A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures

Samuel Kounev, Nikolaus Huber, and Fabian Brosig,
University of Würzburg

Xiaoyun Zhu, Futurewei Technologies

Results of a five-year research project and several industrial collaborations have produced tools that model the individual effects and complex dynamic interactions between an IT system's application workload and resource contention at multiple levels in the execution environment. An evaluation shows significant resource efficiency gains without sacrificing the performance specified in service-level agreements.

Modern IT system architectures are becoming increasingly distributed, have loosely coupled services, and are often deployed on virtualized infrastructures that abstract physical layers to improve system efficiency. The benefits of distributed architectures and virtualized infrastructures come at the cost of higher system complexity and dynamics; the inherent semantic gap between application-level metrics and resource allocations at the physical and virtual layers significantly increases the complexity of managing end-to-end application performance.

To tackle this challenge, techniques for online performance prediction are needed that enable the continuous prediction of three performance aspects: application workload changes, the effects of these changes on system performance, and the expected impact of possible adaptation actions. Online performance prediction can be the basis for designing systems that proactively adapt to changing operating conditions, thus enabling *self-aware* performance and resource management.[1] (See the "Self-Aware Computing Systems" sidebar for more information.)

We have developed a model-based approach to designing self-aware IT systems along with the Descartes Modeling Language (DML),[3] an architecture-level language that is central to online performance prediction and proactive model-based system adaptation. We have applied our model-based design approach in several case studies with realistic environments and in cooperation with industrial partners.[4,5] In an evaluation against a trigger-based approach (which relies on custom metrics and specified thresholds to execute predefined reconfiguration actions), our approach maintained acceptable

SELF-AWARE COMPUTING SYSTEMS

The consensus at the 2015 Dagstuhl Seminar 15041 (www.dagstuhl.de/15041) was that self-aware computing systems have two main properties. They

- *learn models*, capturing *knowledge* about themselves and their environment (such as their structure, design, state, possible actions, and runtime behavior) on an ongoing basis; and
- *reason* using the models (to predict, analyze, consider, or plan), which enables them to act based on their knowledge and reasoning (for example, to explore, explain, report, suggest, self-adapt, or impact their environment)

and do so in accordance with *high-level goals*, which can change.[1]

A major application domain for self-aware computing is the runtime management of modern IT systems.[1] In this context, an IT system is considered self-aware if it possesses three properties or can acquire them at runtime—ideally to an increasing degree:

- *Self-reflective*—is aware of its software architecture and execution environment, the hardware infrastructure on which it runs, and its operational goals, such as performance requirements.
- *Self-predictive*—can predict the effects of dynamic changes, such as changing service workloads, and of possible adaptation actions, such as adding or removing resources.
- *Self-adaptive*—proactively adapts as the environment evolves to ensure that it always meets its operational goals.

For the most part, existing research and industrial approaches do not address these properties. Most state-of-the-art industrial approaches for performance and resource management, like Amazon EC2 or Microsoft Windows Azure, are rule-based or heuristics-driven and have custom triggers. However, application-level metrics, such as response times, normally exhibit a nonlinear relationship to system load and typically depend on the behavior of multiple virtual machines (VMs) across several application tiers. Therefore, it is hard to determine general thresholds for firing triggers to enforce service-level agreements at the application level, which violates the self-predictive and self-adaptive properties.

Most research approaches to performance and resource management are based primarily on coarse-grained performance models that typically abstract

resource efficiency and avoided 60 percent of service-level agreement (SLA) violations.

DESCARTES MODELING LANGUAGE

Figure 1 is a high-level structural diagram of DML (http://descartes.tools/dml), which consists of five metamodels (from the bottom up): resource landscape, application architecture, usage profile, adaptation points, and adaptation process.

Resource landscape metamodel

The resource landscape metamodel describes the structure and properties of both the physical and logical resources that make up the IT system infrastructure. A common pattern in modern IT infrastructures is the nested containment of system entities: for example, data centers contain racks, racks contain servers, servers typically contain a set of virtual machines (VMs), a VM contains an OS, an OS can contain a middleware layer, and so on. DML provides constructs to model this hierarchy of nested resources as well as their internal configuration.

In Figure 2, the core elements of the resource landscape metamodel are described as a Unified Modeling Language (UML) class diagram.

A `CompositeInfrastructure` entity can be nested inside another `CompositeInfrastructure` entity, which might be used to model the nesting of datacenter resources (for example, datacenters contain server racks consisting of server and storage nodes). The central element of each `CompositeInfrastructure` entity is the abstract entity `Container`, which has a containment relation to the `RuntimeEnvironment` entity. The latter is also a `Container` entity that can contain additional `RuntimeEnvironment` entities. Thereby, it is possible to model the container nesting (OS, virtualization platform, and middleware). To further specify the resource-configuration properties of a `Container` entity, `Container` refers to a `ConfigurationSpecification`. Finally, the metamodel provides the `ContainerTemplate` entity to ease the modeling of containers with similar configurations.

Application architecture metamodel

We modeled the system's application architecture after the principles of component-based software systems.

systems and applications at a high level.[2–4] As such, they cannot exhibit the self-reflective property because they do not explicitly model the software architecture and execution environment and therefore cannot distinguish performance-relevant behavior at the virtualization level versus at the level of applications hosted inside the running VMs. Their self-prediction capabilities are limited, which makes them ill suited for complex scenarios, such as predicting how application-workload changes propagate through the system architecture to the physical resource layer or the effect on different services' response times of migrating a VM in one application tier to a different host type.

In autonomic computing, software models play an important role in managing complexity and supporting adaptation decisions.[5,6] However, existing model-based approaches usually focus only on adaptation at the application level, excluding the system's operational environment.[7–9] Adaptation decisions typically depend on rule-based policies and heuristics without the ability to predict the effects of any adaptation actions on end-to-end system performance, which is essential to informing the adaptation process.

References

1. S. Kounev et al., "Model-Driven Algorithms and Architectures for Self-Aware Computing Systems," *Dagstuhl Reports*, vol. 5, no. 1, 2015; http://drops.dagstuhl.de/opus/volltexte/2015/5038.
2. A. Gandhi et al., "Autoscale: Dynamic, Robust Capacity Management for Multitier Data Centers," *ACM Trans. Computing Systems*, vol. 30, no. 4, 2012, pp. 14:1–14:26.
3. G. Jung et al., "Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures," *Proc. IEEE 30th Int'l Conf. Distributed Computing Systems* (ICDCS 10), 2010, pp. 62–73.
4. Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multitier Applications," *Proc. 4th IEEE Int'l Conf. Autonomic Computing* (ICAC 07), 2007; http://dx.doi.org/10.1109/ICAC.2007.1.
5. B.H.C. Cheng et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," *Software Eng. for Self-Adaptive Systems*, B.H.C. Cheng et al., eds., LNCS 5525, Springer, 2009; doi: 10.1007/978-3-642-02161-9_1.
6. M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Trans. Autonomous and Adaptive Systems*, vol. 4, no. 2, 2009, pp. 14:1–14:42.
7. P. Oreizy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, 1999, pp. 54–62.
8. D. Garlan, B. Schmerl, and S.-W. Cheng, "Software Architecture-Based Self-Adaptation," *Autonomic Computing and Networking*, Springer, 2009, pp. 31–55.
9. B. Morin et al., "Models@Run.time to Support Dynamic Adaptation," *Computer*, vol. 42, no. 10, 2009, pp. 44–51.

A software component is a unit of composition with explicitly defined interfaces. To describe the performance behavior of a service offered by a component, the application architecture metamodel supports multiple (possibly coexisting) behavior abstractions at different granularity levels—from black box to fine-grained behavioral descriptions. The novelty of the support for multiple abstraction levels is that the model is usable in different online performance prediction scenarios with different goals and constraints, from a quick analysis of performance bounds to a detailed system simulation.

Deployment and usage profile metamodels

The deployment metamodel captures the link between the resource landscape and the application architecture. It associates software components with their allocated containers in the resource landscape. The usage-profile metamodel captures workload type (open or closed) along with a probabilistic description of workload intensity (such as request-arrival rates), user behavior, and which services are called and in what order.

Adaptation points and process metamodels

The adaptation points metamodel describes the elements of the resource landscape and the application architecture that can be reconfigured at runtime. On the basis of this model, the adaptation process metamodel enables designers to describe the way the system adapts to environmental changes. This metamodel has three main parts: actions, tactics, and strategies. Figure 3 shows the main DML metamodel elements for each part.

Actions. Actions capture an adaptation operation's execution at the model level. Examples include increasing or decreasing a VM's processing resources, cloning or removing a VM, and migrating a software component.

Tactics. Tactics allow the description of more complex adaptations. A set of actions is composed to an `AdaptationPlan` through the use of control-flow elements, such as branches and loops. For example, a tactic to add resources might be: "if possible, increase a VM's processing resources; otherwise, start another VM." Once the tactic is applied in the model, online prediction techniques

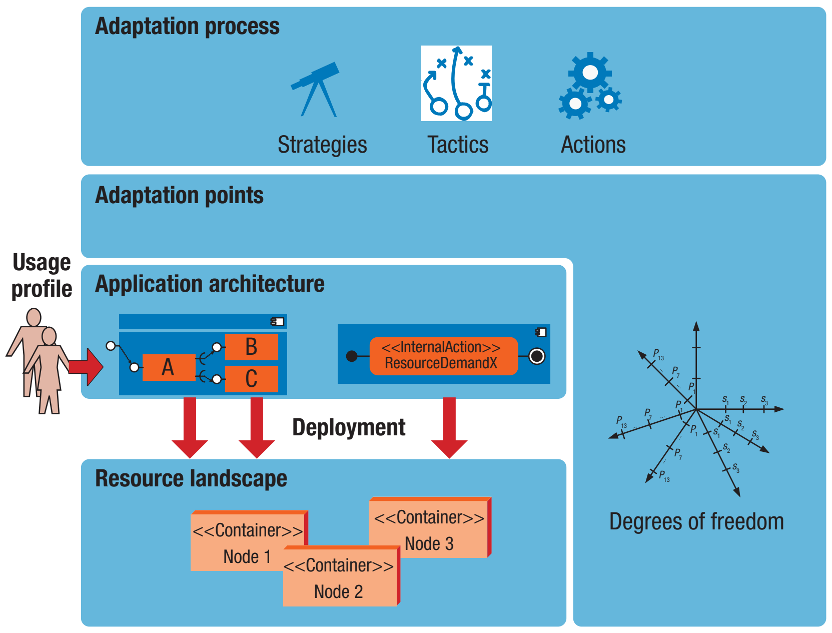


FIGURE 1. High-level structural overview of the Descartes Modeling Language (DML). DML's modular structure reflects the major aspects relevant for modeling IT system performance and resource management, such as available resources, application architecture, points at which the system can adapt at runtime, and how adaptation will occur.

evaluate its potential impact. If applying the tactic is likely to help achieve the stated adaptation goal, it is maintained as part of the adaptation plan. Otherwise, it is rolled back and another tactic is applied.

Strategies. Strategies capture the adaptation process's logical aspects by defining objectives and conveying ideas for satisfying them. A strategy can be a simple one-tactic plan or a complex, multilayered plan that uses multiple tactics to accomplish an objective. The tactic applied depends on the system state and the tactic's predicted impact on system performance. Because the tactical sequence is not predefined, the system can flexibly react with different tactics (defensive or aggressive) in unforeseen situations. A defensive strategy might be, "add as few resources as possible stepwise until response time violations are resolved." An aggressive strategy might be, "add a large amount of resources in one step so that response time violations are eliminated, ignoring resource inefficiencies."

SAMPLE DML METAMODEL

A sample DML model instance illustrates these metamodels. Figure 4, which depicts the resource landscape model instance, shows the resource hierarchy as well as resource-configuration templates and adaptation points. A full DML implementation of this instance is available from the DML website (http://descartes.tools/dml). The root element is `DataCenterA`, which represents the local datacenter in the computer science department at the Karlruhe Institute of Technology (KIT). `DataCenterA` contains `CompositeInfrastructure`, which relates to `ServerCluster1`, the DML label for a computing cluster, and a separate database server (`DatabaseServer`), which relates to a separate computing infrastructure (`CompositeInfrastructure`). The cluster consists of five computational nodes (`ComputeNodes`) connected by a 1-Gbit Ethernet LAN. Each node runs the XenServer 5.5 as a hypervisor, and two VMs run on top of each XenServer. The database server also connects to the cluster through four 1-Gbit Ethernet connections.

The templates to specify the resource configuration for various container types are stored in the `Container-TypeA_Specs` repository. The repository reduces modeling overhead by referring to templates instead of modeling each container's details. For example, `ComputeNodeTemplate` specifies the hardware-resource configuration of the computational nodes in the cluster. Each node has two `ActiveResource-Specification` specifications, one for each of its CPUs. Each CPU, in turn, has four cores with 2.66 GHz and uses the `PROCESSOR_SHARING` scheduling policy. `XenServer5.5Template` is a template for the runtime environment (`Runtime-Environment`) of class `HYPERVISOR`. Finally, `VMTemplate` specifies the configuration of the VMs hosted by the XenServer.

The model instance in Figure 4 also describes the system's adaptation points: the number of a VM's CPUs (`NrOfVcpus`), the number of VM instances (`VmInstances`), and the VM's location (`VmHost`). Corresponding to these variable elements, the model instance contains three adaptation points—one `ModelVariableConfigurationRange` and two `ModelEntityConfigurationRange`—the boundaries of which can be specified using the Object Constraint Language (OCL; www.omg.org/spec/OCL). Figure 5 shows the code for the two `ModelEntityConfigurationRange` points.

BUILDING MODELS IN DML

We have built a series of DML models to enable online performance prediction and model-based self-adaptation, which is based on a modified control loop. We also developed an adaptation framework that takes a DML instance as input and interprets the adaptation process, applying the modeled

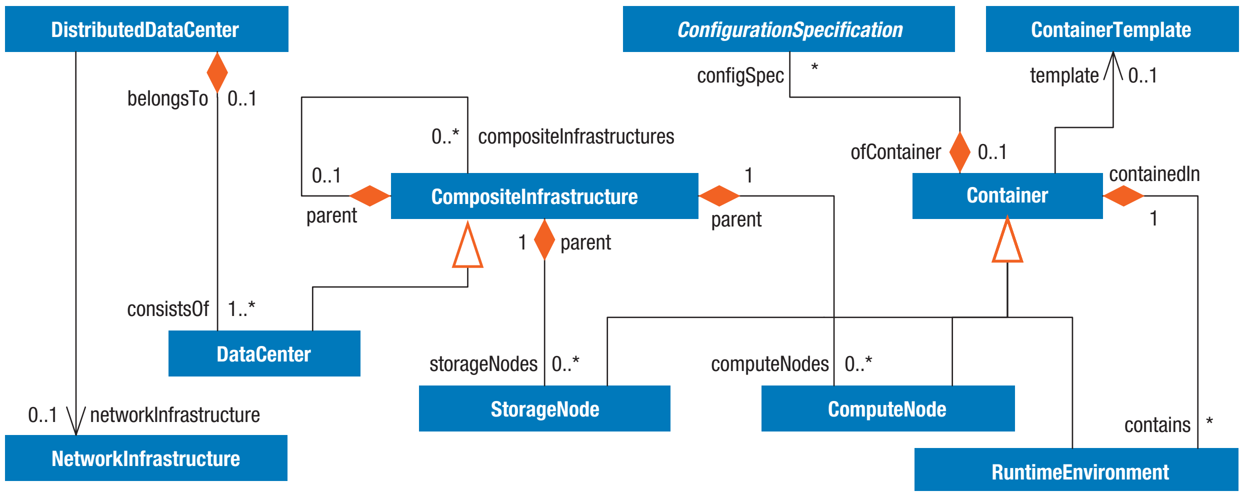


FIGURE 2. Core of resource landscape metamodel. The root entity is the distributed datacenter (DistributedDataCenter), which consists of one or more datacenters (DataCenter) and communicates using a network infrastructure (NetworkInfrastructure). Each DataCenter is an instance of a CompositeInfrastructure, which contains a set of nodes (Containers) of two types (ComputeNode or StorageNode). Each node contains a set of nested RuntimeEnvironment entities, which can be any infrastructure software that runs on a physical node, such as a virtualization platform, an OS, or middleware. Orange diamonds denote "consists of" or "contains," white arrowheads denote "is a," and open arrowheads denote "refers to" or "is associated with."

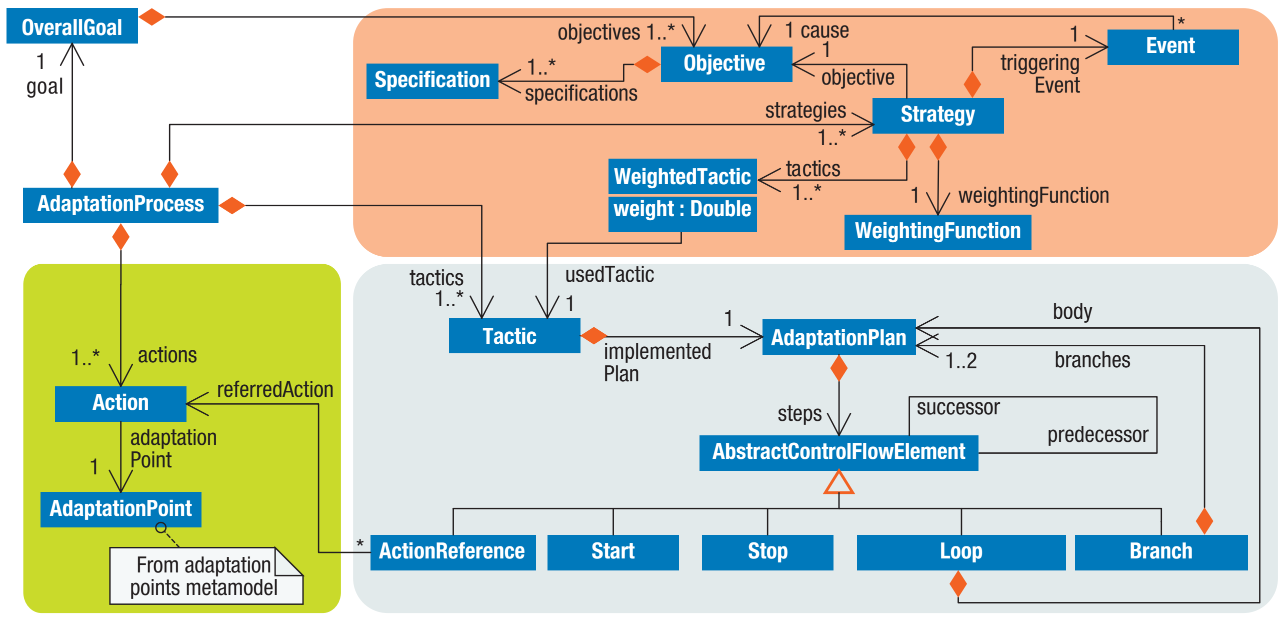


FIGURE 3. Core of the adaptation process metamodel. A strategy (orange section) captures the logical aspects of system adaptation, defining the objective to be accomplished and some idea of how to achieve it. A tactic (light blue section) composes a set of actions (green section), which capture the execution of an adaptation operation at the model level.

changes on the application architecture, resource landscape, and deployment models. Finally, we have constructed an open source tool chain (http://descartes.tools) to support the design of systems with self-aware performance and resource management capabilities. The tools include editors and solvers for DML models, a workload classification and forecasting tool, and a library for resource-demand estimation.

Online performance prediction

Online predictions include both the effects of workload changes on system performance and the expected impact of adaptation tactics. The impact of a workload change or an applied tactic

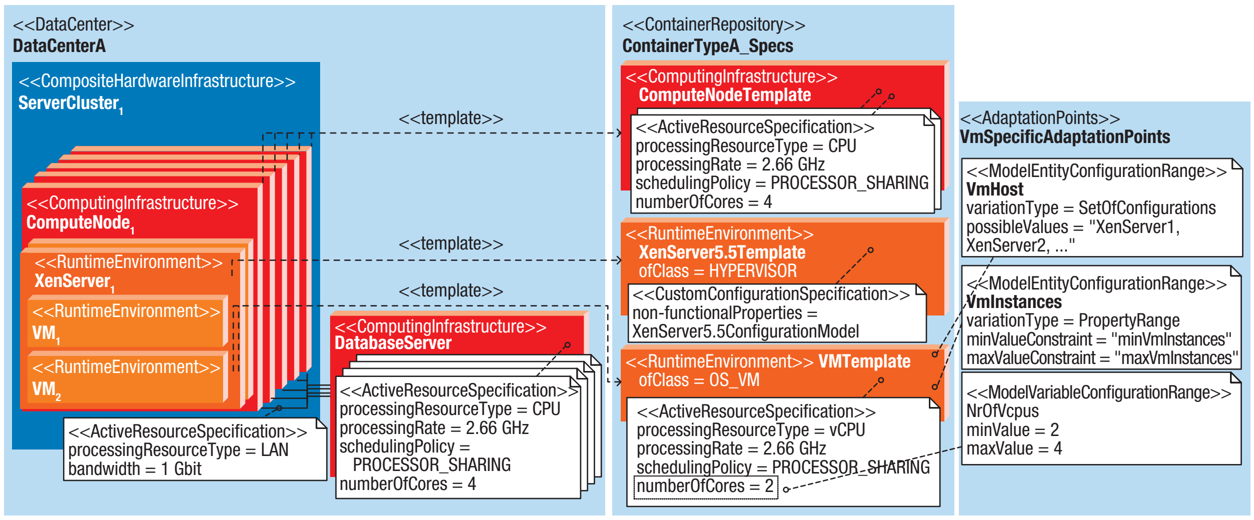

<<DataCenter>>
DataCenterA
<<CompositeHardwareInfrastructure>>
ServerCluster1
<<ComputingInfrastructure>>
ComputeNode1
<<RuntimeEnvironment>>
XenServer1
<<RuntimeEnvironment>>
VM1
<<RuntimeEnvironment>>
VM2
<<ActiveResourceSpecification>>
processingResourceType = LAN
bandwidth = 1 Gbit
<<ComputingInfrastructure>>
DatabaseServer
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<template>>
<<template>>
<<template>>
<<ContainerRepository>>
ContainerTypeA_Specs
<<ComputingInfrastructure>>
ComputeNodeTemplate
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<RuntimeEnvironment>>
XenServer5.5Template
ofClass = HYPERVISOR
<<CustomConfigurationSpecification>>
non-functionalProperties =
XenServer5.5ConfigurationModel
<<RuntimeEnvironment>> VMTemplate
ofClass = OS_VM
<<ActiveResourceSpecification>>
processingResourceType = vCPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 2
<<AdaptationPoints>>
VmSpecificAdaptationPoints
<<ModelEntityConfigurationRange>>
VmHost
variationType = SetOfConfigurations
possibleValues = "XenServer1, XenServer2, ..."
<<ModelEntityConfigurationRange>>
VmInstances
variationType = PropertyRange
minValueConstraint = "minVmInstances"
maxValueConstraint = "maxVmInstances"
<<ModelVariableConfigurationRange>>
NrOfVcpus
minValue = 2
maxValue = 4

FIGURE 4. Sample resource landscape instance annotated with adaptation points. The container repository (`<<ContainerRepository>>`) stores and refers to templates instead of modeling all the details of each container, which reduces modeling overhead. System adaptation points include the number of a VM's CPUs (`NrOfVcpus`), the number of VM instances (`VmInstances`), and the VM's location (`VmHost`).

```
context ModelEntityConfigurationRange
inv  minVmInstances:
       let similarContainers   :  Set(Container) = Container.allInstances()
              -> select(c | c.template = self.adaptableEntity)
       in similarContainers -> size() > 1;
context ModelEntityConfigurationRange
inv  maxVmInstances:
       let similarContainers   :  Set(Container) = Container.allInstances()
              -> select(c | c.template = self.adaptableEntity)
       in similarContainers -> size() < 4;
```

FIGURE 5. Boundaries for two `ModelEntityConfigurationRange` adaptation points, which are specified in the Object Constraint Language.

at the model level can be predicted by using stochastic modeling and analysis to evaluate the adapted DML model instance. One of DML's novel aspects is that it supports different abstraction levels of service behavior as well as different stochastic analysis techniques—which, combined, allow tradeoffs between prediction accuracy and time to result (time from triggering the prediction to obtaining its result). Our current DML version supports an approximate analytical technique based on mean-value analysis and two more detailed and accurate solving techniques based on discrete-event simulation.[4]

Model-based self-adaptation

Both software engineers and the autonomic computing community use the notion of a control, or feedback, loop as an essential generic concept to build adaptive and self-adaptive systems. The control loop generally specifies four phases—monitor, analyze, plan, and execute (MAPE)—and can add "with knowledge" (MAPE-K).[6]

We refined this generic control loop to fit the requirements of our model-based adaptation approach. The modified loop, shown in Figure 6, exploits DML's online performance prediction capabilities to implement adaptation processes at the model level.

Observe/reflect. The system collects monitoring data (observations of the system and its environment), which is used to extract, refine, calibrate, and continuously update the DML models, providing the basis for online workload forecasting and performance prediction.

Detect/predict. Monitoring data and online DML models are used to analyze the current system state so as to detect or predict performance problems, such as SLA violations or inefficient resource use. Proactive system adaptation requires anticipating performance problems. To this end, we developed an approach for self-adaptive workload classification and forecasting that uses techniques from time-series analysis.[7] When a change in the workload intensity is forecast, these techniques are applied to the online DML model using online prediction techniques to predict the impact on the system performance.

Plan/decide. The online DML models are used to find an adequate solution to a detected or predicted problem by adapting the system at runtime. Three steps are executed iteratively in this phase: selection of an adaptation tactic applied at the model level, prediction of the tactic's impact, and incremental construction of an adaptation plan. The three steps are driven by the adaptation process model.

Act/adapt. The actual adaptation is performed on the real system by executing the adaptation actions that have been successfully applied at the model level.

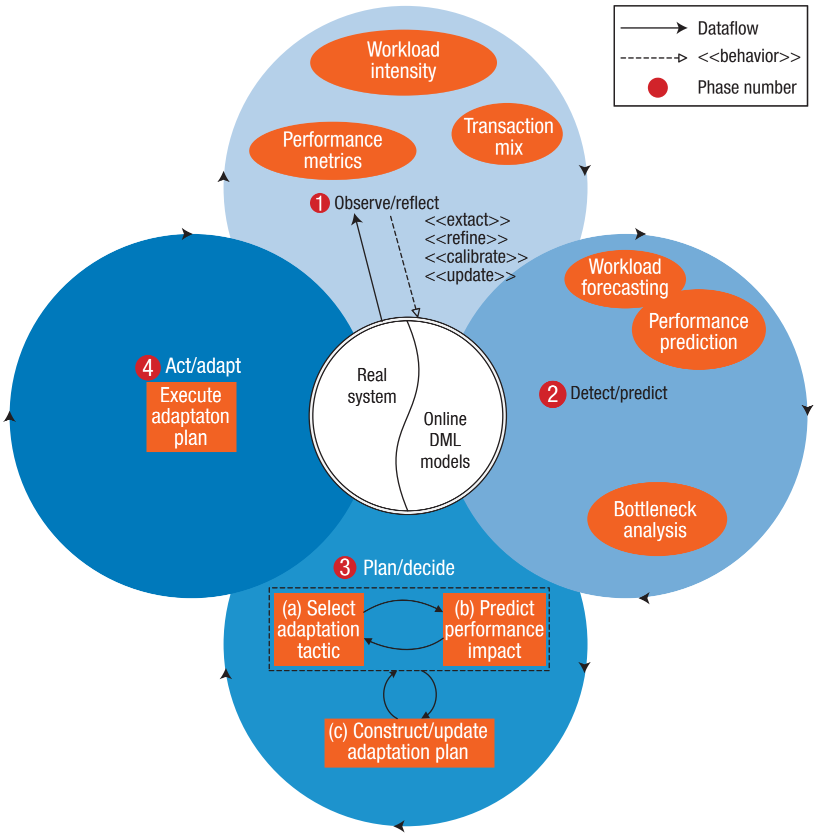


FIGURE 6. Self-aware system adaptation loop in a real system. The system executes the DML models that implement the adaptation plan, which includes system architecture, operational goals and policies, dynamic system state, and adaptation strategies and process.

EVALUATION RESULTS

One of the industrial case studies in which we applied our approach was to model a representative business application (as defined by the SPEC-jEnterprise2010 benchmark). Figure 4 shows the resource landscape and the adaptation points for the case study; the application architecture model instance can be seen at http://descartes.tools/dml/examples. The adaptation process is shown in Figure 7.

Efficiency gains

To demonstrate the efficiency gains of our model-based system adaptation—a proactive approach—we compared the total amount of allocated resources and the number of SLA violations when applying our approach against conventional static resource allocation—a reactive approach—using data from an industrial partner to ensure that the workload was realistic.

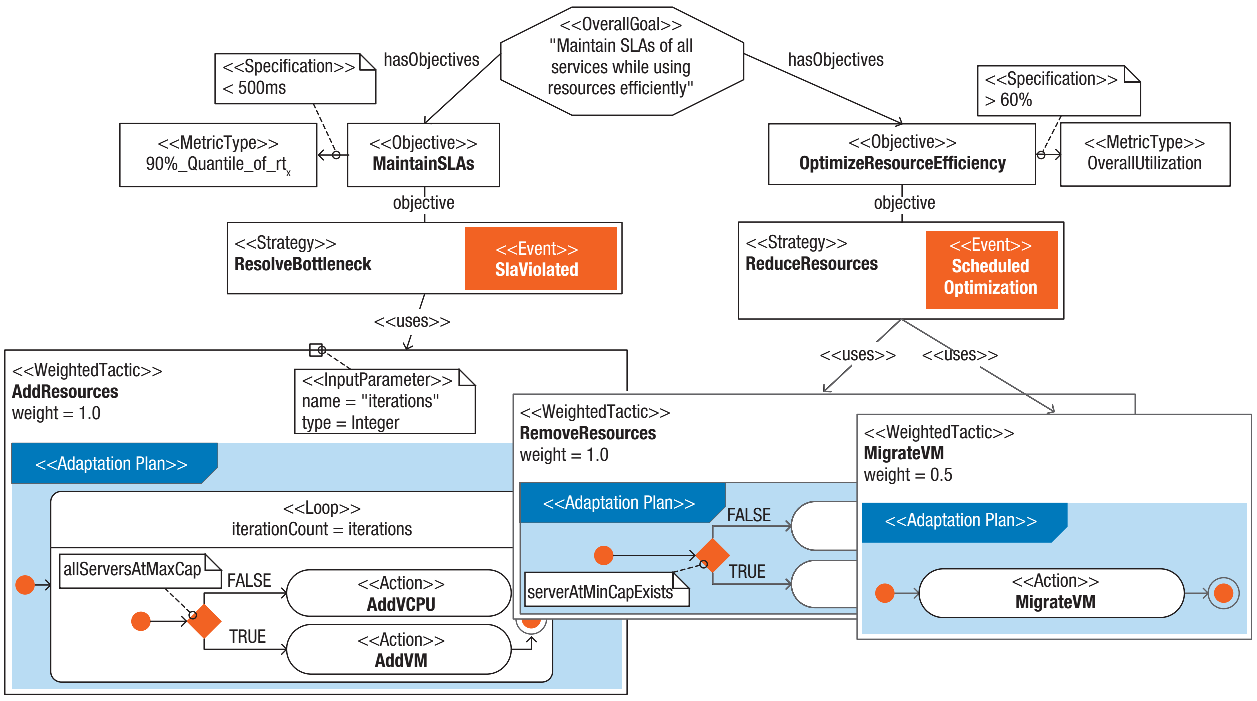
Figure 8 shows the workload as the number of processed transactions from Monday to Sunday in 15-minute frames (a total of 575 frames). We assumed that the maximum workload would be eight times the standard workload (8× workload intensity).

The static approach required 2,300 active nodes (575 time frames × 4 nodes). The reactive approach, which performs an adaptation action (to add or remove a node) when an SLA's response time is violated or when a resource is not used efficiently, performed 109 adaptation actions and used 1,002 active nodes. The number of active nodes decreased—44 percent of the resources used with the static assignment—but at the cost of 109 SLA violations.

Our proactive approach, which adjusts resource allocation to the predicted workload before violations occur, used 1,040 active nodes, but had only 43 SLA violations. Thus, although our approach needs approximately 5 percent more resources than the reactive approach, it can avoid approximately 60 percent of the SLA violations for that approach.

Overhead analysis

To better understand efficiency, we divided the overhead of our proactive adaptation approach into overhead for workload classification and forecasting and overhead for the adaptation process. Our experiments showed that workload classification and forecasting overhead ranged from seconds to a few minutes, depending on the data and configuration settings.[5] The adaptation process overhead is significantly higher because it depends on the number of iterations to find a solution at the model level and model performance–analysis overhead for each iteration. The number of iterations to find a solution is application specific and relies on the adaptation-process specification. In general, the more clearly adaptation goals are specified within the process, the fewer iterations are required to find a solution. The overhead for analyzing model performance depends on the techniques used for online performance predictions and on model complexity.



<<OverallGoal>> "Maintain SLAs of all services while using resources efficiently"
hasObjectives
hasObjectives
<<Specification>> < 500ms
<<MetricType>> 90%_Quantile_of_rt_x
<<Objective>> MaintainSLAs
objective
<<Strategy>> ResolveBottleneck
<<Event>> SlaViolated
<<uses>>
<<Specification>> > 60%
<<Objective>> OptimizeResourceEfficiency
<<MetricType>> OverallUtilization
objective
<<Strategy>> ReduceResources
<<Event>> Scheduled Optimization
<<uses>>
<<uses>>
<<WeightedTactic>> AddResources weight = 1.0
<<InputParameter>> name = "iterations" type = Integer
<<Adaptation Plan>>
<<Loop>> iterationCount = iterations
allServersAtMaxCap
FALSE
TRUE
<<Action>> AddVCPU
<<Action>> AddVM
<<WeightedTactic>> RemoveResources weight = 1.0
<<Adaptation Plan>>
FALSE
TRUE
serverAtMinCapExists
<<WeightedTactic>> MigrateVM weight = 0.5
<<Adaptation Plan>>
<<Action>> MigrateVM

FIGURE 7. A schematic representation of the adaptation process for the case study. The overall objective is to maintain service-level agreements (SLAs) while using resources efficiently, which branches to `MaintainSLAs` (left) and `OptimizeResourceEfficiency` (right), each of which has an attendant strategy to either increase resources (`ResolveBottleneck`) or decrease them (`ReduceResources`). Each strategy has corresponding tactics and actions.

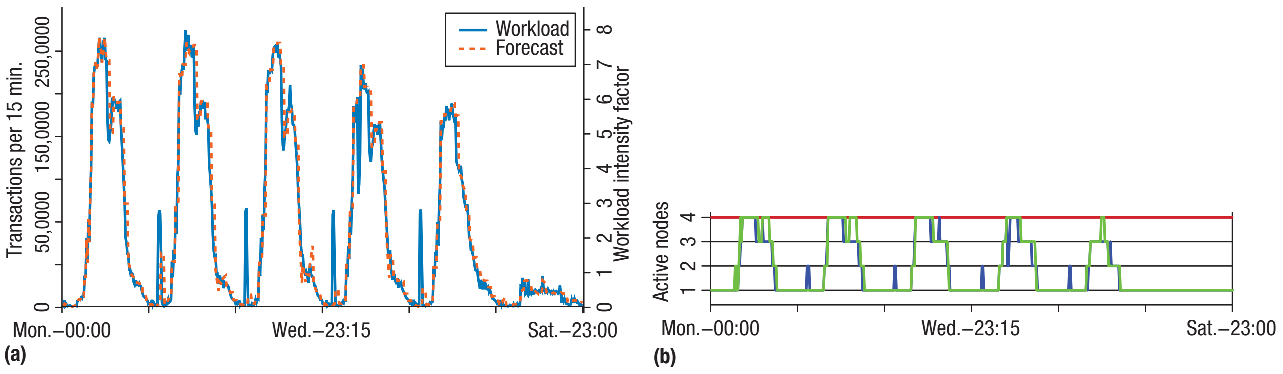


FIGURE 8. Workload intensity as related to node use in the case study. (a) Forecast versus actual workload intensity for four nodes over time. Transactions are in 15-minute time frames for a total of 575 possible time frames over six days. (b) Same timeline with three possible node-allocation approaches. The peaks and valleys represent node use. In the static approach (red horizontal line), all nodes must be active for all 575 time frames. The reactive (green lines) and proactive (blue lines) approaches add and remove nodes more efficiently to cope with workload intensity, but the proactive approach avoids many more SLA violations.

Experiments revealed that the time to obtain prediction results varied between seconds and a few minutes in the worst case.[4,5] As demonstrated in representative case studies, this is sufficient for different scenarios such as business information systems and computationally intensive applications.[5]

Our DML-based framework to design self-aware IT systems has advantages over trigger-based and black-box modeling in that it considers the individual effects and complex interactions between application workload profiles and resource contention at multiple levels and can describe dynamic aspects like adaptation processes at the model level. Descriptions are easy to understand, can be machine processed, and are reusable. Validation in several case studies shows that significant resource-efficiency gains are possible without sacrificing SLA performance requirements.

REFERENCES

1. S. Kounev et al., "Model-Driven Algorithms and Architectures for Self-Aware Computing Systems," *Dagstuhl Reports*, vol. 5, no. 1, 2015; http://drops.dagstuhl.de/opus/volltexte/2015/5038.
2. G. Blair, N. Bencomo, and R.B. France, "Models@Run.time," *Computer*, vol. 42, no. 10, 2009, pp. 22–27.
3. S. Kounev, F. Brosig, and N. Huber, *The Descartes Modeling Language*, tech. report, Univ. of Würzburg, 2014; https://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/10488.
4. F. Brosig, "Architecture-Level Software Performance Models for Online Performance Prediction," PhD dissertation, Computer Science Dept., Karlsruhe Inst. of Technology, 2014; http://nbn-resolving.org/urn:nbn:de:swb:90-435372.
5. N. Huber, "Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures," PhD dissertation, Computer Science Dept., Karlsruhe Inst. of Technology, 2014; http://nbn-resolving.org/urn:nbn:de:swb:90-432462.
6. J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–50.
7. N.R. Herbst et al., "Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning," *Concurrency and Computation—Practice and Experience*, vol. 26, no. 12, 2014, pp. 2053–2078.

ABOUT THE AUTHORS

SAMUEL KOUNEV is a professor and head of the Chair of Software Engineering at the University of Würzburg. His research interests include software systems' performance and dependability, autonomic computing, and systems benchmarking. Kounev received a PhD in computer science from the Technical University of Darmstadt. He is a member of ACM, IEEE, and the German Computer Science Society. Contact him at skounev@acm.org.

NIKOLAUS HUBER is a software architect at CarGarantie Versicherungs AG and an associate researcher in the Chair of Software Engineering at the University of Würzburg. He received a PhD in computer science from the Karlsruhe Institute of Technology (KIT). Contact him at nikolaus.huber@uni-wuerzburg.de.

FABIAN BROSIG is a lead developer at Minodes GmbH and an associate researcher in the Chair of Software Engineering at the University of Würzburg. His research interests include software performance engineering with an emphasis on runtime performance and resource management. Brosig received a PhD in computer science from KIT. Contact him at fabian.brosig@uni-wuerzburg.com.

XIAOYUN ZHU is a senior architect at Futurewei Technologies. Her research interests include the application of control theory, optimization, and statistical learning to the automation of IT systems and services management. Zhu received a PhD in electrical engineering from Caltech. She is a member of ACM, IEEE, and USENIX. Contact her at xiaoyun.zhu@huawei.com.