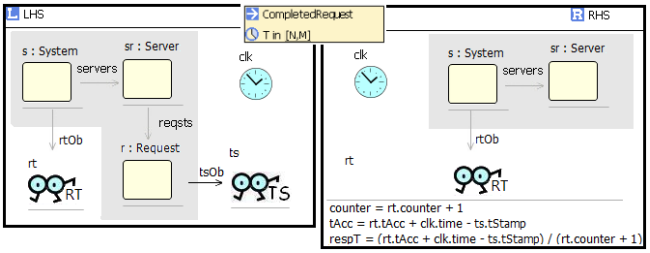
(a) Abstract syntax



(b) Behavior: `CreateRespTOb`



(c) Behavior: `RequestArrives`



(d) Behavior: `CompletedRequest`

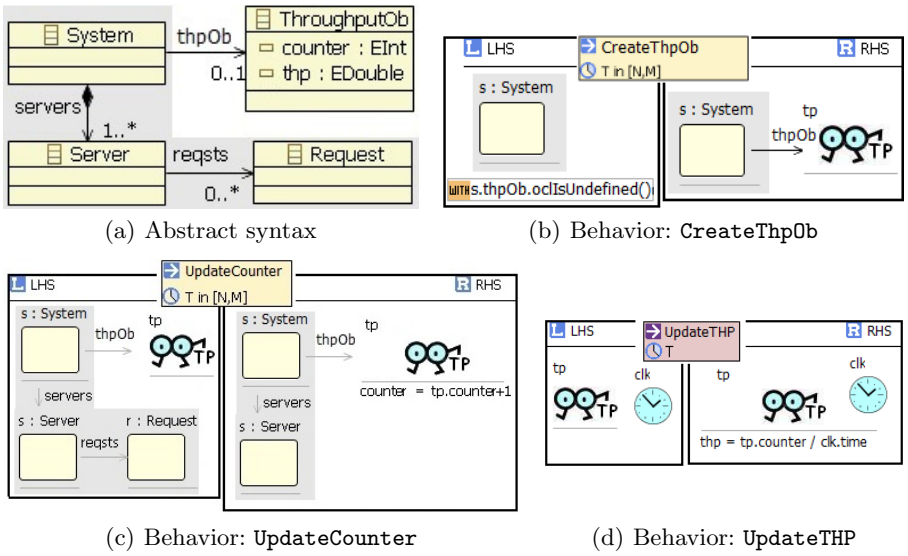
Fig. 7. Response Time observer DSL definition.

classes as the one for response time and the `ThroughputOb` class that represents the observer. The `counter` attribute stores the number of `Requests` that are completed, while `thp` is used to keep the actual throughput.

Its behaviour is also defined by three transformation rules. The `CreateThpOb` rule creates the observer, as the corresponding rule for the response time observer. Rule `UpdateCounter` increases the `counter` attribute of the observer every time a `Request` is served. Finally, we have an ongoing rule where the value of throughput is computed, which keeps the value `thp` updated as time evolves.

4.2 Adding Observers to System Specifications

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. This is done by defining a correspondences model [6, 10]. For example, for weaving the metamodel of



(a) Abstract syntax

(b) Behavior: `CreateThpOb`

(c) Behavior: `UpdateCounter`

(d) Behavior: `UpdateTHP`

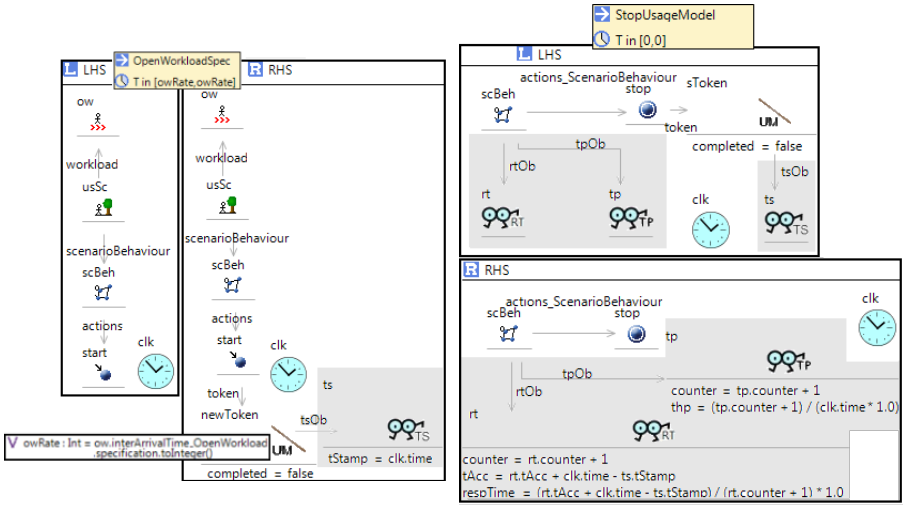
Fig. 8. Throughput observer DSL definition.

response time (Fig. 7(a)) with the metamodel of our Palladio implementation in e-Motions, the `System` class is mapped to the `ScenarioBehaviour` class, `Server` to `Start` and `Request` to `SToken`. The weaving of metamodels is quite straightforward, and we do not show the resulting metamodel due to space limitations. Let us focus here on the weaving of rules.

Regarding rules, we basically need to map each rule in the source DSL to a rule in the target one. The mapping defined for the metamodel does most of the rest. Rule `RequestArrives` (Fig. 7(c)) is woven with the `OpenWorkloadSpec` rule of our Palladio system (Fig. 5(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. 7(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 behaviour enriched with the observer objects. Figs. 9(a) and 9(b) show the rules `OpenWorkLoad` (Fig. 5(a)) and `stop` as resulting from the weaving process.



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

Fig. 9. Woven rules.

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.

5 Evaluation

Once the e-Motions definition of the Palladio DSL has been enriched with the desired observers, we may use it for analysing its performance by simulation. More specifically, since the Palladio* metamodel is a conservative enrichment, we may take models designed in the Palladio Bench and load them into e-Motions for simulation using the e-Motions definition of Palladio. The information in the observers can be accessed when the simulation has completed.

Following this procedure, we have simulated the Palladio model presented in Section 2 in the e-Motions definitions of Palladio, whose results are summarized in Table 2 (for a simulation of 1000 tasks). We can observe that the value obtained for response time is coherent with the one obtained in Palladio (cf. Table 1), since the e-Motions' value fits within the confidence interval returned by Palladio. Fig. 10 shows the cumulative distribution function for the simulation in e-Motions, while Fig. 11 shows the response time as a function of the time when a request entries the system, based on the e-Motions output. Since the queues get saturated, response times keep increasing.

Table 2. Case study's e-Motions results

Mean System Response Time	43.6626 seconds
Throughput	0.4804 seconds

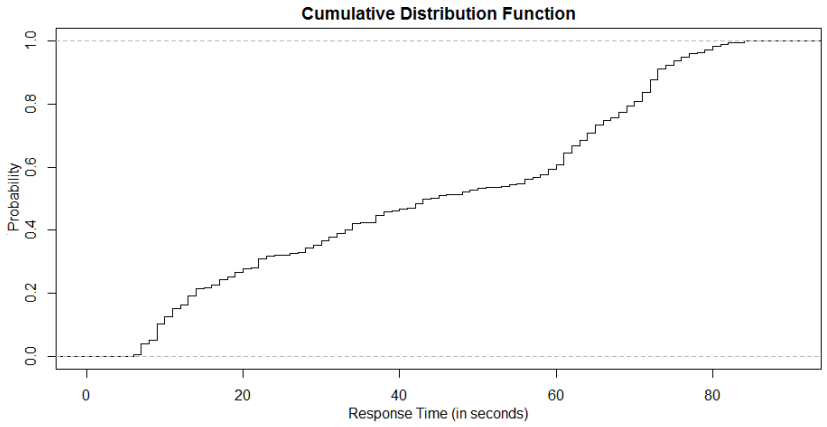


Fig. 10. Cumulative distribution function for the simulation in e-Motions.

6 Conclusions and future work

Non-functional properties of software, such as performance, reliability, or security, can determine success or failure of software systems. It is therefore important to be able to provide estimates of these properties as early as possible in the development process. Model-driven engineering has been viewed as a promising technology for addressing this problem because of its ability to transform early design models into analysis models. However, the semantics of the properties to be analysed as well as of the analysis methods themselves are typically encoded implicitly in the transformations and support tooling. Often, these encodings tangle semantics for multiple properties to be analysed. As a result, it becomes difficult a) to add new properties and analyses and b) to validate the transformation and analysis implementations themselves.

We have addressed this 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 analysed have been encoded as separate, parametrised 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 currently provide support for key Palladio features for the definition of usage models (start, stop, delay, and entry level system call) and component

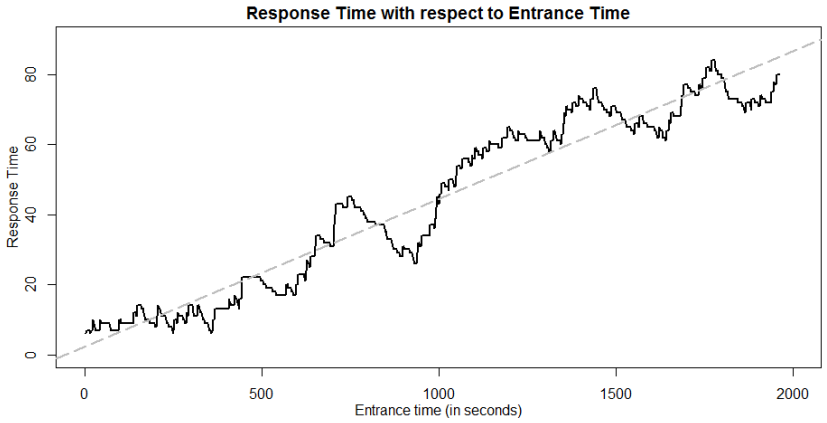


Fig. 11. Response Time obtained from e-Motions output.

models (start, stop, branch with any number of probabilistic branches, internal action, and CPU specifications). Currently, we only have partial support of stochastic variables. Their full support is left as future work.

We have demonstrated two main benefits of our approach: 1) The semantics of the simulation and the non-functional properties to be analysed 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 incorporate additional features to our definition of Palladio, as, e.g., 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. For instance, in [17], we showed how to maintain the value of cycle time around a specific goal. The dynamic system consisted of a production line where machines had two modes of processing parts: fast and slow. In this case, when the cycle time of parts was higher than the goal, the speed of the machines was increased. The opposite occurred when the parts were produced too fast. This self-adaptive behaviour was achieved by consulting the value of the cycle time observer during simulation.

Acknowledgments. We thank Samuel Kounev for his help in getting access to the Palladio internals. This work is partially funded by Project TIN2011-23795, by U. de Málaga, Campus de Excelencia Intl. Andalucía Tech, and by the EU under the ICT Policy Support Programme (grant no. 317859).

References

1. Balsamo, S., DiMarco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (May 2004)
2. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of component-based systems: A survey from an engineering perspective. In: Dagstuhl Seminar 04511: Architecting Systems with Trustworthy Components. LNCS 3938. Springer (2006)
3. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the Palladio component model. In: Proc. 6th Int'l Workshop on Software and Performance (WOSP'07). ACM (2007)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS 4350. Springer (2007)
5. 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. LNCS 7841. Springer (2013)
6. Durán, F., Zschaler, S., Troya, J.: On the reusable specification of non-functional properties in DSLs. In: Czarnecki, K., Hedin, G. (eds.) Proc. 5th Int'l Conf. on Software Language Engineering (SLE'12). LNCS 7745, pp. 332–351. Springer (2013)
7. Fritzsche, M., Johannes, J., Zschaler, S., Zherebtsov, A., Terekhov, A.: Application of tracing techniques in model-driven performance engineering. In: 4th ECMDA Traceability Workshop (2008)
8. Grassi, V., Mirandola, R.: A model-driven approach to predictive non functional analysis of component-based systems. In: Proc. Workshop on Models for Non-functional Aspects of Component-Based Software. (2004)
9. Happe, J., Koziolek, H., Reussner, R.: Facilitating performance predictions using software components. IEEE Software 28(3), 27–33 (2011)
10. Moreno-Delgado, A., Troya, J., Durán, F., Vallecillo, A.: On the Modular Specification of NFPs: A Case Study. In: Proc. of XVIII JISBD, pp. 302–316 (2013)
11. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proc. of VL/HCC'09. IEEE (2009)
12. Röttger, S., Zschaler, S.: Tool support for refinement of non-functional specifications. Software and Systems Modeling journal (SoSyM) 6(2), 185–204 (Jun 2007)
13. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Object-Technology Series, Addison-Wesley (2002)
14. 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 7347, pp. 388–397. Springer (2012)
15. 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)
16. Troya, J., Vallecillo, A.: A domain specific visual language for modeling power-aware reliability in wireless sensor networks. In: Proc. of NFPinDSML'12, pp. 3:1–3:6, ACM (2012)
17. 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)