# A Run-Time Engineering Knowledge Base for Reconfigurable Systems

*Thomas Moser, Alexander Schatten, Wikan Danar Sunindyo, and Stefan Biffl*
Institute for Software Technology and Interactive Systems, Vienna University of Technology, Austria

**Abstract**
Engineers of reconfigurable software-intensive systems, who want to adapt a system at runtime to make it more robust against relevant failures, need engineering knowledge from both design-time and run-time software models. Design models usually do not exist in a machine-understandable format suitable for automated reflection on reconfiguration at runtime. Thus domain and software experts are needed to integrate the fragmented views from these models.
In this paper we propose an ontology-based engineering knowledge base to provide relevant design-time and run-time engineering knowledge in machine-understandable form to make better informed decisions on failure-related system adaptation. We illustrate and evaluate the approach with decisions and data from a real-world case study in the area of software-intensive production automation systems. While the ontology approach was found useful on the conceptual level, the use for decision support at run time seems in practice limited by the performance of the ontology implementation.

Keywords: Software-intensive systems engineering, run-time decision making, failure and recovery, ontology.

Software-intensive systems in production automation need to be flexible to adapt to changing business situations and to become more robust against relevant classes of failures. Production automation systems consist of components, for which a general design and behavior is defined during the design phase, but much of the specific design and behavior is defined during implementation, deployment, and run time with a range of configuration options. The "Simulator for Assembly Workshops" (SAW) [9] simulates complex reconfigurable production automation systems to maximize the overall system output by scheduling sequences of transport and machine tasks over 100 times faster than the actual hardware at the ACIN lab[1]. Figure 1 (left hand side) illustrates an example assembly workshop layout that consists of software-controlled manufacturing components: transport components such as conveyor belts (dark green), crossings (light green), and stoppers (small yellow/red circles); and assembly machines (colored rectangles with round corners); product parts are transported on pallets (colored rectangles; colors represent the target machines). The SAW has been validated with real hardware components in an assembly workshop lab to make sure the outcome of simulation runs is relevant for real-world production automation systems.

Engineers, who want to adapt the system at runtime, need information from software models that reflect dependencies between components at design and run time, e.g., the workshop layout, customer orders and assembly procedures that translate into needs for machine function capacities over time; and the coordination of tasks for redundant machines in case of a failure. During development design-time software models like data-oriented models (e.g., class or EER diagrams) or workflow-oriented models (e.g., sequence diagrams or state charts) are the basis to derive run-time models but are often not provided in machine-understandable format to reflect on changes at runtime, i.e., the knowledge is kept in an explicit human-understandable way but cannot be accessed by components automatically. Domain and software experts are needed to integrate the fragmented views (e.g., propagating model changes into other models, cross-model consistency checks) from these models, which often is an expensive and error-prone task due to undetected model inconsistencies or lost experience from personnel turnover.

Practitioners, especially designers and quality assurance (QA) personnel, want to make reconfigurable software-intensive systems (which like SAW consist of components defined by general design-time behavior, derived run-time configuration, and run-time specific behavior enactment) more robust against important classes of failures: machine failures, misuse from invalid supply, and failure-related changes in machine capacities at runtime. QA people could benefit from more effective and efficient tool support to check system correctness, by improving the visibility of the system defect symptoms (e.g., exceptions raised from assertions).

Challenges to detect and locate defects at run-time come from the different focus points of models: e.g., components and their behavior are defined at design time, while configurations may change at run time and violate tacit engineering assumptions in the design-time models. Without an integrated view on relevant parts of both design-time and run-time models inconsistencies from changes and their impact are harder to evaluate and resolve between design and run time.

Better integrated engineering knowledge can improve the quality of decisions for run-time changes to the system, e.g., better handling severe failures with predictable recovery procedures, lower level of avoidable downtime, and better visibility of risks before damage occurs.

[1] Automation & Control Institute; http://www.acin.tuwien.ac.at

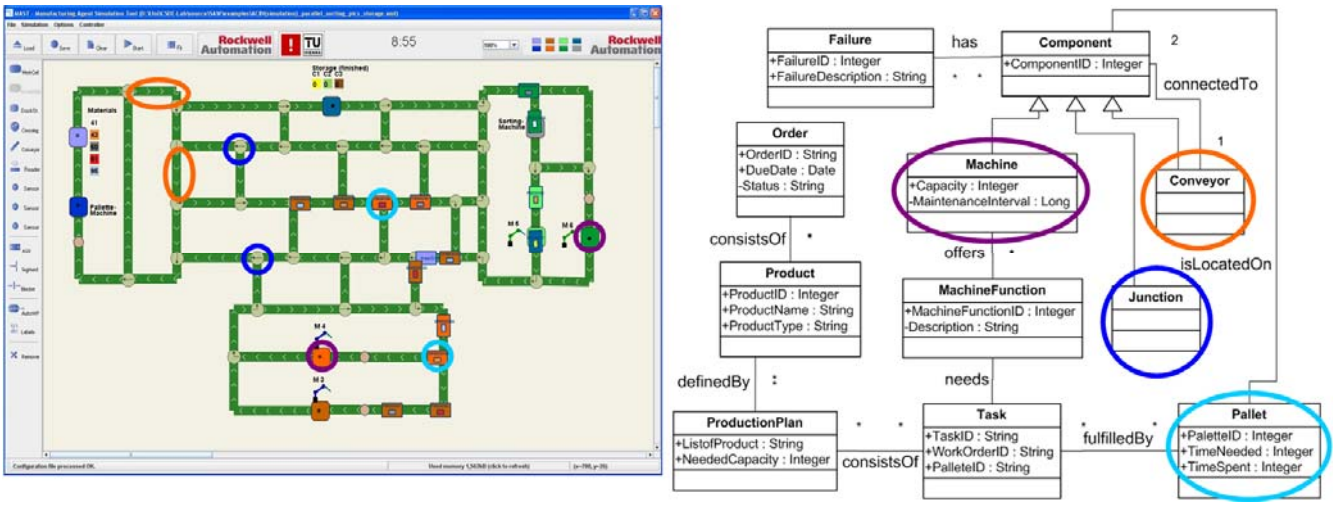


**Figure 1: SAW Simulator and underlying data model.**

In this paper we present an approach to improve support for run-time decision making with an ontology: a domain-specific *engineering knowledge base* (EKB) that provides a better integrated view on relevant engineering knowledge in typical design-time and run-time models, which were originally not designed for machine-understandable integration. The EKB can contain schemas on all levels and instances, data, and allows reasoning to evaluate rules that involve information from several models that would be fragmented without machine-understandable integration.

The EKB provides the following benefits:

**A. Uniform and efficient access to related data** in design-time, deployment, and run-time models on the levels of schemata, instances, and configurations. This feature allows reasoning to effectively evaluate specific decision alternatives, e.g., the importance of defect messages. Further the feature supports checking the impact of changes in one model on the consistency of other models.

**B. Derivation of assertions** that should hold at run-time to establish and maintain causal links between design/runtime models and the running software. These assertions can support systematic exception handling and escalation procedures; e.g., check correct sequences of messages; valid behavior of component groups; or check actual run-time performance with design estimates/limits.

We illustrate and evaluate the ontology-based approach with two types of run-time decisions (RTDs) from a real-world case study in the area of software-intensive production automation systems.

**RTD-1. Error message filtering and sequencing.** The decision on how to present a (potentially very large) set of error messages from distributed components. These components and their errors may be interrelated in several ways (logical, process, physical, etc.); accurately understanding these relationships may greatly help to inform the operator which errors are really important and should be addressed first with appropriate recovery procedures. Models that provide fragmented, inconsistent, or outdated views on the actual system are of limited help beyond the expertise of the operator.

**RTD-2. Machine work load adaptation.** The decision coordinates a set of autonomous components that achieve a common goal, in our case adapt work load and work intensity parameters for a set of redundant machines in a workshop to allow coordinated machine maintenance without unnecessarily interrupting a production line. Traditional design-time solutions often prescribe solutions that depend on design estimates and hence are less able to exploit opportunities of concrete configurations that may change at run time.

In the remainder of the paper we survey relevant engineering models for their contributions and limitations to support run-time decision making; we describe a real-world case on system adaptation to accommodate run-time failures for collecting evidence to which extent richer and better integrated semantic knowledge can translate into better decision making; further important practical issues are to investigate the effort to import the data from the relevant models into the knowledge base and the processing speed of data access and reasoning at run time.

## Evolution of Engineering Models towards Run-Time System Analysis and Adaptation

Engineering models have evolved from means to structure complex domains and designs at design time towards model-driven approaches that bring domain information closer to implementation and runtime. However, systems that are designed for adaptation at runtime need more advanced approaches to provide relevant and accurate engineering knowledge to guide runtime system analysis and adaptation.

**Structuring design complexity.** Models are used on various levels in software engineering: Data models like entity relationship (ER) diagrams originate in the late 1970s [5]. With the unified modeling language (UML) [3] a standardized set of diagrams and modeling techniques were introduced for object-oriented design. However, these models are mostly used initially during the

design time of a project and get seldom adapted to changes during implementation or operation.

**Connecting design and implementation.** There are tool providers[2,3] who claim to support roundtrip engineering from design models to source code and back. However, a stronger and more consistent integration between the design models and the implementation phase artifacts comes from the model-driven architecture (MDA), model-driven development (MDD) [11], and model-driven configuration management: MDD concepts allow a clear connection between various types of design models and generated code, usually a series of models starting with computation-independent models (CIMs), which describe the domain without hard/software in mind, and platform-independent models (PIMs) that are transformed to target platform-specific models (PSMs). The final transformation leads from the PSM to the platform-specific implementation (PSI).

There are several interesting aspects about MDD: The models develop from high-level abstractions to concrete code over several intermediate steps. The initial model can be a general UML model or a domain-specific language model. In MDD (opposed to earlier modeling approaches) the model is used also in the implementation phase, i.e., used to create platform-specific code, but is not used at run time.

**Connections between design, implementation, and runtime.** Traditional software engineering approaches mostly focus on the development phase and see configuration management (CM) as a support task. However, CM is an example model that is valuable at design time, implementation, deployment, and runtime. Some approaches thus suggest including CM and application lifecycle management[4] (ALM) into the MDD concept [6].

**Run-time needs of distributed reconfigurable software-intensive systems.** Ahluwalia et al. [1] observe a shift from "monolithic to highly networked, heterogeneous, interactive systems" that has led to a "dramatic increase in both development and system complexity", where at the same time the "demands for safety, reliability, and other qualitative attributes have increased across application domains." Oreizy et al. [10] additionally mention the necessity of "run-time evolution" of modern multi-user, distributed systems. Today many deployed applications gradually evolve over time (ideally without downtime for users) rather than undergo "big bang" version updates. Examples for such applications are e-Commerce Services like Online-Banking applications as well as most “Web 2.0” applications. Also many traditional IT-services in companies shift from regular service down-times to 365/24/7 operations. This is particularly the evident for internationally operating companies.

The problem gets particularly critical in domains like distributed real-time and embedded systems as in automotive and production automation industry applications. Krishna et al. [8] analyze Quality of Service (QoS) for middleware platforms and associated challenges for CM and customization. In many cases these systems have 100s of configuration parameters maintained for several customized versions. Currently, these challenges are mostly tackled manually by domain and systems experts, which seems expensive and error prone. The authors argue how model-driven techniques could support the packaging, deployment, configuration, and also QoS assurance and adaptation in the run-time system.

**Feedback of run-time experience to design.** An underlying trend is to bring development activities closer to the run-time environments, i.e., using data from the deployed system for engineering purposes (e.g., QoS parameters). Additionally, we observe more intensive research activities [12] to design and apply MDD where the models do not stop at development but also support the run-time environment of the system. Recent research investigates mechanisms towards automatic run-time failure detection [12] and ultimately self-healing systems.

Garlan et al. [6] describe model-based approaches for self-healing autonomous systems with a similar idea: remove the traditional separation between system creation/modification and run-time environment with an integrated approach. "Traditional mechanisms that allow a system to detect and recover from errors are typically wired into applications at the level of code where they are hard to change, reuse, or analyze. An alternative approach is to use externalized adaptation: one or more models of a system are maintained at run time and external to the application as a basis for identifying problems and resolving them." The authors particularly point out that system-internal exception handling and configuration (hence not always easy to change) is problematic in many modern systems as they are designed to run continuously, and also updates and reconfiguration should be done without system shutdown.

Long-running systems can also show behavior that is not necessarily an error in the strict sense of the definition: like degradation of the system performance. Such issues are difficult to tackle with conventional exception handling mechanisms.

Unfortunately, models that are useful at design time are not necessarily (directly) useful at run time, if different information is needed. Wuttke [12] suggests taxonomies or ontologies to model the necessary information. However, we believe that the connection between models at design time and run time should not be lost, i.e., models at run time should be (at best) not a new category of models, but an enrichment of models that are already in use at design time.

**Ontologies for software engineering.** An ontology is a representation vocabulary for a specific domain or subject matter, e.g., production automation. More precisely, it is not the vocabulary as such that qualifies as an ontology, but the (domain-specific) concepts that the terms in the vocabulary are intended to capture [4].

The infrastructure of MDA provides architecture for creating models and meta-models, defining transformations between these models, and managing meta-data.

[2] Visual Paradigm: http://www.visual-paradigm.com

[3] IBM Rational Software Development Tools: http://www.ibm.com

[4] John Carrillo and Scott McKorkle, Application Lifecycle Management Meets Model-Driven Development, Dr.Dobb's, September 2008, http://www.ddj.com/architect/210300020

Although the semantics of a model is structurally defined by its meta-model, the mechanisms to describe the semantics of the domain are rather limited compared to machine-understandable representations using, e.g., knowledge representation languages like RDF[5] or OWL[6]. In addition, MDA-based languages do not have a knowledge-based foundation to enable reasoning (e.g., for supporting QA), which ontologies provide [2].

Beyond traditional data models like UML class diagrams or entity relationship diagrams, ontologies provide methods for integrating fragmented data models into a common model without losing the notation and style of the individual models [7].

## An Integrating Engineering Knowledge Base

In this section, we introduce the Engineering Knowledge Base (EKB), a set of relevant information elements about components in machine-understandable format using ontology syntax. Components can query the EKB at run time to retrieve information for decision making, e.g., enriching and filtering failure information or run-time coordination of machine workloads due to changes in the available machine capacity.

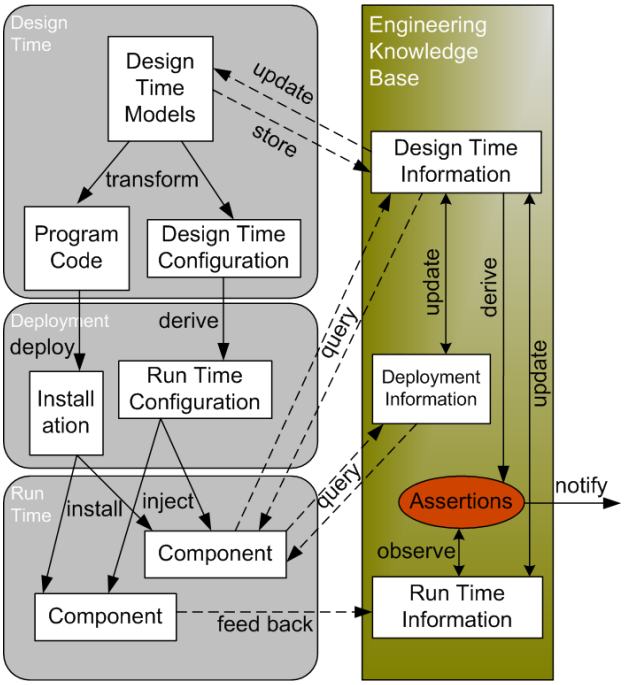


**Figure 2: An Engineering Knowledge Base in context.**

Figure 2 illustrates 3 major phases in the life cycle of software-intensive systems in the production automation domain: 1. *Design time*: Models that describe the workshop layouts, the building plans of manufactured products, etc. are transformed into executable program code and design-time configuration instructions. 2. In the *Deployment* phase, the executable program code is deployed into installable packages and the run-time configuration is derived from the design-time configuration. 3. At *Run Time* the deployed program code for system operation gets installed to a set of components and the run-time configuration gets injected into these components. This architecture has proven effective for systems whose properties change seldom, since the effort needed for transformation, deployment, and injection is considerable.

However, typical software-intensive systems also undergo reconfiguration phases, e.g., if some components fail or become unavailable. To support reconfiguration, the components need to be able to perform decisions at run time, since a complete new iteration of model transformation, program code deployment, and configuration injection would take too long. A major challenge of run-time decisions is to provide access to relevant design-time information that is usually stripped away during transformation for efficiency reasons.

The Engineering Knowledge Base (EKB) provides a place for storing design-time information that seems valuable for supporting run-time decisions of components, especially in the case of handling failures or unplanned situations (but not transformed into run-time code or configuration to limit their complexity).

Components can query the EKB at run time with query languages like SPARQL[7] (SPARQL Protocol and RDF Query Language) or SWRL[8] (Semantic Web Rule Language), which provide to the components the full expressive power of ontologies, including the ability to derive new facts by reasoning. In addition, components can feed back interesting observations into the run-time information collection of the EKB and therefore help to improve the design-time models (e.g., by improving estimated process properties with analysis of actual run-time data) and/or check the information based on a certain set of assertions. Furthermore, valuable deployment information can also be stored in the EKB in order to support and enhance for further deployments.

Based on the design-time information, it is possible to define a set of run-time assertions in the EKB. These run-time assertions observe the run-time information fed back into the EKB and can notify a specific role or system if the violation of an assertion has been detected.

## Real-World Use Case and Results

In this section, we describe a real-world use case on system adaptation to accommodate runtime failures. The use case is based on a Java simulation of an adaptive system that has been validated with a hardware version available at VUT[9]. In the simulation context we collect evidence to which extent a richer and better integrated semantic knowledge base can translate into more accurate faster and cheaper making.

[5] Resource Description Framework: http://www.w3.org/RDF/
[6] Web Ontology Language: http://www.w3.org/2007/OWL
[7] www.w3.org/TR/rdf-sparql-query/
[8] www.w3.org/Submission/SWRL/
[9] http://www.acin.tuwien.ac.at

Figure 1 (right hand side) illustrates parts of the data model, which are interesting for run-time decisions in UML-class-diagram style notation. Both machines and conveyor belts offer functions (either production or transport functions). These functions need a specific amount of time, represented through the min, max and expected value. Each machine provides a specific set of machine functions, has a kind of virtual input buffer, which contains all pallets that are scheduled to the particular machine; and stores the number of operations until the machine needs the next maintenance break. In contrast, a conveyor stores only the information about pallets that are at the moment using the particular conveyor. Both machines and conveyor belts can experience exceptional situations, so-called machine or conveyor failures. A failure is defined by a failure code, a description of the failure, and an estimate of the time needed to recover from the particular kind of failure.

Based on this data model, we derived two use case scenarios which show the run-time decision (RTD) support and derived assertions based on the capabilities of the Engineering Knowledge Base.

**RTD-1: Error message filtering and sequencing.** Discussions with industry partners showed that there exists need for a more effective detection and handling of failures at runtime. Currently, an operator conducts front-line failure handling and may receive a (potentially very large) set of error messages from distributed components. These components and their errors may be interrelated in several ways (logical, process, physical, etc.). As example for a wide range of failures of components that are interdependent (not always in obvious ways): an initial failure of a non-redundant conveyor may cause follow-up failures at (distant) machines that depend on transport from the failing conveyor. Typically, the operator will be notified about a set of failures and may not know which failure is of first importance; in most cases the operator tends to handle failures of expensive machines first before attending to less expensive components, such as the conveyor belt, which may lead to an ineffective sequence of recovery activities.

Using the EKB approach, it is possible a) to model the component interdependencies and their relationships to errors/failures and b) derive the original source of failure using ontology-based reasoning as well as present additional important failure information like instructions and responsibilities to the operator. Important run-time decisions in this scenario are which messages should be provided to the operator and which messages should be filtered out (e.g., less important defects) as well as the sequence in which the messages should be presented to the operator (e.g., messages from most important components first – most important does not necessarily mean largest or most expensive component).

An assertion for this run-time decision can be to flag not only direct failures of components but also non-reachable components or parts of the overall workshop. If relevant parts of the workshop are not reachable for an extended period, both the operator and some higher-level roles like a dispatcher or a ERP system need to be notified about the (temporary) change of the workshop system capacity so they can react to the foreseeable capacity change in time.

Figure 3 (bottom left) shows how a failure in Conveyor X (1) will result in unreachable Machines A and B (2) and consequent follow-up failure messages. However, the operator will receive in general several failure warnings, and has to reason on their overall meaning. If he chooses to handle the failure of Machines A or B first, valuable time will be lost. The EKB can deduce that Conveyor X is a predecessor of both Machines A and B using the *connectedTo* relation between Conveyor and Component (see the data model in Figure 1).

Thus it is possible to define a query whether the failure of a component should be reported to an operator or should be filtered out; see Listing 1 for a simple query definition which implies a failure of a component C should be filtered out if a failure of any predecessing component P occurs at the same time and if the Failure ID is not severe, e.g., of failure class “3”; in Listing 1 *reportFailure*(C) will return false, since the failure code is “5” and P is a predecessor of C. This query can be used as soon as a number of failures occur in order to identify a possible source of failure and to suppress confusing follow-up failures.

```
Failure(C,5,”noPalletInput”).
Failure(P,4,”motorBroken”).
predecessor(X,Y) :- connectedTo(X,Y).
predecessor(X,Y) :- connectedTo(X,Z),
                    predecessor(Z,Y).

reportFailure(C) :- not predecessor (C,P).
reportFailure(C) :- Failure(C,3).
```

**Listing 1: Rules for Reporting Failures.**

**RTD-2: Individual Preparation of Machine Maintenance Tasks.** A second relevant run-time decision in the production automation scenario is when to perform machine maintenance tasks in order to keep a certain minimum level of production output.

This decision could also be taken during design time, resulting in a decreased ability to react to new or changing environment conditions (e.g., failures, reconfiguration). Since system flexibility supports operational efficiency in production automation, the decision when to perform machine maintenance tasks (e.g., cleaning, refurbishment) and the preparations for these tasks (i.e., emptying the machine buffers) could be taken by the machines themselves taking into account the state of other machines and workshop environment conditions. The idea is to coordinate the maintenance tasks of a set of related machines to minimize the impact on the overall production process. This planned maintenance should also be reported to a controlling system (e.g., an ERP system) in order to allow in-time reaction to the future capacity changes.

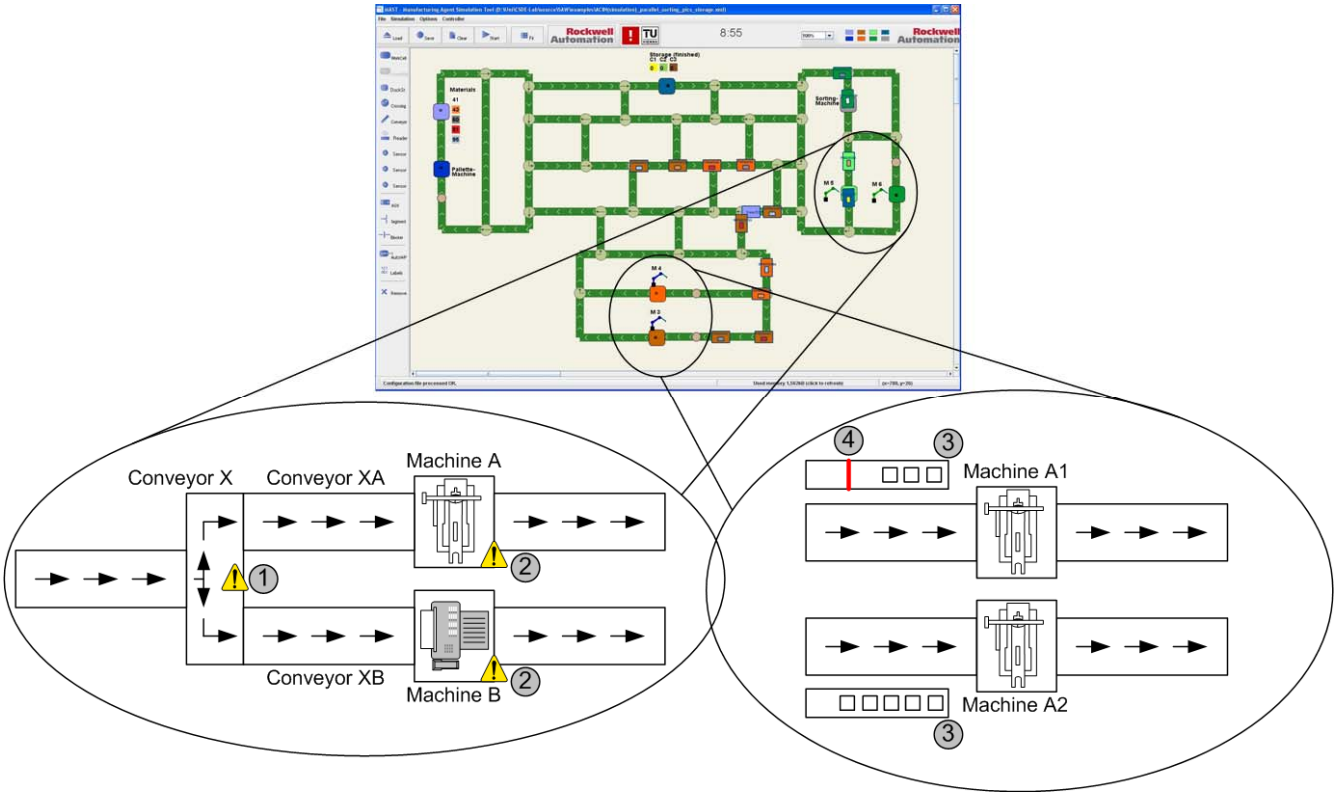


**Figure 3: Scenarios – RTD-1: "Error message filtering" and RTD-2: "Preparation of machine maintenance".**

An assertion for this run-time decision can be to detect a situation in which too many machines plan to go into maintenance mode at the same time and react by preventing some machines from entering maintenance mode and notifying an operator to investigate the actual situation.

Figure 3 (bottom right) illustrates two redundant machines (machines that provide at least in part similar functionality): Machines A1 and A2, have independent virtual input buffers (3) that contain all work orders which are scheduled to be processed at the particular machine. In addition, every machine has a counter counting the number of performed operations, as well as a threshold defining the approximate number of operations after which machine maintenance tasks should be performed (see the data model in Figure 1). Based on the average time a machine function takes, a machine can derive the optimal moment for starting maintenance tasks. As preparation of these tasks, the virtual machine buffer needs to be closed (4) and all remaining work orders need to be processed. To avoid that all redundant machines switch to maintenance mode at the same time, the planned maintenance time needs to be stored in the EKB, so other machines can adjust their maintenance plans.

Listing 2 shows a simple query to check whether a capacity change caused by a machine going into maintenance mode can be handled by the system itself or should be reported to a dispatcher and/or ERP system. There are three tasks which need the same machine function F1 and two machines A1 and A2, which offer machine function F1. A1 has a capacity of 1 task but needs to go into maintenance mode now since its number of allowed operations before a planned maintenance is 0. Machine A2 has a capacity of 2 tasks and 15 more allowed operations before its planned maintenance phase. The query now checks whether the number of tasks needing a certain machine function (F1 in our example) is greater than the available capacity of the other machines which offer this machine function (in our example machine A2) and returns true if the number of tasks is greater than the available capacity. In our example the query would return true, since there are three tasks needing the machine function F1, but the available capacity of machine A2 is only 2 tasks.

```
Task(T1,F1).
Task(T2,F1).
Task(T3,F1).
Machine(A1,F1,1,0).
Machine(A2,F1,2,15).

ReportCapacityChange(M1) :-
   Machine(M1,X,_,_),
   count(Task(_,X)) > Machine(M2,X,Y,_).
```

**Listing 2: Reporting capacity changes to ERP system.**

A current limitation of the EKB is the performance of the underlying ontology infrastructure, which allows making a few 100s to 1000s rule evaluations per second, but can be a bottleneck if too many new fact updates and queries come in from the distributed system, which consists of dozens of components that run task in 10- to 100-millisecond cycles.

## Summary and Further Work

In this paper we described an ontology-based approach to provide relevant design-time and run-time engineering knowledge stored in a so called Engineering Knowledge Base (EKB). The EKB provides a better integrated view on relevant engineering knowledge contained in typical design-time and run-time models in machine-understandable form to support runtime decisions. This approach is useful in the automation domain, and can more generally be used for other (distributed) engineering systems. We illustrated our approach with two types of run-time decisions from a real-world case study in the area of software-intensive production automation systems.

Major results of the evaluation of the proposed EKB approach were: Due to separation of automation code, diagnosis and decision support the complexity of single components can be reduced by approximately 20-30%, since these components now can rely on external information. Another benefit is the possibility to define assertions in the EKB which are checked based on the runtime information input of the running components. This can be seen as external Quality Assurance (QA) without interfering with the original production system and therefore it has proven to be easier to enrich existing applications without the need to make changes to legacy systems (smoother migration path). Further, the quality of information presented to an operator is improved since all information both from design-time as well as from runtime is available, leading to more intelligent run-time analysis and decision support.

Further important practical issues are to investigate the effort needed to import the data from the relevant models into the Engineering Knowledge Base and improving the performance of data access and reasoning at run time. The use of assertions for checking QoS parameters like system throughput is open to further research too.

## References


1. J. Ahluwalia, I.H. Kr\üger, W. Phillips, and M. Meisinger, “Model-based run-time monitoring of end-to-end dead-lines,” 5th ACM international conference on Embedded software (EMSOFT '05), ACM, 2005, pp. 100--109.
2. K. Baclawski, M.K. Kokar, P.A. Kogut, L. Hart, J. Smith, J. Letkowski, and P. Emery, “Extending the Unified Modeling Language for Ontology Development,” International Journal of Software and Systems Modeling (SoSyM), vol. 1, no. 2, 2002, pp. 142-156.
3. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.
4. B. Chandrasekaran, J.R. Josephson, and V.R. Benjamins, “What are ontologies, and why do we need them?,” Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems], vol. 14, no. 1, 1999, pp. 20-26.
5. P.P.S. Chen, “The entity-relationship model—toward a unified view of data,” ACM Transactions on Database Systems (TODS), vol. 1, no. 1, 1976, pp. 9-36.
6. F. Garlan, J.E. Lopez de Vergara, D. Fernandez, and R. Munoz, “A model-driven configuration management methodology for testbed infrastructures,” IEEE Network Operations and Management Symposium NOMS 2008, IEEE, 2008, pp. 747-750.
7. M. Hepp, P. De Leenheer, A. De Moor, and Y. Sure, Ontology Management: Semantic Web, Semantic Web Services, and Business Applications, Springer-Verlag, 2007.
8. A.S. Krishna, E. Turkay, A. Gokhale, and D.C. Schmidt, “Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems,” 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS’05), IEEE Computer Society, 2005, pp. 180-189.
9. M. Merdan, T. Moser, D. Wahyudin, and S. Biffl, “Performance evaluation of workflow scheduling strategies considering transportation times and conveyor failures,” International Conference on Industrial Engineering and Engineering Management (IEEM 2008), IEEE Computer Society, 2008, pp. 389-394.
10. P. Oreizy, N. Medvidovic, and R.N. Taylor, “Architecture-Based Runtime Software Evolution,” International Conference on Software Engineerings (ICSE 2008), IEEE Computer Society, 2008, pp. 177-187.
11. M. Völter, and T. Stahl, Model-driven Software Development, John Wiley, 2006.
12. J. Wuttke, “Runtime failure detection,” Companion of the 30th International Conference on Software Engineering, ACM, 2008, pp. 987-990.


## Biographies


***Stefan Biffl*** *is an associate professor of software engineering at the Institute of Software Technology and Interactive Systems, Vienna University of Technology. He received MS and PhD degrees in computer science from the Vienna University of Technology and an MS degree in social and economic sciences from the University of Vienna He received an Erwin-Schrödinger research scholarship for research at the Fraunhofer Institute of Experimental Software Engineering, focusing on quality management and empirical software engineering. In 2006 he worked as guest researcher at Czech Technical University, Department of Cybernetics on process improvement for mission-critical automation systems.*
*Contact him at stefan.biffl@tuwien.ac.at.*

***Thomas Moser*** *received his master degree in business informatics at the Vienna University of Technology and works as a PhD researcher in the research area “Complex Systems” since 2007 and project manager of the “Simulation of Assembly Workshops“ (SAW) project that develops advanced multi-agent system software simulators for production automation simulation.. His main research areas*

*are data modeling and semantic web technologies for model transformations to connect design-time and run-time engineering processes.*
*Contact him at thomas.moser@tuwien.ac.at.*

***Alexander Schatten** is a senior researcher at the Institute of Software Technology and Interactive Systems, Vienna University of Technology. He received his Master degree in technical analytical chemistry and his PhD in computer science from Vienna University of Technology. His main research area is software engineering with emphasis on event-driven architecture and open source technologies and practices. He is additionally working on contributions of ICT for sustainable development.*
*Contact him at alexander.schatten@tuwien.ac.at.*

***Wikan Danar Sunindyo** received his master degree in computational logic at the Dresden University of Technology, Germany, and works as a PhD researcher at the Vienna University of Technology in the research area "Complex Systems" since 2008. His main research areas are data warehousing and semantic web technologies to better integrate heterogeneous engineering environments.*
*Contact him at wikan@ifs.tuwien.ac.at.*