# Toward a new landscape of systems management in an autonomic computing environment

by G. Lanfranchi
P. Della Peruta
A. Perrone
D. Calvanese

In this paper we present IBM Tivoli Monitoring, a systems management application that displays autonomic behavior at run time, and we focus on extending it in order to encompass the design and the deployment phases of the product life cycle. We review the resource model concept, illustrate it with an example, and discuss its role throughout the product life cycle. Then we introduce basic concepts in ontology and description logics and discuss representing Common Information Model constructs using description logics. Finally, we propose Systems Management Ontology, an approach to enhancing the autonomic properties of IBM Tivoli Monitoring based on an ontology service and the technique of "contextual pulling" applied to the resource model.

*Problems cannot be solved at the same level of awareness that created them.*

—Albert Einstein

The information technology (IT) infrastructure that supports business systems today continues to evolve at a breakneck pace, with the integration of new devices, servers, and applications creating highly complex systems. Systems management needs to similarly evolve in order to cope with this increasing complexity in IT. Management tools need to become "smarter" if they are to drive successful business systems. Performance and availability management tools are, in particular, key enablers of an efficient and profitable business. A business cannot execute efficiently unless the mission-critical business applications and the supporting middleware and operating systems are available and performing well. Any component failure or poor performance could adversely impact the business.

To minimize the business downtime, the speedy execution of the appropriate corrective action is needed. An important component of performance and availability management is the monitoring of the system in order to detect anomalies as soon as they occur and to take the necessary corrective actions (e.g., restore the failing objects to their desired state, or activate backup resources). The monitoring and the taking of corrective action should be optimized with respect to the overall objectives of the entire business system. Whenever feasible, systems management tools with predictive capabilities should be used in order to detect future problem states, and to allow action to be taken before a failure occurs and adversely impacts users.

When the tools monitoring the system resources simply collect raw data (e.g., performance metrics), it can be difficult to draw any conclusion about the health of a system. A graphical performance monitor or an event viewer tool can be useful, but only if a skilled administrator is able to interpret the information provided, and to determine if a problem exists or not. Only an experienced administrator has the ability to correlate the appropriate domain

knowledge with the data collected, such as metrics and events, and come up with answers to questions such as: Is there a memory bottleneck?, What is its cause?, and How can I fix it? In the current monitoring environment, the system administrator inherently owns the best practices and applies them in order to identify and cure problems.

We describe here the approach taken in IBM Tivoli Monitoring,[1] in which the monitoring tool directly models and implements the relevant aspects of the domain entities as logical objects, and via this implementation it transforms the raw data into the information required in order to detect, correct, and, whenever possible, predict abnormal system behavior. The system administrator is thus better able to address IT problems quickly, and thus concentrate on better serving business demands.

The best practices embedded in the monitoring tool enable the autonomic behavior of the monitoring tool at run time.[2] This autonomic behavior can be extended to the other two phases of the application life cycle: the design phase (where the best practices are created), and the deployment phase (where the best practices are deployed into the managed systems).

In this paper we first present IBM Tivoli Monitoring as an existing solution that displays autonomic behavior at run time, and then we focus on extending this solution to encompass the design time and the deployment time. In the next section, we provide an overview of IBM Tivoli Monitoring with a particular focus on the "resource model" concept. In the following section we present the Systems Management Ontology with an overview of "description logics" and their use in representing Common Information Model constructs. Finally, we link the previous themes in order to illustrate the proposed approach for autonomic systems management.

## The IBM Tivoli Monitoring solution

The IBM Tivoli Monitoring (ITM) solution is based on the resource model concept. In this section we define the resource model and illustrate it through an example. We then discuss the application of the resource model to all phases of the ITM life cycle.

**The resource model concept.** The resource model is the main tool for implementing an "identify, notify, and cure" systems management strategy. A resource model has two parts: a dynamic model and a reference model.

The Common Information Model (CIM) formalism promoted by the Distributed Management Task Force is a way to organize the available information about the managed environment that applies the basic structuring and conceptualization techniques of the object-oriented paradigm.[3] The approach uses a uniform modeling formalism that, together with the basic repertoire of object-oriented constructs, supports the cooperative development of an object-oriented schema across multiple organizations. The *dynamic model* uses CIM classes and CIM properties to describe resources (such as memory) and their performance metrics (such as process working set). Moreover, it uses CIM methods to describe actions that can be executed against the resource (such as starting a process). The CIM association is then used to represent the resource within a high level object (logical object).

The *reference model*, implemented as a decision procedure and coded using either Visual Basic** or JavaScript**, incorporates best practices that:

- Interpret the status of a problem object against a defined baseline using the metrics provided by the dynamic model
- Discover the root cause of the problem
- Correct the problem
- Log performance data related to the domain objects

The reference model describes the "critical paths" within systems or applications. It interprets the "quality" of applications and systems against a predefined service level in order to discover the root cause of the problem and to react accordingly. The reference model has a direct link to the dynamic model describing the resources and it analyzes the properties of objects aggregated in the model, generates indications about the status of those resources, and invokes methods to correct a problem. The reference model, in effect, implements the best practices normally used by a system administrator to detect problems and to identify their root cause.

A resource model is created for each problem and it contains the best practices used to identify and correct a well-defined problem. Figure 1 contains an example of a resource model used to identify memory problems in a Microsoft Windows** machine. The dynamic model is created using a CIM representa-

Figure 1 Resource model for detecting memory problems in Microsoft Windows machines

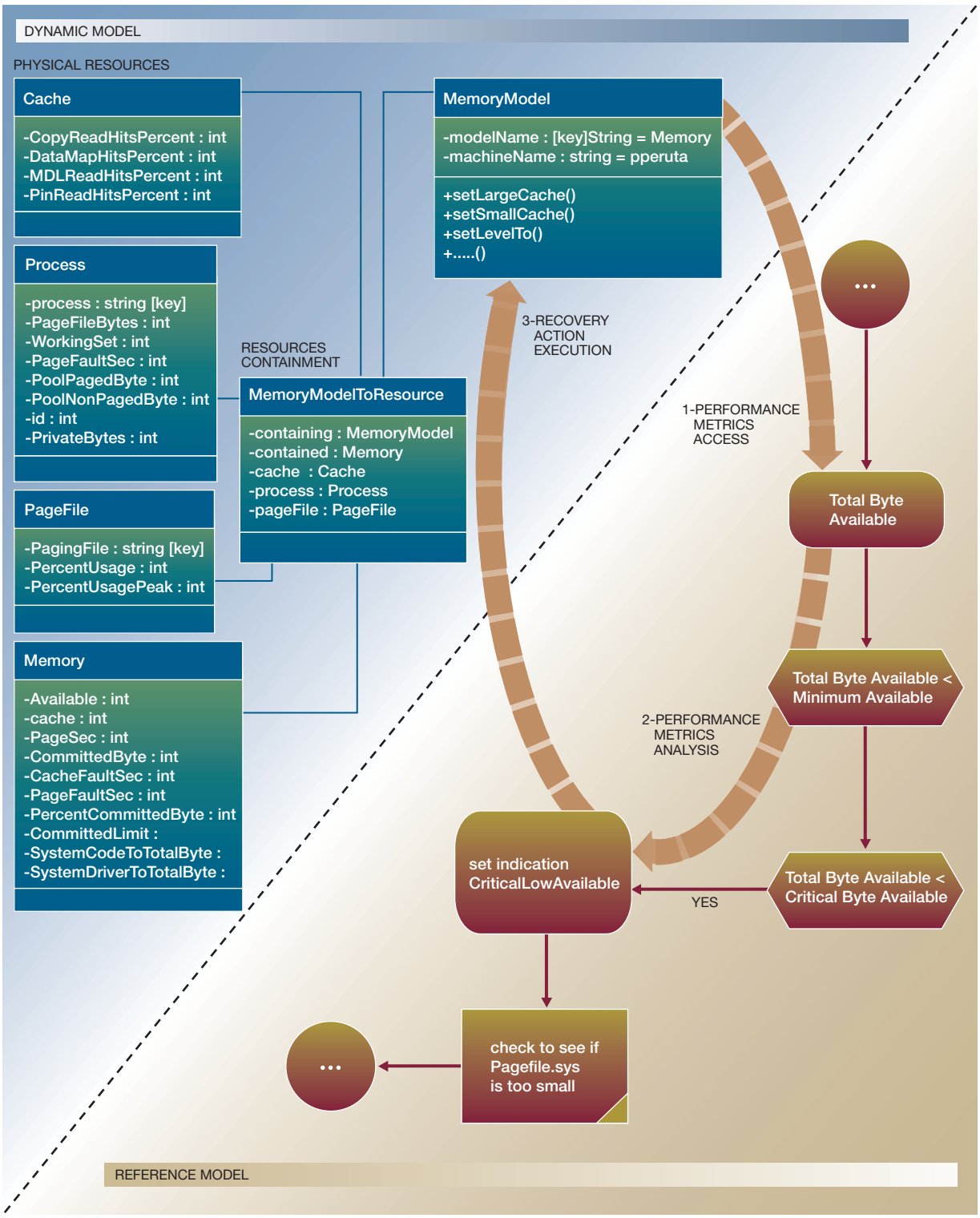
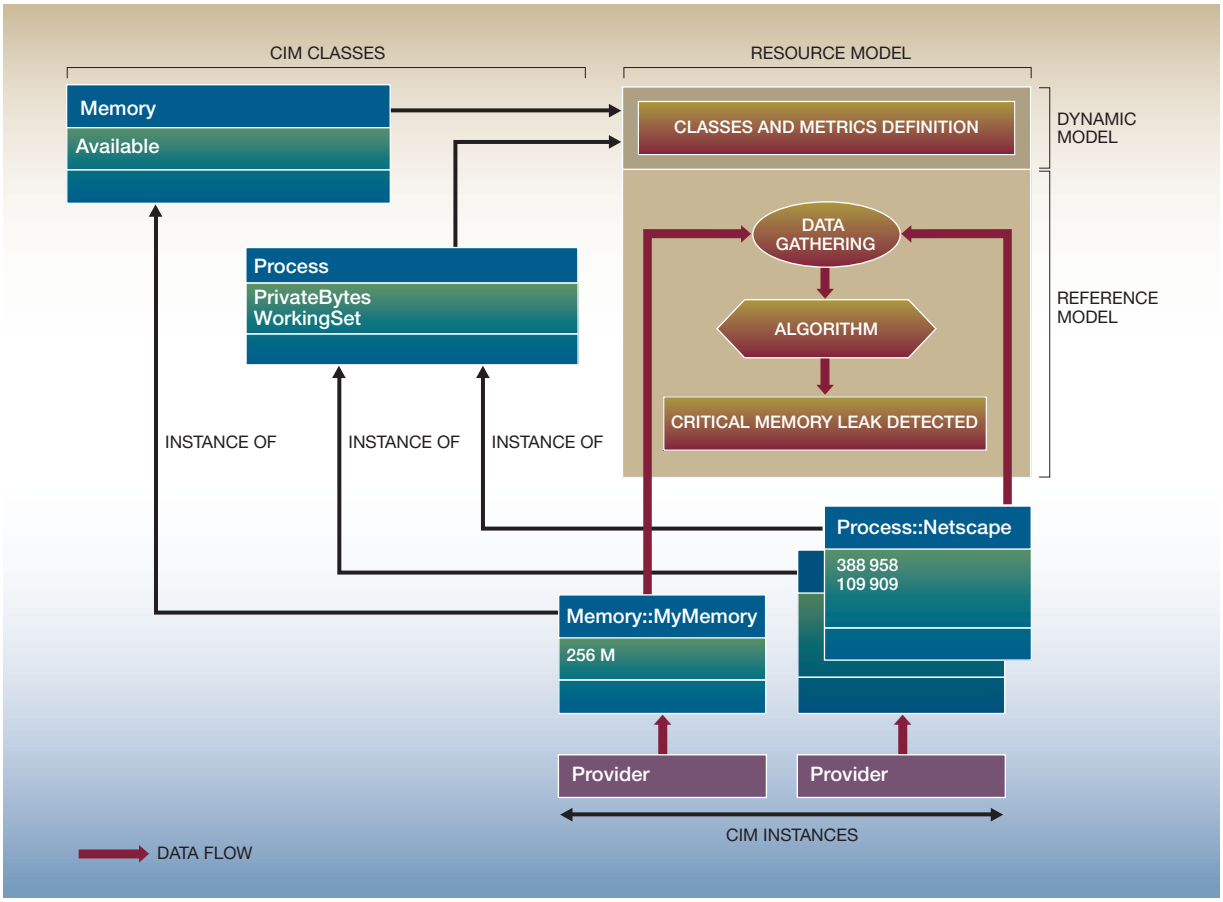


Figure 2 A simplified resource model for detecting critical memory leaks




tion of the resources (Cache, Process, PageFile, Memory) and the properties that are useful in detecting memory problems. These classes are then associated via association class MemoryModelToResource with object MemoryModel. Class MemoryModel also contains a set of methods, such as setLargeCache, that can be used to modify the status of some resource (e.g., Cache).

The reference model is linked to class MemoryModel (and thus to the dynamic model) and, using this class, it can access the performance metrics of the associated resources and apply the encoded best practices in order to discover if a memory problem exists. When a problem is detected, the reference model can invoke a well-defined method in MemoryModel to fix the problem, and also to notify the system operator about the problem resolution. The best practices are implemented in the form of an algorithm that analyzes and correlates metrics, and compares the results against a well-defined baseline.

Let us analyze a simplified resource model for detecting critical memory leaks (see Figure 2). Classes Memory and Process, as well as other additional classes not shown in Figure 2, constitute the dynamic model. MyMemory is an instance of class Memory, whereas Netscape is an instance of class Process. Notice that property Available of class Memory, and properties PrivateBytes and WorkingSet of class Process are represented in class instances MyMemory and Netscape by their numerical values, respectively 256, 388958, and 109909. The function of layer Provider is to gather the raw metrics (from the actual resources) in order to supply the above-mentioned values for properties. The reference model controls the

data gathering and interprets the data according to best-practices algorithms in order to detect the critical "memory leak" condition.

A "critical memory leak" occurs when the conditions "low available memory with high working set" and "memory leak in private bytes" exist. The condition "low available memory with high working set" can be detected by examining the number of bytes used for the working set of the process (property bytTotalWorkingSet), the number of bytes used as cache (bytTotalCache) and the total memory available (bytTotalAvail). Those three values can be calculated using "raw" metrics as follows.

- List all the process working sets and store the high working set in bytTotalWorkingSet
- Store the metric Cache from Memory object in bytTotalCache
- Store the metric Available from the Memory object in bytTotalAvail

The values collected for system resources are further processed in order to detect the "low available memory with high working set" condition.

```
bytTotalRAM = bytTotalWorkingSet +
  bytTotalCache + bytTotalAvail
numPercentWS =
  (bytTotalWorkingSet/bytTotalRAM) * 100
numPercentCache =
  (bytTotalCache/bytTotalRAM) * 100
numPercentAvail =
  (bytTotalAvail/bytTotalRAM) * 100
```

If (numPercentWS > numPercentCache and numPercentWS > numPercentAvail) then the condition "low available memory with high working set" is satisfied.

The condition "memory leak in private bytes" is generated from a metric of the Process resource PrivateBytes and observing its growth over time.

Store the process private bytes in PrivateBytes.

If the current value of process private bytes > the previous value of the process private bytes then the condition "memory leak in private bytes" is satisifed.

As shown, modeling the "critical memory leak" condition requires metrics provided by the CIM classes Memory and Process, and the above algorithm.

**The design, deploy, and run life-cycle phases.** The complete life cycle of the application, and of the resource model, consists of three phases: the design phase, the deployment phase, and the run-time (or operational) phase.

During the design phase, the resource model is designed, built, and tested. IBM Tivoli Monitoring supports this phase by providing an integrated development environment known as the Workbench. This is the phase when best practices are implemented and the domain knowledge is represented in the dynamic model and the reference model. Workbench users (developers, administrators, etc.) select the CIM classes for the dynamic model and then code the algorithms that analyze the resource metrics in order to detect and fix any abnormal behavior.

At deployment time, the resource model produced using the Workbench is installed in the management server from where it can be deployed to the target machines using the profile managers of the Tivoli Management Environment.[4,5]

On the target machine the resource model runs within ITM. Here, instances of classes included in the dynamic model are located, and the values of the properties for each of these objects are analyzed by the reference model. Abnormal conditions and performance degradation are detected and corrected. The information produced by the resource model and the performance metrics collected can be retrieved and analyzed using the ITM Web Health Console.[1]

**The next step.** Although the operational phase of monitoring displays autonomic properties—the resource model is able to keep resources under control with reduced human intervention—the design and deployment phases still require human intervention. Work is under way in the two following aspects to implement a more autonomic approach.

1. Apply automated reasoning procedures to the building of resource models. These procedures would help identify common knowledge, avoid redundancy, and optimize the set of resource models needed to manage a system or an application.
2. Provide a capability of the managed resource to "contextually pull" an appropriate set of resource models needed for its control. This will allow the management agent itself to pull the resource models from the central server, thus avoiding the need for the administrator to configure each monitor-

ing agent according to the exact set of applications present on the machine, and to push to the target the right set of resource models.

The next section highlights our blueprint for addressing these two objectives.

## Systems Management Ontology

In this section we introduce basic concepts in ontology and description logics in the context of Systems Management Ontology (SMO) and discuss representing CIM models in description logics. We use the memory model example to highlight the application of description logics reasoning capabilities to resource models.

**Ontologies and description logics.** *Ontologies* are metadata schemas, providing a controlled vocabulary of terms, each with an explicitly defined and machine understandable semantic.[6] They provide a shared and common understanding of a domain that can be communicated to people and intelligent systems, and thus facilitate knowledge sharing and reuse. Ontologies are intended to overcome the problem of implicit and hidden knowledge by making the conceptualization of a domain explicit. Researchers in artificial intelligence advocate expressing ontologies in logical terms, which makes them suitable for automated reasoning. Thus, information sharing and reuse are facilitated not only by the explicit representation of knowledge provided by ontologies, but also by the automated reasoning services that allow one to infer consequences from the available knowledge.

Nowadays there is consensus on structuring knowledge on a domain of interest in terms of classes of objects and relationships between classes. State-of-the-art ontology languages such as OIL and DAML+OIL[7] follow exactly this point of view. The need to model domains in terms of classes and relationships and the use of automated reasoning on such representations leads to *description logics* being offered as an ideal candidate for capturing ontologies on a given domain in logical terms. Indeed, both OIL and DAML+OIL can be viewed formally as dialects of description logics.

Description logics were introduced in the early 1980s in the attempt to provide a formal foundation to semantic networks and frame systems. Since then they have evolved into knowledge representation languages that are able to capture virtually all class-based representation formalisms used in artificial intelligence, software engineering, and databases.[8] One of the distinguishing features of the work on these logics is the detailed computational complexity analysis, both of the associated reasoning algorithms and of the logical implication problem that the algorithms are supposed to solve. Practical systems implementing such algorithms are now used in several projects.[9] Generally speaking, in description logics, the domain of interest is modeled by means of *concepts* (unary predicates) and *relations* (n-ary predicates), which denote classes of objects and relationships, respectively. Description logics are characterized by three basic components.

1. A *description language*, which specifies how to construct complex concept and relation expressions, by starting from a set of atomic concepts and relations and by applying suitable constructs
2. A *knowledge specification mechanism*, which specifies how to construct a description logic *knowledge base*, in which properties of concepts and relations are specified by means of suitable *assertions*
3. A set of *reasoning procedures*, which allow one to carry out suitable inferences from the knowledge expressed in the knowledge base

One of the most expressive description logics studied so far is DLR.[10–12] DLR is equipped with most of the concepts and relation constructs typical of decidable description logics (concrete domains are a notable exception) and allows for the most general form of knowledge assertions while still admitting decidable reasoning procedures.

**Representing CIM in description logics.** DLR has been successfully applied to representing UML (Unified Modeling Language) class diagrams[13] and to reasoning with them. Intuitively, classes are represented by atomic concepts, attributes, and aggregations by atomic (binary) relations, and each association of arity $n$ by an atomic concept and $n$ atomic binary relations. The latter allows for dealing with associations, with and without a corresponding association class, in a uniform way. The semantics of the various UML constructs is captured by suitable DLR assertions. We refer the reader to Reference 14 for the details of the encoding, while pointing out that the expressive power of DLR allows one to formalize several types of constraints that produce a better representation of the application semantics and that are typically not dealt with in a formal way. By making use of the encoding of UML class diagrams in DLR,

the following reasoning activities can be reduced to DLR reasoning tasks.[15,16]

- Consistency of the class diagram, that is, whether the diagram admits an instantiation. If this is not the case, there is no set of instances that satisfies the diagram, which indicates that the definitions altogether are inconsistent.
- Class consistency (association/aggregation consistency), that is, whether there is an instantiation of the diagram such that a given class (association/aggregation) has a nonempty extension. Observe that, if this is not the case, then there is an inconsistency in the class specification, or at the very least the class is inappropriately named since it is a synonym for the empty class.
- Class subsumption (association/aggregation subsumption), that is, whether the extension of one class (association/aggregation) is a subset of the extension of another class (association/aggregation) in every instantiation of the diagram. This property suggests the possible omission of an explicit generalization. Alternatively, if all instances of the more specific class are not supposed to be instances of the more general class, then something is wrong in the rest of the diagram, since it is forcing an undesired conclusion.
- Detection of redundancies, for example, if two classes (associations/aggregations) subsume each other, then one of them is redundant.
- Refinement of properties, for example, when the properties of various classes and associations/aggregations interact to yield stricter multiplicities or typing than those explicitly specified by the designer. The simplest cases arise with multiple inheritance.

The idea of using description logics to deal with resource models has its root in two considerations: (1) Both CIM models and resource models are expressed using UML class diagrams with specific conventions, and (2) DLR is able to fully capture UML class diagrams together with the specific conventions adopted in CIM. The approach can be summarized in the following way:

- The CIM core model and the parts of the common model relevant for a specific management task are expressed in terms of a DLR knowledge base.
- CIM-based resource models (actually, the dynamic model part) that are specific for components under analysis are in turn expressed as additional assertions included in the knowledge base.
- Queries posed to the CIM system specification are translated into queries to the corresponding DLR knowledge base, and are answered by making use of typical description logics reasoning tasks.

Representing CIM and resource models in terms of DLR allows for automated use of the knowledge represented in such models, making use of state-of-the-art reasoning tools developed for expressive description logics.

**Resource model knowledge exploitation.** The expressive power of description logics and their ability to capture and model a great variety of constraints allow for powerful reasoning operators to be applied across the CIM classes and CIM class properties representing a particular IT system. (Note that the standard UML representation of CIM schema does not allow this type of reasoning across classes.) We have successfully applied the capabilities offered by description logics in our work focused on the "automatic" definition of the dynamic model (aggregation of CIM classes) during the design phase. In our validation work we used two state-of-the-art description logic reasoning systems: FaCT[17,18] and RACER;[19] we formalized a CIM core model knowledge base given as input to FaCT and RACER in order to apply reasoning and inference operators. This technology appears suited to provide the reasoning engine for the resource model ontology service.

Referring again to the critical memory leak example (see Figure 2), CIM classes, attributes, and associations/aggregations are used in the resource model (actually, the dynamic model) to establish relationships between the various system components. Such relationships implicitly determine those that may contribute to the definition of a problem (e.g., "critical memory leak" between Memory and Process classes). If this model is loaded on the ontology server, the description logic reasoning engine can exploit it to determine relevant properties to be included in other resource models.

As an example, consider a new application installed on the managed endpoint with the requirement that resource leaks should be kept under control. This new application can be viewed as being composed of multiple processes. The monitoring engine asks the reasoning engine (by invoking suitable description logic inference requests) to identify the system components that may be involved in a resource leak. The reasoning system determines those classes that are subsumed by the description logic term representing the resource leak. In our example this is the

"critical memory leak" condition via the Process class. Thus, the model for the application includes the classes Memory and Process and the resource model class representing "critical memory leak." Other kinds of resource leaks could be similarly identified, such as "file descriptor leak" or "queue handle leak."

Further research into System Management Ontology aims at extending the description logic formalization from the domain of CIM classes and properties (dynamic model) to the domain of "best practices" and baselines (reference model). This may require extending the expressiveness of the adopted description logics (to include, for example, concrete domains) or taking into account the representation of extensional aspects and/or more expressive querying mechanisms.

The implications of this ongoing research effort are fascinating and challenging: we could apply reasoning mechanisms not only across classes and class properties but also across resource models. The reasoning mechanisms working across resource models could gracefully extend from the autonomic management of a single element (such as an operating system, a database, a business application) toward the autonomic management of an entire business solution.

In fact, because the resource model is fully represented via description logics, the ontology service can be used for reasoning across resource models, for detecting and correcting problems, for optimizing system behavior, or for achieving a specified quality-of-service (QoS). The derived resource model, or set of resource models, would then be "contextually pulled" to the relevant resource.

The use of description logics to represent and perform reasoning across resource models significantly augments the autonomic features of the IBM Tivoli Monitoring solution during the design and deployment phases. Figure 3 illustrates the basic flow when a managed resource uses SMO services.

Every time the managed resource undergoes a significant state change (such as when a new application is installed, or a new QoS is enforced on a running application) the monitoring agent uses the ontology service for reasoning about the appropriate management model for enforcing the desired QoS. This phase is controlled by means of user-defined policies that define the configuration for the monitoring agent, for example, "call the ontology service whenever application X is installed or when QoS is modified" (see items 1 through 3 in Figure 3).

The end result of the reasoning process performed by the ontology engine is the resource model (or set of resource models) best suited to address the specific condition at the target (e.g., "application X with QoS Q has been installed"). Default configuration parameters (cycle time, thresholds, etc.) are also applied to complete the definition of the reference model[20] (see item 4 in Figure 3).

The target knows (usually a URL is provided) where the resource model is located, or where it will be built. The resource model is then pulled to the target, typically in a Web environment (item 5 in Figure 3), where it exercises its problem-detecting capabilities (item 6 in Figure 3).

The described contextual pulling approach, with the support of an ontology service, represents a step toward the autonomic computing vision:

- The resource model can be selected and built, without human intervention, based on the context (state changes, specified QoS, etc.).
- The resource model is pulled "on demand" according to the state of the target.

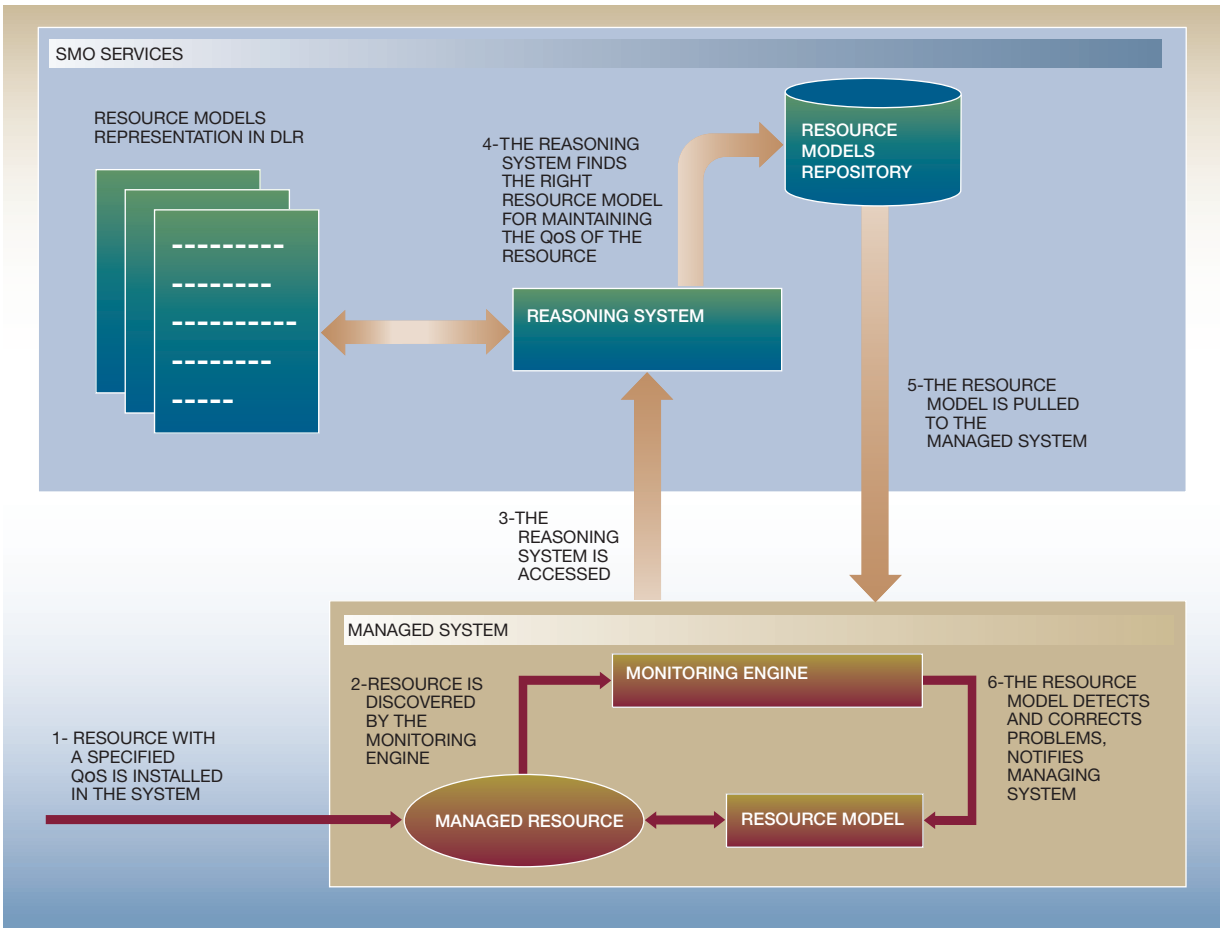
These enhanced autonomic capabilities do not intend to exclude completely the human intervention from the monitoring process. The human administrator will continue to be involved in the definition of the appropriate policies, in the supervision of the entire process, and in the tuning/correcting of specific "pathological" conditions.

## Conclusions

The resource model technology, currently implemented in the ITM product, helps realize the autonomic computing vision in systems management and specifically in the monitoring space: the resource model captures the key characteristics of a managed resource and allows the system to detect and correct any behavior not in line with the expected QoS.

In this paper, we proposed to use the resource model technology in order to extend the autonomic behavior across the entire life cycle of the application, including the design phase and the deployment phase. The CIM ontology and its representation by description logics raises the possibility to make better use

Figure 3 A managed resource using SMO services—the basic flow




of the knowledge available in CIM itself, and ultimately to perform automated reasoning on this knowledge base system. Finally, the contextual pulling approach is seen as an autonomic evolution of the management paradigm offered today by the Tivoli Management Environment:[5] it is the managed resource that, through the use of the SMO service, will be able to find and deploy the right resource model for keeping its QoS under control.

## Acknowledgments

The authors wish to thank Giuseppe De Giacomo, Richard Szulewski, Scot MacLellan, Daniela Berardi, Andrea Cali, and Maurizio Lenzerini for sharing their insight and for their suggestions on the topic of this paper.



## Cited references and notes

1. IBM Tivoli Monitoring, IBM Corporation, http://www.tivoli.com/products/index/monitor/.
2. Autonomic Computing, IBM Research Division, IBM Corporation, http://www.research.ibm.com/autonomic/.
3. The Distributed Management Task Force, Inc., http://www.dmtf.org/index.php.
4. Tivoli Management Framework 3.7.1, IBM Corporation, http://www.tivoli.com/support/public/Prodman/public_manuals/td/ManagementFramework3.7.1.html.
5. Tivoli Management Environment provides centralized management and a single point of control for data distribution to groups of systems. In the Tivoli environment, a *profile* provides a container for application-specific information about a particular type of resource. Profiles can be specialized and configured, distributed across a network, and applied to a di-

verse set of machines, according to a subscription paradigm. The profile feature is scalable with the number of distributed systems.
6. D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer, New York (2001).
7. D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "OIL: An Ontology Infrastructure for the Semantic Web," *IEEE Intelligent Systems* **16**, No. 2, 38–45 (2001).
8. D. Calvanese, M. Lenzerini, and D. Nardi, "Unifying Class-Based Representation Formalisms," *Journal of Artificial Intelligence Research* **11**, 199–240 (1999).
9. *The Description Logic Handbook: Theory, Implementation and Applications*, F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, Editors, Cambridge University Press (2002). To appear.
10. D. Calvanese, G. De Giacomo, and M. Lenzerini, "On the Decidability of Query Containment under Constraints," *Proceedings of the 17th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems* (PODS'98), ACM, New York (1998), pp. 149–158.
11. D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, "Description Logic Framework for Information Integration," *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (KR'98), Morgan Kaufmann Publishers, San Francisco, CA (1998), pp. 2–13.
12. D. Calvanese, G. De Giacomo, and M. Lenzerini, "Identification Constraints and Functional Dependencies in Description Logics," *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (IJCAI 2001), Morgan Kaufmann Publishers, San Francisco, CA (2001), pp. 155–160.
13. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Publishing Co., Boston, MA (1998).
14. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "A Formal Framework for Reasoning on UML Class Diagrams," *Proceedings of the 13th International Symposium on Methodologies for Intelligent Systems* (ISMIS 2002), Lecture Notes in Computer Science 2366, Springer, New York (2002), pp. 503–513.
15. D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML Class Diagrams Using Description Logic Based Systems," *Proceedings of the KI'2001 Workshop on Applications of Description Logics*, CEUR Electronic Workshop Proceedings (2001), http://ceur-ws.org/Vol-44/.
16. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "Reasoning on UML Class Diagrams in Description Logics," *Proceedings of IJCAR Workshop on Precise Modelling and Deduction for Object-Oriented Software Development* (PMD'01), (2001), http://i12www.ira.uka.de/~beckert/PMD/proceedings.html.
17. I. Horrocks, "The FaCT System," H. C. M. de Swart, Editor, *Proceedings of the 2nd International Conference on Analytic Tableaux and Related Methods* (TABLEAUX'98), Lecture Notes in Artificial Intelligence 1397, Springer, New York (1998), pp. 307–312.
18. I. Horrocks, "Using an Expressive Description Logic: FaCT or Fiction?" *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (KR'98), Morgan Kaufmann Publishers, San Francisco, CA (1998), pp. 636–649.
19. V. Haarslev and R. Moeller, "RACER System Description," *Proceedings of the International Joint Conference on Automated Reasoning* (IJCAR 2001), Lecture Notes in Artificial Intelligence 2083, Springer, New York (2001), pp. 701–705, http://www.globus.org/OGSA
20. The most appropriate values for the configuration parameters heavily depend on the operational conditions at the target. In ITM Version 5.1 a default setting is given, which the user may change during the configuration phase. In the future, we envision an Adaptive Manager running at the target whose function is to tune the configuration parameters of the resource models according to the operational conditions of the hosting environment.

*Accepted for publication September 20, 2002.*

**Giovanni Lanfranchi** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: Giovanni_Lanfranchi@it.ibm.com).* Mr. Lanfranchi is currently technical strategist for the Performance and Availability department in Tivoli. He received his degree in physics in 1986 at the University of Milan. Before joining IBM he was responsible for research and development at several companies in the robotics and automotive field. At IBM he was an architect in the Computer-Aided Engineering department, where he developed leading-edge technologies in the surface modeling area. Mr. Lanfranchi is currently with the IBM Tivoli Division where he is involved in the design of the "new wave" of systems management applications with specific focus on performance and availability.

**Pietro Della Peruta** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: pietro_della_peruta@it.ibm.com).* Mr. Della Peruta is currently principal engineer in the performance and availability unit of IBM Tivoli. In this role he is responsible for technical strategy and implementation of performance and availability management applications. Previously he worked on the design of the IBM Tivoli Monitoring product. He joined Tivoli in 1996 after six years as technical architect of IBM performance products such as NetView Performance Monitor. He holds a B.A. degree in electronics engineering from the University of Naples.

**Antonio Perrone** *IBM Software Group, Rome Tivoli Laboratory, Via Sciangai 53, I-00144 Roma, Italy (electronic mail: antonio.perrone@it.ibm.com).* Mr. Perrone graduated *cum laude* in information sciences in 1989 at the University of Bari, Italy. He joined IBM in 1990, where he has held several positions in development and support. He has been with the IBM Tivoli Division since 1996, where he is currently an architect of IBM Tivoli Monitoring. His interests include systems management, performance, and availability, knowledge representation, data integration, transaction and process modeling, and software development.

**Diego Calvanese** *Università di Roma "La Sapienza," Dipartimento di Informatica e Sistemistica, Via Salaria 113, I-00198 Roma, Italy (electronic mail: calvanese@dis.uniroma1.it).* Dr. Calvanese is Assistant Professor in the Department of Informatics and Systems of the University of Rome, where he received his Ph.D. in computer science in 1995. His research interests include theoretical aspects of data and information integration, semi-structured data, logics for knowledge representation, and in particular description logics and their relationship to data models.