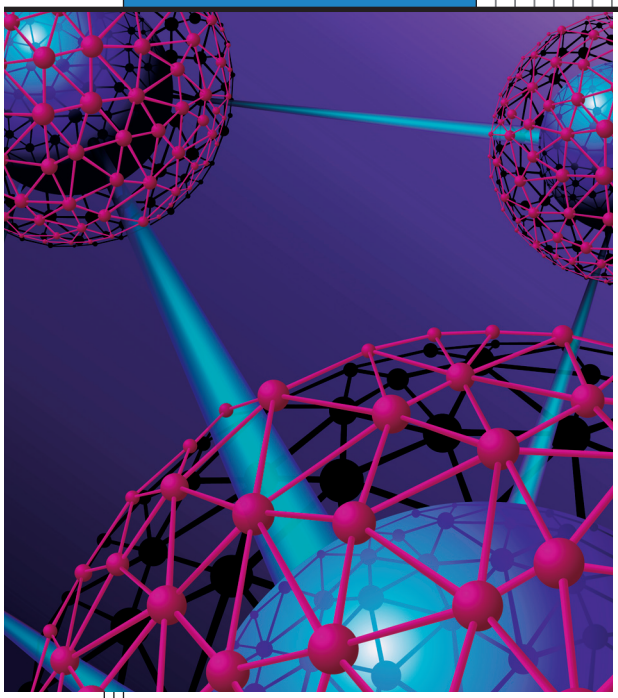

EVOLVING SOFTWARE ARCHITECTURE DESCRIPTIONS OF CRITICAL SYSTEMS

Tom Mens, *Université de Mons, Belgium*
Jeff Magee, *Imperial College London, UK*
Bernhard Rumpe, *RWTH Aachen University, Germany*

To manage the complexity of developing, maintaining, and evolving a critical software-intensive system, its architecture description must be accurately and traceably linked to its implementation.

Software-intensive systems, whether real-time embedded systems or information-processing systems, present critical concerns for stakeholders. A system may be mission-critical for a company, in that it could lose its competitive advantage or even be unable to survive if the system doesn't function properly. A system may be resource-critical in terms of time, personnel, hardware, or any other crucial resource on which it might rely; unavailability or malfunction of these resources could cause the system to fail. A system may be critical in a more traditional sense—having specific nonfunctional characteristics that must be satisfied at all times. For example, financial systems are security-critical, whereas nuclear power plants, medical applications, and public transportation are safety-critical, as human lives might be at stake.

Software architectures provide a sound basis for explicitly documenting these concerns. IEEE standard 1471-2000, which has also become ISO/IEC 42010:2007, recommends providing architectural descriptions of software-intensive systems to cope with their increasing complexity and to mitigate the risks incurred in constructing and evolving these systems. According to this standard,[1] as Figure 1 shows, a system fulfills a particular mission in the environment it inhabits and has one or more stakeholders that have concerns relative to the system and its mission. Concerns are defined as "those interests that pertain to the system's development, its operation, or any other aspects that are critical or otherwise important to one or more stakeholders." Runtime concerns include performance, reliability, security, and distribution; development concerns focus on maintenance—in particular, evolvability.

The software architecture deals with multiple views of a system including both its functional and nonfunctional aspects. A structural view looks at the system as a set of components that interact via connectors. Complexity is mastered by means of hierarchical decomposition; a component can be composed from subcomponents with the hierarchy's leaf components representing coded functionality. As the "Architecture Description Languages" sidebar describes, the research community has proposed numerous ADLs, some of which have found their way into commercial practice.

An explicit architecture description is important but not sufficient to manage the complexity of developing, maintaining, and evolving a critical software-intensive

Published by the IEEE Computer Society

[MMR10] T. Mens, J. Magee, B. Rumpe
Evolving Software Architecture Descriptions of Critical Systems
In: Computer vol. 43, IEEE Computer Society, May 2010.
www.se-rwth.de/publications/

system. The description must also be accurately and traceably linked to the software's implementation, so that any change to the architecture is reflected directly in the implementation, and vice versa. Otherwise, the architecture description will become rapidly obsolete as the software evolves to accommodate changes. The architecture description must thus be an integral part of the software-intensive system and its documentation.

WHY EVOLVE ARCHITECTURE DESCRIPTIONS?

Any software-intensive system is constantly subject to software changes, usually driven by external stimuli from the system environment over which the developers have little or no control. These stimuli may be as diverse and unforeseeable as technological changes, enhanced user

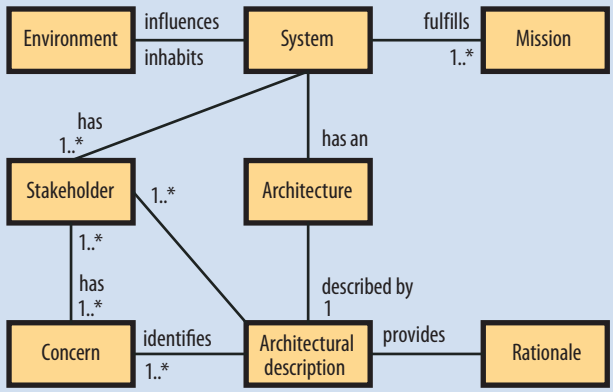

Figure 1. Fragment of IEEE Std. 1471 conceptual model of architectural description. A software-intensive system fulfills a particular mission in the environment it inhabits and has one or more stakeholders that have concerns relative to the system and its mission.

ARCHITECTURE DESCRIPTION LANGUAGES

ADLs have emerged as formal languages to define and document the software architecture of systems.[1-4] They facilitate communication between software architects and other stakeholders and make it possible to express, verify, and impose properties upon the software that will implement the architecture. In contrast to programming languages, ADLs are usually declarative and describe a system's architecture as a set of components, connectors, and configurations of these elements.

Researchers have developed numerous ADLs such as AADL (Architecture Analysis and Design Language), Acme, COSA (Component Object-based Software Architecture), Darwin, Rapide, and Wright. Appropriate architecture-centric software development tools have also been developed, including ArchStudio, AcmeStudio, and SafArchie Studio.

Koala[5] is one of the few ADLs to have found application in commercial practice. Philips uses it to define the software architecture for consumer electronic products. Koala is model-driven in that it directly uses the architectural description to construct the software loaded into products.

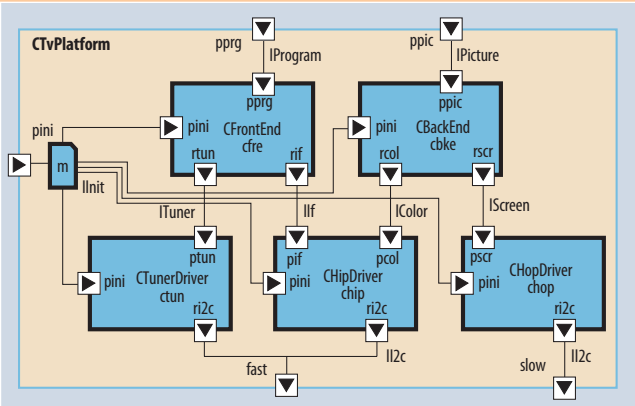

Figure A. Architectural description of the software for a TV set using Koala. The components can be configurations of more primitive components or they can be base-level components with their implementations defined in C.

Figure A[5] shows an example of the architectural description of the software for a TV set using Koala. The components shown in the figure can be configurations of more primitive components or they can be base-level components with their implementations defined in C. This ability to describe systems as hierarchical compositions of components is the key to managing complexity and is a feature of practically all ADLs.

In the figure, the boxes with arrows represent interfaces defined by sets of function calls. If the arrow points into a component, then the component provides or implements that interface; if it points out of the box, then the component requires access to the interface. The lines or connectors represent connections between required and provided interfaces and represent runtime function call paths. Connectors in other ADLs represent more general connector semantics that can encompass streams, events, and message-passing protocols.

Koala restricts itself to a structural description of software architecture. However, much of the power of ADLs and their importance to critical systems arises from the ability to associate behavioral, functional, and nonfunctional properties with components and reason about the preservation of overall system properties.

With the advent of Unified Modeling Language v. 2.x, more modern ADL proposals are essentially profiles that extend UML 2.x by means of stereotypes to extend the existing UML 2.x structural elements with additional properties and constraints.

References

1. R.N. Taylor, N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, Wiley, 2009.
2. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
3. N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.*, Jan. 2000, pp. 70-93.
4. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
5. R. van Ommering et al., "The Koala Component Model for Consumer Electronics Software," *Computer*, Mar. 2000, pp. 78-85.

organizational structures or business processes, new legislation, or changes in resources.

To cope with any of these issues, all software artifacts produced and used by the software-intensive system must evolve. Depending on the software artifacts' type and granularity, the impact and rate of change may differ. Source-code artifacts need to be changed frequently—for example, to fix bugs—but often have a local impact only. Changes to the architecture occur less frequently but have a global impact.

Evolving a software architecture by modifying its description to accommodate change requests faces numerous research challenges. In particular, the evolution of an architectural description should typically preserve its purpose and criticality concerns. There are two ways to verify that such properties are preserved: by analyzing and verifying the resulting architectural description after the changes, or by analyzing the initial architectural description together with the "delta" or "increment" applied to it to make the changes.

The evolution of an architectural description should typically preserve its purpose and criticality concerns.

Current ADLs provide little support for architectural evolution, leaving it to processes, tools, and techniques outside the architecture description's concern.[2] Nevertheless, researchers agree that evolving the architecture description is beneficial, particularly in the case of critical systems, and in recent years have made promising gains.

MODEL-TRANSFORMATION-BASED EVOLUTION

The model-driven-engineering community uses models as artifacts to describe well-defined software aspects at a higher abstraction level than source code. *Model transformation* is a well-established technique to modify and evolve models.[3] Researchers have developed various model-transformation languages, some of which—such as ATL (ATLAS Transformation Language)—are seeing widespread industry adoption. Others are part of a standardization process, such as QVT (Query/View/Transformation), the de facto standard proposed by the Object Management Group to accompany UML (Unified Modeling Language). Because an architectural description can be seen as a software model, it makes sense to apply model-transformation approaches to architectural evolution.

Developers are applying the proven program-transformation technique of refactoring to models and specifications as well. Due to their semantic richness, models are often easier to evolve than programs. For almost any modeling language, various techniques exist to systematically modify the models to achieve certain effects. For example, composite structure diagrams can be transformed and refined[4] in a semantic-preserving way.

Many researchers have studied the formal foundations of model transformation. One well-known formalism used for this purpose is *graph transformation*, which enables reasoning about the formal properties of model transformations—in particular, how an architecture evolves. For example, this approach can be used to verify whether a given architectural transformation preserves certain structural, behavioral, or other properties. This is particularly useful in the context of architectural restructuring, which aims to improve the structure of an architectural description while improving its behavioral properties.

Using model transformation, and especially graph transformation, to express and formalize the evolution of architectural descriptions isn't new. Daniel Le Métayer[5] proposed such an approach more than a decade ago. More recently, Michael Wermelinger and José Luiz Fiadeiro[6] used graph transformation theory as a formal foundation for software architecture reconfiguration. Even more recently, Lars Grunske[7] formalized architectural refactorings as graph transformations that can be applied automatically. In a similar vein, Dalila Tamzalit and one of the authors[8] used graph transformations to express architectural evolution patterns as a means to introduce architectural styles as well as to verify whether a given architectural evolution preserves the constraints imposed by an architectural style. Automated support for this approach is currently under development using the COSA ADL and associated tools.

Another interesting approach to transformation-based architectural evolution, though not directly relying on graph transformation, is work by Olivier Barais and colleagues.[2] Their TranSAT framework supports architectural evolution based on ideas borrowed from aspect-oriented software development. The idea is to encapsulate new architectural concerns as architectural aspects and to use an architectural-transformation language to weave these aspects into the existing architecture description. This approach makes it possible to analyze transformations statically and incrementally to verify whether the resulting architecture description is structurally consistent—this saves considerable time and effort compared to doing a complete analysis of the resulting architecture description. Examples of such architectural restructuring include the transformation of a monolithic architecture into a distributed client-server architecture or into a three-tiered architecture that clearly separates the user interface, business logic, and data layer.

ARCHITECTURAL COEVOLUTION

While in many disciplines architectural descriptions are primarily concerned with structure, architectural descriptions of software serve as structural containers in which the complex behavior resides. From the end-user viewpoint, achieving correct and reliable behavior and functionality is the ultimate goal of a critical software-intensive system. The internal structure is only relevant to the software architects and developers who use it to master the software complexity. To reconcile both types of stakeholders, we need different views to represent the structural and behavioral descriptions of architecture.

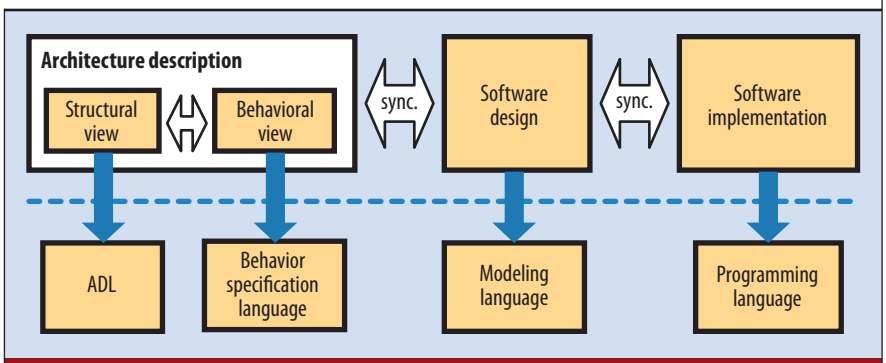

Figure 2. Coevolution of architectural viewpoints, design, and implementation. The architecture must be synchronized with other artifacts produced during software development.

Behavioral descriptions are often modeled in a precise formal form. Various modeling languages such as state-machine diagrams, sequence and activity diagrams, Petri nets, and temporal or other forms of logic are used to describe a system's behavioral aspects. All these behavioral languages either incorporate their own structural description or can be combined with a separate one expressed using some ADL or modeling language.

Evolving architectural descriptions inevitably requires the *coevolution* of different viewpoints: the structural viewpoint, the behavioral viewpoint, and often many other viewpoints as well. In addition, as Figure 2 shows, the architecture must be synchronized with other artifacts produced during software development such as system requirements, documentation, and, of course, implementation.

While most modeling languages have transformation techniques to evolve models in small, understandable steps, keeping models synchronized remains a challenge. Tool chains currently translate all models into a logic language and feed that into a verifier, but this clumsy technique fails to capture the modeling language's semantic richness and structure, and a modified model often can't be translated back into the original model.

Understanding how to transform structural descriptions and accompanying behavioral models in a synchronized, consistent way is critical to software development. Even more important is the coevolution of analysis or certification arguments, which can retain already validated properties if not affected directly. Proof-replay techniques for verifiers have had some success in this regard. However, researchers don't yet grasp how heterogeneous modeling languages semantically fit together or how to consistently coevolve them. This is especially true for structural ADLs and behavioral models. It's even unclear how state-, activity-, and flow-based models of the same architecture complement one another.

PRESERVING CRITICAL BEHAVIORAL PROPERTIES

It's essential to ensure that any evolutionary software adaptation retains desired properties that have been modeled, validated with stakeholders, or even formally proven correct versus requirements and implementation. This is even more important for critical systems, in which errors are often introduced during badly managed evolutionary steps. Making large architectural changes in one step is especially problematic. After such a "big bang," considerable validation and modification must occur to adapt behavioral models as well as any implementation. In contrast, a stepwise approach to evolution lets developers manage change more effectively through small, incremental transformations.

Transformations that refine or preserve behavior while adapting the architectural description to new requirements or technical needs are relatively complex, even in small evolutionary steps. Tools are therefore necessary to assist such transformations. Unfortunately, none of today's tools adequately preserve syntactical correctness and semantics. Further, researchers have mainly applied them to isolated modeling viewpoints and not to loosely coupled heterogeneous views, which are needed to describe an architecture's structure and behavior.

Transformation-based evolution of behavioral models is much harder to achieve than evolution of purely structural models. Tools usually carry out structural transformations rather efficiently. When behavior is involved, however, undecidability problems pop up such as semantic equivalence of logical preconditions. A simple solution to these problems would be to review them by hand; the most complex would be to feed them into an interactive verifier and enforce their formal correctness proof. This is why evolution techniques for behavior in

architecture descriptions will first arise only in certain kinds of critical systems.

A less expensive alternative is to use automated tests and invariants to iteratively check whether each evolution step is carried out correctly. However, this raises another problem: When evolving software architecture based on architectural descriptions, how do you keep the architecture consistent with the implementation?

One way to keep architectural artifacts consistent during evolution is to trace information-flow dependencies through them. Horizontal tracing aims to ensure consistency between architectural descriptions at the same stage of development, while vertical tracing aims to maintain consistency between the stages of development—for example, by aligning artifacts with code. Informal tracing is difficult because dependencies are easy to forget. Formal tracing techniques exist—for example, to formally check source-code annotations.[9] Explicitly adding evolution operators to the language helps to alleviate this problem, as the original information is still available and no trace is needed to recover dependencies. The optimal solution would be to generate parts of the code in such a form that it can be regenerated after each evolutionary step; automated tests could then regressively test system behavior.

It's impossible, whichever development process is adopted, to foresee all possible future requirements for evolving a system.

ARCHITECTURAL CHANGE AS A FIRST-CLASS CONSTRUCT

Current ADLs such as Koala don't directly address evolution, regarding it as extrinsic to architectural descriptions. The alternative is to provide first-class structural constructs to express and capture architectural change during both initial development and subsequent evolution. This necessitates dealing with unplanned modification, for it's impossible, whichever development process is adopted, to foresee all possible future requirements for evolving a system. While this approach may initially seem unusual, some programming languages already contain explicit constructs for system evolution. For example, subclassing could be interpreted as a form of evolution of classes where the "old" class taken from the library isn't evolved but adapted through the subclass only. However, subclassing permits only conservative extension—adding elements to but not removing them from a class.

Designer dilemma

Unplanned evolutionary change introduces a dilemma when designing built-in ADL language constructs to support change and extension. On one hand, constructs that always result in structurally well-formed and type-correct systems would inevitably permit only a subset of all possible valid system changes. On the other, constructs that result in invalid systems could only be permissible in an environment that comprehensively detects structural problems and type errors, especially with critical systems. There is thus a need to combine the freedom to perform incorrect changes with the ability to detect these errors to achieve sufficient expressiveness for unplanned changes. This comprehensive approach can accommodate destructive change—deleting elements from an architecture description—in addition to constructive change—adding elements to an architecture description.

When defining architectural changes as a first-class construct in an ADL, software architects should consider the different requirements of organizations responsible for system development, deployment, and modification. Consider, for example, a common scenario in the domain of enterprise resource planning software. A development organization produces a software framework product used by other organizations to build applications. To meet their local development requirements, these organizations may need to customize (modify and extend) the framework to support their applications. The original framework will evolve over time, so the organizations that use it must apply their local changes to the framework before using the evolved framework for their applications. In addition, a third party might wish to use applications from more than one framework customizer and thus needs to merge changes from both these organizations and the original framework provider.

Regarding an architecture description only as design documentation leads to the coevolution problem shown in Figure 2: keeping this documentation in synch with the software implementation as the system evolves. A model-driven-engineering approach ensures that an architecture definition isn't just a documentation artifact but a precise model for constructing both initial implementations and extensions to these implementations.

Example: Resemblance and replacement

Figure 3 illustrates two techniques, *resemblance* and *replacement*, that can be used to extend UML 2.x to permit the intrinsic definition of architectural evolution.[10]

Resemblance defines a new component as the difference in structure from one or more existing components. It's the delta—the set of additions, deletions, and replacements—of the components' elements applied to arrive at the new definition. Component elements include

- *parts*—instances of subcomponents,
- *ports*—instances of interfaces,
- *connectors*—bindings between ports, and
- *attributes*—component parameters.

Resemblance can also be applied to interfaces, in which case the modified elements are *operations*. If a resemblance delta consists only of additions, then when applied to an interface, it defines a proper subtype and thus can safely replace the original component.

Figure 3a depicts the architecture description of a simple database server that has two internal component parts: Database and FrontEnd. Figure 3b shows an evolution of this simple server that has been extended using resemblance to add managed access to the data stored in the server. ManagedServer resembles Server, and the text note defines the delta that results from editing Server to arrive at ManagedServer.

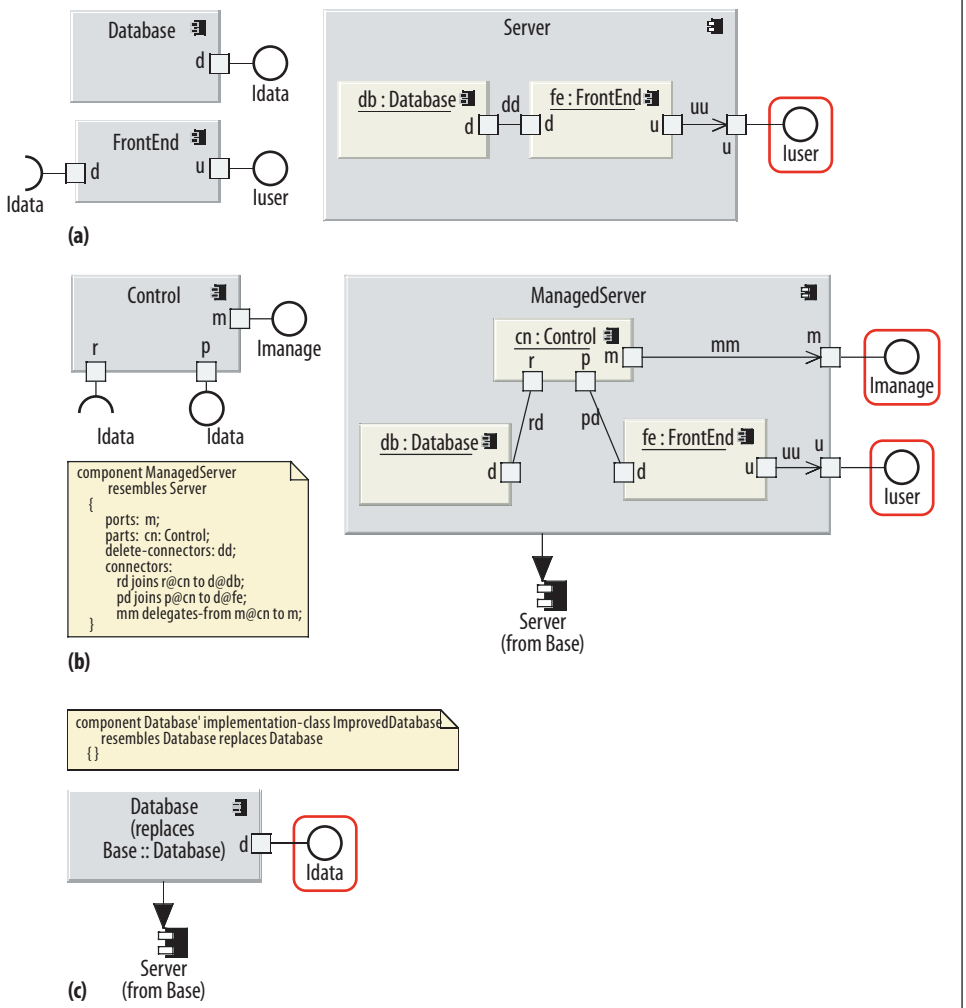

Figure 3. Evolving a software architecture description using Evolve, a UML 2.x evolution tool developed by Andrew McVeigh. (a) Architecture description of a simple database server. (b) Resemblance: architecture description of managed database server. (c) Replacement: replacing the Database component.

Resemblance's many-to-one relation permits the merging of multiple component definitions that may have arisen due to, for example, distributed development. Applying a sufficiently radical delta to a component may result in a new definition that bears little or no resemblance to the component definitions from which it's derived. Tracing evolutionary origins remains very important in many project contexts, as both engineering and nature provide many examples of systems that have dramatically evolved from their original form.

Replacement globally substitutes the definition of one component for another while preserving the original definition's identity, thereby maintaining any relations that a larger system has with this component. Combined with resemblance, replacement permits the incremental evolution of a component definition without having to change the composite definitions that use this component. Replacement is the key to managing change in composite hierarchical definitions because it enables substitution of definitions at one level of the hierarchy without necessarily affecting higher layers. For example, Figure 3c shows an improved implementation of the Database component that replaces the original Database when applied to the simple database server system (Figure 3a) or the managed server system (Figure 3b).

Resemblance allows elements to be deleted in forming a new definition from existing ones, but it isn't destructive editing in the traditional sense. Using resemblance to replace a definition in a base model with a new definition in an extension model doesn't remove the old definition; instead, it records the deletion in a delta. This approach enables history tracing, the use of base models instead of derivatives, and the resolution of conflicts when independently evolved extensions are subsequently merged.

Incremental change is integral to both the initial development and subsequent evolution of software-intensive critical systems. Making evolution intrinsic to architecture description is a principled and manageable way to deal with unplanned change. This intrinsic definition facilitates decentralized evolution of software by multiple independent developers. Unplanned extensions can be deployed to end users with the same facility that plug-in extensions are currently added to systems with planned extension points.

Acknowledgments

Tom Mens is supported by ARC project AUWB-08/12-UMH19, "Model-Driven Software Evolution," funded by the Ministère de la Communauté française—Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, and by the project TIC, cofunded by the European Regional Development Fund (ERDF) and the Walloon Region (Belgium).

References

1. *IEEE Std. 1471-2000* and *ISO/IEC 42010:2007, Recommended Practice for Architectural Description of Software-Intensive Systems*, 2007.
2. O. Barais et al., "Software Architecture Evolution," *Software Evolution*, T. Mens and S. Demeyer, eds., Springer, 2008, pp. 233-262.
3. S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, 2003, pp. 42-45.
4. J. Philipps and B. Rumpe, "Refinement of Pipe-and-Filter Architectures," *Proc. World Congress on Formal Methods in the Development of Computing Systems* (FM 99), LNCS 1708, Springer, 1999, pp. 96-115.
5. D. Le Métayer, "Describing Software Architecture Styles Using Graph Grammars," *IEEE Trans. Software Eng.*, vol. 24, no. 7, 1998, pp. 521-533.
6. M. Wermelinger and J.L. Fiadeiro, "A Graph Transformation Approach to Software Architecture Reconfiguration," *Science of Computer Programming*, vol. 44, no. 2, 2002, pp. 133-155.
7. L. Grunske, "Formalizing Architectural Refactorings as Graph Transformation Systems," *Proc. 6th Int'l Conf. Software Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing and 1st ACIS Int'l Workshop Self-Assembling Wireless Networks* (SNPD/SAWN 05), IEEE CS Press, 2005, pp. 324-329.
8. D. Tamzalit and T. Mens, "Guiding Architectural Restructuring through Architectural Styles," *Proc. 17th Ann. IEEE Int'l Conf. and Workshop Eng. of Computer-Based Systems* (ECBS 10), IEEE Press, 2010, pp. 69-78.
9. H. Krahn and B. Rumpe, "Towards Enabling Architectural Refactorings through Source Code Annotations," *Proc. der Modellierung 2006*, Gesellschaft für Informatik, 2006, pp. 203-212.
10. A. McVeigh, J. Kramer, and J. Magee, "Using Resemblance to Support Component Reuse and Evolution," *Proc. 2006 Conf. Specification and Verification of Component-Based Systems* (SAVCBS 06), ACM Press, 2006, pp. 49-56.

Tom Mens *is a professor and directs the Software Engineering Lab at the Institut d'Informatique, Faculty of Sciences, Université de Mons, Belgium. His research interests are in formal foundations and automated tool support for software evolution. Mens received a PhD in sciences from Vrije Universiteit Brussel, Belgium. He is a member of IEEE, the IEEE Computer Society, the ACM, the European Research Consortium for Informatics and Mathematics (ERCIM), and the European Association of Software Science and Technology (EASST). Contact him at tom.mens@umons.ac.be.*

Jeff Magee *is a professor, and heads the Department of Computing, at Imperial College London, UK. His research interests include software architecture, distributed systems, and mobile computing. Magee received a PhD in computer science from Imperial College London. He is a Chartered Fellow of the British Computer Society. Contact him at j.magee@imperial.ac.uk.*

Bernhard Rumpe *is a professor of software engineering in the Department of Computer Science at RWTH Aachen University, Germany. His research interests include modeling, software architecture, and evolution. Rumpe received an Habilitation in computer science from Munich University of Technology (TUM). He is a member of the IEEE Computer Society, the ACM, and Gesellschaft für Informatik (GI). Contact him at rumpe@se-rwth.de.*

Selected CS articles and columns are available for free at http://ComputingNow.computer.org.