# PERFORMANCE PROTOTYPING - GENERATING AND SIMULATING A DISTRIBUTED IT-SYSTEM FROM UML MODELS

ANDREAS HENNIG, ANJA HENTSCHEL and JAMES TYACK
*Siemens AG, Corporate Technology, CT SE 1,*
*Otto-Hahn-Ring 6, 81739 München, Germany,*
*Andreas.Hennig@siemens.com, Anja.Hentschel@siemens.com, jamesahtyack@hotmail.com*

***ABSTRACT:*** In this paper, we present the concept of "performance prototyping" – the automatic generation and deployment of small components emulating the intended behaviour of real components under design into real IT-infrastructure and environments. Allowing far more effective and consistent production of prototypes than manual prototyping, performance prototyping enables the designer of systems and their infrastructure to assess the impact of various load scenarios, design choices and configuration alternatives very early in the project, and thus allows to synchronize infrastructure planning and system development closely. Rather than the "build first – tune, change & upgrade later" approach, performance prototyping enables to design, plan and build hard-, soft and middleware in closer coordination and to meet performance targets in fewer cycles.

The basic concepts of the UML-based notation of performance aspects is presented which was designed to be compatible with current UML-tools and fit into their normal usage in development practises. We then discuss the interaction and differences of performance prototyping ("in-vivo performance simulation"), performance prediction in dedicated methods and tools ("in vitro" performance simulation) and load-testing as well as the differences to manual prototyping and benchmarking. A method of integrating performance prototyping into commercial UML-tools is presented, particularly with view on the challenges of generating multi-target and multi-protocol prototypes that interact across targets, platforms and protocols in the way prescribed by the model. We then describe the model, prototype, experiments and findings based on a JSP example before the conclusion of the paper.

***KEYWORDS:*** Software Performance Engineering, UML Modelling, Performance Annotation, Performance Prototype Generation, Deployment, Simulation, Load Testing, Benchmark

## INTRODUCTION

Larger IT-Systems or products, particularly if distributed and networked often have complex interactions between the various **hardware** (HW) entities (hosts, network nodes and links, peripherals), the different layers of operating system, execution environments and server processes (like web, servlet, application or database-server) which in this paper we will summarily call **middleware** (MW), and finally the main behavioural software components implementing the **business logic** of the system (SW). With the trend to standardized off-the-shelf products with standard interfaces and protocols, the division between the teams responsible for "SW-development" and "HW planning, installing configuration and operation" tends to be somewhere within the middleware layers, where the "SW-team" focuses on functionality and interfaces, the "HW-team" on configuration issues.

For most large systems (e.g. ERP systems, intra/internet-portals, online shops, information and control systems…), performance is a central criteria and critical success factor. With increasing expectations of the users to perceived performance, an unsatisfactory performance might - and frequently does - endanger the system's/product's/project's success irrespective of functionality and design. Since these systems are often business critical and/or highly image critical, insufficient performance can incur heavy costs (e.g compensation, penalties, superfluous hardware, loss of market shares and value), delay the going-live, and reduce the system's benefit (e.g. through lack of user acceptance and retention, sub-optimal decisions based on out-of-date information). Performance problems can derive from a variety of sources, including sub-optimal configuration, insufficient computational power, inefficient implementations of individual modules and design flaws. The earlier lie within the responsibility of the HW-team and can be rectified by tuning or – although more expensive – by additional HW. The later are not only more difficult


 *ISBN 3-936150-25-7*

to detect, they are also caused much earlier in the project and are therefore far more difficult to correct in time if detected towards the release date – apart from the much higher cost.

A reason behind the numerous performance failures of IT-Projects (and the ensuing mutual accusations) is the far-too-late assessment of performance, which derives partly from lack of performance-awareness, partly from the restrictions found in the predictive methods that could be used to measure and control progress in terms of performance goals. Ideally, for large and performance-critical systems, there should be the role of an overall "performance engineer", who gathers performance assumptions and requirements, predicts overall final performance based on the current implementation progress, coordinates and mediates between the conflicting interests of HW and SW teams and executes in-development and pre-release load-test to substantiate development and release decisions.

We group the methods predicting the live performance coarsely into "benchmarking", "simulation", "prototyping" and "load-testing" (ref. Fig. 1):

"**Benchmarking**" employs small standardized activities (e.g. integer of floating point operations, memory or disk access…) and measures how many a given system can execute per second. While these "synthetic" benchmarks accurately describe one performance aspect of a given HW (and thus help to compare between a larger number of HW/MW alternatives), they - inherently – do not measure the performance in terms of the transaction types of the intended system (for the purpose of this paper, we include specialized application-specific benchmarks under prototypes, below). Benchmarks thus provide a basis for HW-choices, but can only rarely be used for good estimates of the new system's performance.

"**Simulation**" builds an abstracted model of the infrastructure (HW and MW), the behaviour of the business logic as well as the expected load from internal sources and users. The resulting performance aspects are obtained using various methods (e.g. queuing networks, stochastic methods, discrete event simulation...) of the software performance engineering domain (SPE, [Smith90]). Since the models are built and evaluated in an environment completely separate from the physical system (somehow "in vitro"), Simulation can be applied even before the first HW is purchased. However, a common problem of simulation lies in the need to build complex models (where particularly the MW-models depend on product release-cycles in a fast paced industry), to ascertain numerous model parameters and to – often manually – transfer the design of the intended system into a suitable representation. The resulting uncertainties and potential inconsistencies as well as the effort and time required restrict the use of simulation to (parts of) systems with clear boundaries that can be abstracted easily and reliably and have focused performance inquiries.

Manual "**Prototyping**" is normally performed in the early phases to establish suitability of a platform/ middleware/ technology under consideration for the intended purpose, i.e. often functionality and interoperability orientated. The prototypes can then be load-tested, but since the effort required to manually develop prototypes restricts their

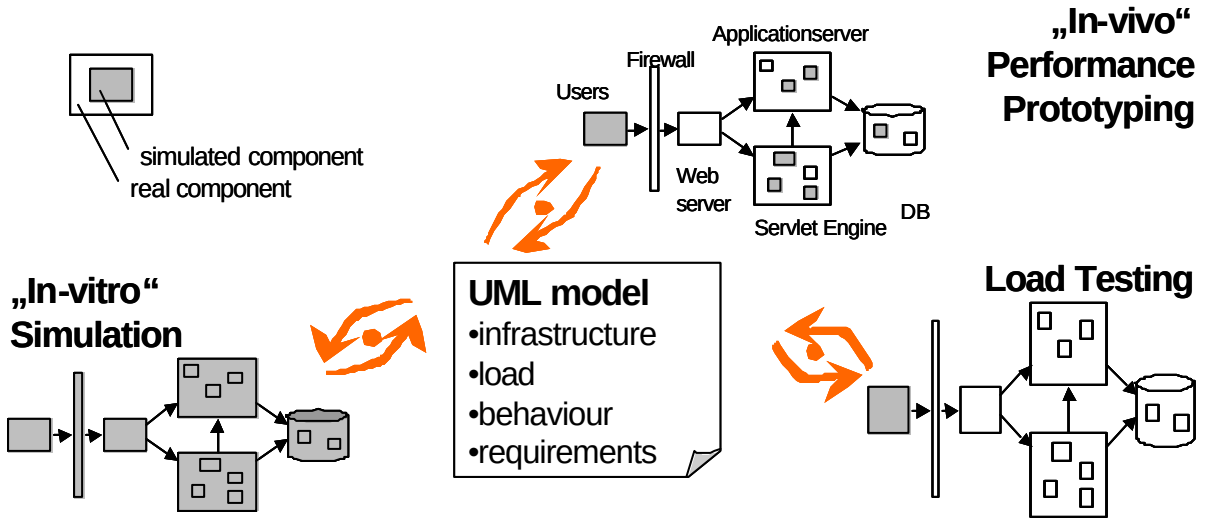



***Fig. 1: One model throughout performance development lifecycle – integrating prediction, prototyping and acceptance testing by varying real and simulated components***

comprehensiveness and variability, the results are at best a basis for extrapolation. Manual Prototypes differ from benchmarks since they measure small application-specific activities, but suffer from the same necessary restriction to few aspects. Furthermore, manual prototypes also bear the danger of errors, inconsistencies and incompatibilities.

"**Load-testing**" transforms the expected user behaviour into small simulated "virtual users" which are deployed onto real infrastructure (sometimes called "load-generators" or "load-injectors"), which exercise a real system or prototype with real and realistic load. Load-tests are either used as gruelling acceptance tests or during development to test performance aspects of individual modules or prototypes. Load-testing a finished system is the least predictive method discussed – it is however, predictive in terms of the anticipated user behaviour, which might vary greatly from the behaviour of real users.

We therefore propose automated "**Performance Prototyping**" as a means to overcome the above limitation of flexibility, coverage, efficiency and application-specificness. For this, models of the system's intended business logic and planned infrastructure are maintained in UML. Since these documents normally need to be produced during development in some way, they only need to be annotated with some additional information like resource consumption, performance requirements and deployment and access locations.
This furthers

a) understanding and consistency between the development teams and the performance engineer,
b) reduces the effort required to build the main model (ideally now through the teams themselves),
c) provides a simple and concise notation of performance aspects and
d) permits to automatically generate and deploy comprehensive prototypes ready for load-test.

Unfortunately, the UML is currently not always used to document the entire system, particularly the infrastructure and HW/MW aspects thereof. But even if the few relevant HW parts need to be transformed into UML based on the input of the HW-team, above benefits still apply. From the model, a performance prototype is generated automatically and can be deployed on real target systems, where it can be tested "in-vivo". Due to the fast cycle times with automatic performance prototypes, various alternatives in HW, MW or SW can be investigated with affordable effort.

In the remainder of the paper, we shortly present the information required for performance prototypes and how and where they can be modelled in UML based on a JSP example. We then describe the architecture of the prototype generator and how the required flexibility can be achieved. After the presentation of some experimental results, we give an outlook on synergies and interactions between performance prototyping, simulation and load-testing before the conclusion of the paper.

## MODELLING PERFORMANCE PROTOTYPES

A Performance prototype requires the description of the characteristics that are (or could be) performance-relevant. Programming language and execution environment certainly affects the performance (e.g. C++ being faster than VBA, or tomcat generally being faster than JServ) as does the interaction of components (a calls b calls c). The precise values within the requests (e.g. a lookup-key) or the responses (e.g. the retrieved data) do hardly affect performance, while the size, encoding and protocol of requests and responses are likely to have an impact. Performance-relevant information consists of infrastructure, load, behaviour and requirements information.

We model **infrastructure** information in deployment diagrams as the obvious representation in UML (see also [Williams98], [Dimitrov02], [Mirandola00], [Petriu99]). The diagram describes computational resources, their connection and – optional – the number of instances of a component, which we call its multiplicity according to the corresponding UML attribute. In Fig. 2, we show a sample deployment of two webserver hosts, both hosting a JSP-engine; webserver1 hosts additionally a database. A LAN component (used to represent a bus-topology in UML) connects the servers to two types of clients, which differ in the numbers of browsers running on it. The multiplicity of Client1 indicates, that there are $numClients instances (e.g. 20 in different network locations) in the system.
In addition to the core SPE notation as proposed in [Hennig02], performance prototyping requires additional information about the real hosts and servers used (e.g. IP numbers and ports of the webservers). Mainly, this is the "access" information, which denotes how a component is addressed for requests. In the deployment diagram, the access-path can be annotated as the tagged value "spe.ppr.access" of the nodes or objects. For webservers, this would simply be the URL of the JSPs themselves, which could include parameters,

port, username and password as well [RFC 1738, 1808]. For a database, the access-path would for example contain the JDBC connection information. Since performance prototyping aims to automatically deploy the components, further information is needed to specify where and how to deploy the component. The tagged value "spe.ppr.upload" therefore contains a URL that indicates the deployment destination. In the case of a JSP component spe.ppr.upload points to a file:// location or the URL to a cgi-script into which the JSP-source code can be uploaded.

The **behaviour** of a prototype system as well as the **load** placed onto it is described as the generation and exchange of messages, requests and responses, which we model in the sequence diagram. For the discussion concerning the use of UML in general and the use of state vs. sequence diagram in particular, see [Hennig01] and [Hennig02]. In Fig. 3 a simple interaction pattern ("workflow") is depicted, several instances ("jobs") of the same or different workflows can occur concurrently.

The load is generated by the actor, representing multiple users that execute the same workflow with given arrival and think times. In the example in Fig. 3, the webbrowser is modelled to submit various http-request per "click" of the actor (e.g. for nested or consecutive http like redirecting, frames, included image). Since the "browser" corresponds to different client machines in the deployment diagram, we indirectly model the network region, where the load

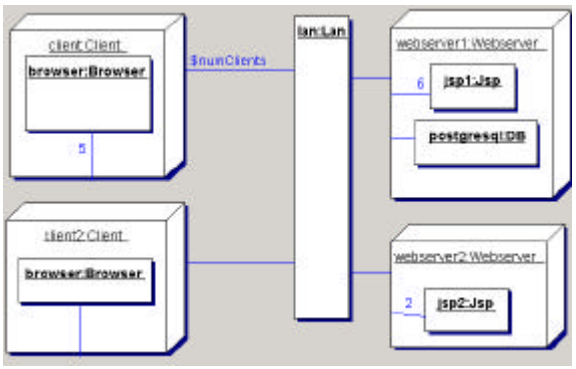



***Fig. 2: UML deployment model of the prototype***

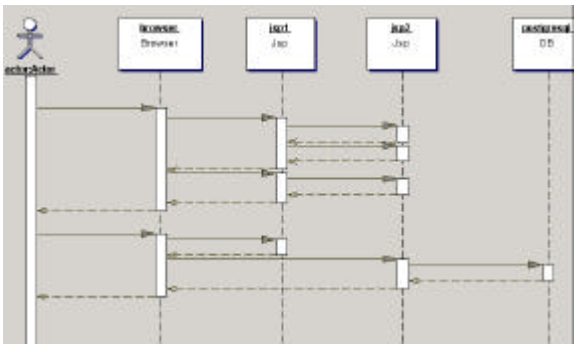


***Fig. 3: UML behavioural model of the prototype***

should originate.
Parameters passed along the http-requests will inform JSPs which step of which workflow they are expected to execute. The generated JSP code contains the information how to execute a specified step (i.e. how much computation is needed, how large the response will be, which other components need to be called). Resource usage can be modelled flexibly (in a spe.use.{resourcetype} tagged value) and currently includes but is not limited to time delay, cpu consumption, memory usage, I/O-volume and various types of semaphores. Since cpu-consumption depends on the speed of the hosting hardware, we specify it in number of iterations of classical benchmark operations like dhrystone, whetstone or of less formal but expansible operations like string-operations or heap-sorting.

Performance **requirements** like the maximum permissible response time for certain requests or the time to complete an entire workflow can be denoted as numerical expressions based on timestamps collected during the execution of the jobs. The timestamps as well as additional job-specific variables (e.g. a randomly chosen think-time of the simulated user) can also be used to gather workflow-related statistics. Infrastructure-related statistics like cpu-usage can be gathered and evaluated using network and performance management tools (e.g. based on SNMP, rstat or the windows performance monitor)

## GENERATING PERFORMANCE PROTOTYPES

After modelling infrastructure, load, behaviour and requirements in various UML diagrams, an experiment definition diagram is used to specify the subset of "investigated diagrams" and overall parameters like a scalability factors for number of clients (e.g. $numClients in. Fig. 2), the workload or an overall think time.

A series of scripts can then be started directly from the UML-Tool (currently TogetherJ from Togethersoft) which controls the prototyping cycle depicted in Fig.4. The prototyping is integrated transparently in order to ensure user acceptance and achieve high impact by frequent use of the method through seamless integration of the end-to-end process. The selected diagrams and required information is extracted from the UML-tool and stored in an intermediate XML representation. From this experiment description, a converter produces the prototype parts for the various target platforms and deploys them into their respective environment. The User behaviour (the load characteristics from the

actor and in our example also the html-nesting logic

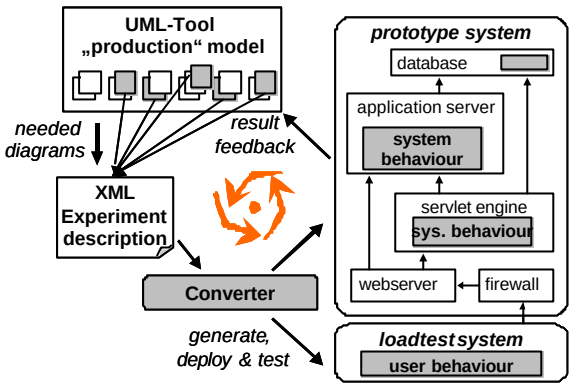



***Fig. 4: End-to-end performance prototyping cycle from UML-tool***

of the browser) results in a script for a commercial load-testing tool (LoadRunner from Mercury Interactive). From the behavioural description of the JSPs, the source code is generated in JSP syntax and uploaded into the JSP directories of the respective servlet-engines. Since in our example, the database request is a simple SQL statement and not a stored procedure, there is no need to generate code for the database as we can include the statement into the generated JSP code. The used tables, however, need to exist in the database.

The results obtained from internal statistics (e.g. response times) and network monitoring could be fed back into the UML model. Since commercial monitoring tools often provide specific analysis modules (e.g. drill-down or regression), this step might be performed in a specialized separate tool. After analysing and interpreting the data, the UML-Model can be updated and modified accordingly and the cycle started again.

The challenge of generating the prototypes lies in the potentially heterogeneous target platforms (programming languages, execution environment) and the communication protocol they use. Ultimately, each implementation platform should be able to issue requests to any other (sensible) type of platform using a number of possible protocols. Over these protocols, the control information of the prototype (e.g. workflow name, instance and current step) needs to be transmitted without altering the protocols. While flexible protocols like http, where additional parameters can easily added to the URL without interferences, easily accommodate for this, more rigid protocols like SOAP or RMI will be more challenging.

## EXAMPLE PROTOTYPE & EXPERIMENT

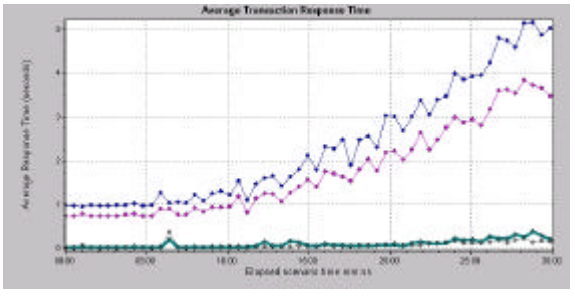
For experimental evaluation, we used the above simple behaviour and varied the deployment configuration by altering the host on which the JSP and database components were deployed to (ref table 1). Both hosts run under Linux, dax is a dual-processor server, ibex a single-processor workstation.

| | JSP1 | JSP2 | DB |
|---|---|---|---|
| Test A | Dax | Dax | Dax |
| Test B | Dax | Ibex | Dax |
| Test C | Dax | Ibex (tuned) | Dax |
| Test D | Dax | Dax (tuned) | Dax |

***Table 1: testing deployment variations***

In the sequence diagram, we specified the first call from the browser to jsp1 to be resource intensive (e.g. for analysing user authorization) as well as the processing of the database results in jsp2. We defined a think time of 3 seconds on average, which represent the time a user would need between clicks in the browser. For tests C and D, we assumed a scenario where a proposed tuning measure is expected to improve database processing by 60%, but since it would entail large modification efforts, an impact analysis should be carried out before any decision is taken. Resource consumptions are therefore adjusted in the sequence diagram, the prototype newly generated, deployed and tested to provide the answer in short time.

Each load test ran for 30 minutes and increased the load every 2 minutes by one additional simulated user. Fig 5. shows the achieved rate of fully completed transactions per second (TPS, number of finished workflow instances/s) of Test A. The system went into saturation after 18 minutes with an approximate capacity of 1.05 TPS caused by 10 concurrent simulated users (Fig. 5). Further users did not increase the transaction rate but only resulted in increased response times due to shared use of the CPU resources.
The response times for the first user (on a basically idle system) where in sum 1.8s without the prescribed think times, but rose to 5.5s at the saturation point of the system. Fig. 6 shows the



response times of the four constituent requests from the browser. The result for the further tests is shown in table 2.

| | Capacity | | Response time | |
|---|---|---|---|---|
| | TPS | Users | "idle" | saturated |
| Test A | 1.05 | 10 | 1.8s | 5.5s |
| Test B | 0.35 | 5 | 2.8s | 5.9s |
| Test C | 0.58 | 7 | 1.9s | 6.6s |
| Test D | 1.33 | 13 | 1.3s | 4.7s |

***Table 2:Performance Measurement of the prototype***

We can see, that – not surprisingly – deploying parts of the application on ibex did not improve the overall capacity of the system. The proposed improved database processing step, however, would significantly improve the overall performance in terms of capacity (by 26%) and response times (28%-36%) and should therefore be attempted.

Once the initial model was built, each complete performance prototyping cycle, i.e. evaluation of a further design variation, was completed in less than an hour. The generation and deployment of the prototype itself took about two minutes, the remainder being spent on running and analysing the load-test. This allows assessing even small design choices for their impact on performance. Tests C and D also demonstrated a possible further application of performance prototyping – setting performance targets for each step of the intended workflows based on the expected impact on the entire system. This "budgeting" of resources and planning of performance could help to coordinate the viewpoints of owner, designer, developer and operator of a distributed system.

**OUTLOOK AND CONCLUSION**

In this paper we presented the ideas and principles behind performance prototyping as well as the integration and deployment concepts. In our example, we showed a basic web-application and demonstrated, how a simple and fast evaluation of a performance prototype could be used to assess the suitability of different design variants.

We are currently working to expand the method to include further platforms (e.g. EJB, ASP, .NET) and protocols (e.g. SOAP, RMI) to support prototypes of more heterogeneous systems.

In our view, the major obstacles in the way of widespread application of software performance engineering are

a) insufficient familiarity of SW-developers with SPE methods,
b) separate, potentially inconsistent and contradicting notation and interpretation of SPE and SW-models and
c) the large time and effort needed to obtain results.

We expect that by overcoming these obstacles, the concepts and methods of SPE would bring a large benefit into SW-engineering. SPE could then contribute more than currently towards SW-products and systems that have better performance at lower development costs and shorter development time. With our UML-based notation we aim to contribute towards a more intuitive modelling of performance aspects in standard UML in mainstream tools. Our works around simulation [Hennig02] and performance prototyping as SPE methods show the flexibility and wide range of the notation. The possibility to evaluate different scenarios fast, consistently and efficiently allows for close interaction of predictive methods like simulation, benchmarking, load testing and performance prototyping. Simulation will be invaluable for extrapolation in the dimensions of scalability, reliability and optimisation. Benchmarking can provide basic measures; performance prototyping can assess specific infrastructures for specific load scenarios. Projecting the findings onto larger planned server farms or networks could again be the contribution of simulation based on the parameters and findings obtained through multi-varied performance prototyping.

At the ESM 2003 we will give a presentation of the integrated end-to-end process of performance prototyping.

**REFERENCES**


[Dimitrov02] Dimitrov E., Schmietendorf A., Dumke, R.: *"UML-based Performance Engineering Possibilities and Techniques"*, IEEE Software, Darmstadt, p. 74-83, Vol 19/1, Jan/Feb 2002

[Hennig02] Hennig, A. R. Wasgint; "Performance Modeling of Software Systems in UML-Tools for the Software Developer", in *Proceedings of European Simulation Multiconference ESM'2002*, Darmstadt, Germany, 2002

[Hennig01] Hennig, A.; Eckardt, H., 2001, "*Challenges for Simulation of Systems in Software Performance Engineering*", in Proc. ESM'01, Prague, p. 121-126, 2001

[Mirandola00] Mirandola, R., Cortellessa, V.; 2000, "*UML Based Performance Modeling of Distributed Systems"*, UML 2000 - 3rd Int. Conf., York, UK, 178-193, 2000

[Petriu99] Petriu, D., Wang, X. „*From UML descriptions of High-Level Software Architectures to LQN Performance Models*", Proc AGVTIVE'99, Springer Verlag, LNCS 1779, p47-62, 1999

[RFC1738]: Berners-Lee, T., Masinter, L., and M. McCahill, Editors, "Uniform Resource Locators (URL)", December 1994.

[RFC1808]: Fielding, R., "Relative Uniform Resource Locators", June 1995.

[Smith90] Smith, C.U., 1990. "*Performance Engineering of Software Systems"*, ISBN 0-201-53769-9, Addison-Wesley, Reading, US, 1990

Togethersoft Corporation, http://www.togethersoft.com/

Mercury Interactive Corporation, http://www.mercuryinteractive.com/

[Williams98] Williams, L.G., Smith C.U., "*Performance Evaluation of Software Architectures*", WOSP 1998, p 164.177, 1998