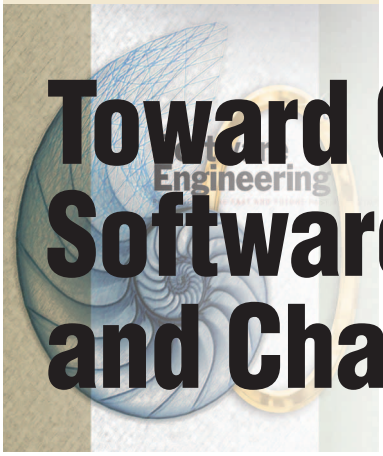

Toward Open-World Software: Issues and Challenges

Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi
Politecnico di Milano

Traditional software development is based on the closed-world assumption that the boundary between system and environment is known and unchanging. However, this assumption no longer works within today's unpredictable open-world settings, which demand techniques that let software react to changes by self-organizing its structure and self-adapting its behavior.

Developers currently handle environment changes by eliciting change requests, modifying the software design and implementation, verifying that the resulting product meets the changed requirements, and, finally, redeploying the application. This approach assumes that the external world changes slowly and that software can remain stable for long periods. It also assumes a closed external world, meaning that requirements leading to a specification for the software system's interaction with the external world can capture all phenomena of interest. The software itself is closed since it's composed of parts that don't change while it's executing.

However, these closed-world assumptions don't hold in an increasing number of cases, especially in ubiquitous and pervasive computing settings, where the world is intrinsically open. Applications cover a wide range of areas, from dynamic supply-chain management, dynamic enterprise federations, and virtual endeavors on the enterprise level, to automotive applications and home automation on the embedded-systems level. In an open world, the environment changes continuously. Software must adapt and react to changes dynamically, even if they're unanticipated.

Moreover, the world is open to new components that context changes could make dynamically available—for example, due to mobility. Systems can discover and bind such components dynamically to the application while it's executing. The software must therefore exhibit a self-organization capability. In other words, the traditional solution that software designers adopted—carefully elicit change requests, prioritize them, specify them, design changes, implement and test, then redeploy the software—doesn't work anymore.

To identify the exact nature of the new problems, we must first understand past achievements and how we can build upon them. Indeed, software's continuous evolution has delivered increasing flexibility. Under closed-world assumptions, software engineering has produced methods and techniques to deal with continuous change, without compromising applications' overall quality. For instance, software architectures have evolved from being static, monolithic, and centralized to dynamic, modular, and distributed. Driven by technical and economic forces, this change has occurred both at the process level—the way software is developed—and the product level—the way software is structured. However, we must go beyond these achievements to cope with the new open-world challenges.

HISTORICAL PERSPECTIVE

Inventors of early approaches to systematic software development tried to discipline the process by identifying well-defined stages and developing criteria for advancing from one stage to the next. In doing so, they hoped to avoid endless iterations of code-and-fix activities, reduce process costs, and improve predictability and product quality. They blamed poor software qual-

Published by the IEEE Computer Society

ity on continuous change—removing errors, meeting customers' expectations, and improving the implementation.

The waterfall model that Winston W. Royce proposed in 1970,[1] two years after software engineering was "officially" born,[2] represented an attempt to deliver that much-needed discipline and predictability. Royce thought that software developers should focus not only on coding but also on higher-level activities, such as requirements analysis and specification, software design, and verification and validation.

Others later argued that careful requirements analysis could improve product quality and avoid costly changes. They viewed software changes as harmful—a disruption to the disciplined development process and the ultimate cause for schedule slips and cost overruns.

These early approaches shared fundamental implicit assumptions about software and the world it would inhabit. They assumed a fixed (static) world, meaning that the requirements were supposed to exist, ready to be discovered and captured by carefully analyzing the world and eliciting its needs. Such needs were supposed to be highly stable, as were most organizations of that time. Under the recommended processes, engineers were guided to delay design and implementation until after they had exhaustively elicited and specified the requirements.

Furthermore, since organizations were mostly monolithic, solutions to their needs were accordingly monolithic and centralized. Improving the efficiency of organizations' internal activities took precedence over interactions among organizations, which were minimal and not perceived as critical.

Waterfall processes were indeed closed and monolithic, as was the structure of software products. Even though engineers developed complex applications modularly, all modules were statically bound to each other, and the resulting frozen application was statically deployed on a physically centralized architecture.

Since in most practical cases requirements can't be fully gathered upfront and frozen, the closed-world assumption soon proved unrealistic. Furthermore, the existence of many stakeholders results in conflicting and intrinsically decentralized requirements. Stakeholders often don't know beforehand what they expect from a system. Thus, it's an illusion to exhaustively gather requirements and preplan the process to avoid future changes: Change isn't a nuisance to avoid, but an intrinsic factor to address.

The history of software engineering shows a progressive departure from the strict boundaries of the closed-world assumption toward flexible support of continuous evolution at both the process and product levels. Researchers developed methods, techniques, and tools to support the need for change without compromising product quality and cost-efficiency. To reduce risks and better tailor solutions to user needs, they introduced evolutionary process models, such as incremental and prototyping-based approaches. More recently, these approaches evolved into agile methods, such as extreme programming.

Along the way, product architectures became modular and distributed, instead of static, centralized, and monolithic, where changes implied recompilation and redeployment of applications. Software engineering researchers discovered design principles and methods to support architecture changes, such as information hiding, encapsulation, and separation of a module's interface from its implementation.[3] To enforce good design practices, researchers eventually embedded these principles in new programming languages.

In particular, object-oriented programming languages, like C++ and Java, can safely accommodate certain anticipated changes in the implementation architecture. If changes to an existing module (a class) can be cast into the subclass mechanism, the changes can be made while the system is running. Thanks to polymorphism and dynamic binding,[4] a client's invocation of a certain operation might be bound dynamically to an operation redefined in a subclass without affecting the client.

As the "Polymorphism and Dynamic Binding" sidebar describes, these characteristics provide flexibility that can coexist with the discipline and safety that strong typing supports. In addition to letting bindings among mod-

Polymorphism and Dynamic Binding

Polymorphism and dynamic binding offer a first step to developing software that runs in an open world.

As a simple example, suppose that an application is initially designed to deal with a certain device, such as a fax machine. Through a variable *f* of class Fax, you can send a fax by writing f.sendFax(*t*, *n*), where *t* is a text and *n* is a fax number.

Suppose that you later add to the system a new device, such as a fax with phone, which provides its own way of sending faxes. If FaxWithPhone is a derived class of Fax, which redefines operation sendFax, variable *f* can refer to it, and the result of f.sendFax(*t*, *n*) would result in sending the fax using the redefined method of class FaxWithPhone. More important, the client component that uses variable *f* to send faxes isn't affected by the change.

The compiler checks for correctness by assuming that *f*'s type is defined by class Fax, but the invocation of sendFax is ensured to be correct even if the dynamic type of the object referred by *f* is FaxWithPhone. Indeed, the client continues to work correctly with the newly defined device as it did earlier.

This simple example shows that the flexibility that polymorphism and dynamic binding provide can coexist with the discipline and safety that strong typing supports.

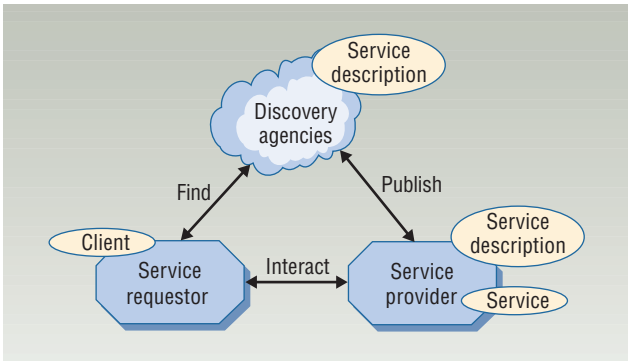

Figure 1. Service-oriented architecture. Components export the services they provide, and clients can discover the services that fit their quality requirements.

ules become dynamic, the software technology evolution lets them extend across network boundaries.

With Java, for example, developers can invoke a method on a remote server object running on a different network node. The server node might need to dynamically download the code that implements the methods used to manipulate object parameters, if their class is a new, dynamically defined subclass of the formal parameter's class.

Component-based software represents another major advance, moving software development processes to the open world. Third parties develop and provide components and are responsible for their quality and evolution. Application development thus becomes partly decentralized. At an extreme, application development consists of gluing components together by using middleware technology as the integration and coordination infrastructure.

In short, the evolution of software development methods and tools has made it possible to support seamless evolution of the product structure to incorporate certain anticipated changes, such as additional or redefined module functionalities, and to support decentralization and distribution. By carefully decoupling modules via well-defined interfaces, developers can now incorporate changes in applications without disrupting the overall correctness.

There is now unprecedented demand for software that continuously evolves for use in an open world. Existing approaches to software development can't cope with these new contexts. Consequently, we must explore new research directions.

TOWARD THE OPEN-WORLD ASSUMPTION

Learning how to operate under the open-world assumption becomes more crucial when we look at emerging domains like ambient intelligence, context-aware applications, and pervasive computing—all of which aim to integrate computation into the environment.

For example, the ambient intelligence concept that the European Union presents in its latest framework program envisions that, "People are surrounded by intelligent intuitive interfaces that are embedded in all kinds of objects. The ambient intelligence environment is capable of recognizing and responding to the presence of different individuals. Ambient intelligence works in a seamless, unobtrusive and often invisible way."[5]

Implementing this concept requires the underlying software system to dynamically change its behavior in response to changes in the environment it's interacting with or controlling. If, for example, a person wearing a device to control vital functions enters a room, the wearable device must start "interacting with the room" to acquire information about environmental conditions and communicate the person's needs (for example, darkened lights or particular meals at specific times).

Technically, the wearable device must discover and communicate with the possibly unknown components controlling the room. Those components, in turn, must be able to react to situations that were unforeseen when the system was developed.

In this setting, component heterogeneity is crucial, in terms of technical characteristics, size, and purpose. In fact, we could imagine a software system supporting the above scenario as composed of RFID tags that provide information about devices in the room, sensors that gather information about the environment, small devices with limited computational capabilities devoted to dedicated tasks, and full-fledged components—all connected through a mixture of ad hoc and traditional networks—to coordinate the whole system.

We view these as a special case of self-healing systems. According to David Garland and colleagues,[6] these systems "adapt themselves at runtime to handle such things as resource variability, changing user needs, and system faults. In the past, systems that supported self-repair were rare, confined mostly to domains like telecommunications switches or deep-space control software, where taking a system down for upgrades wasn't an option, and where human intervention wasn't always feasible. However, today, more and more systems have this requirement, including e-commerce systems and mobile embedded systems."

In this context, detecting critical situations and recovering to a stable configuration is important. In addition, the elements composing such systems usually require proper mechanisms for multicast communication and maintaining a global knowledge of the whole system state.

Researchers and practitioners are also increasingly interested in building applications by assembling existing services executed remotely at the provider site. Such compositions let them use what exists without concern about the computational resources needed to execute selected services. This requires the following:

- Developers and users must trust the services they use to compose the application. Each service should clearly describe its nonfunctional characteristics, as

well as its functionality, to let the client understand if the service fits its needs. Moreover, the client must be assured that the service meets its description's promises.
- Developers should define suitable mechanisms to set up and negotiate service-level agreements between clients and services.
- Developers should allow applications to set the bindings to specific services at runtime. This is necessary when the specific service depends on context information acquired at runtime (for example, the user's physical location) and when the user needs a replacement because the service fails or is unavailable. Dynamic binding also supports optimization of service compositions, since rebinding might occur as new services become available at runtime.
- Service providers should announce and discover new services at runtime to support dynamic binding. Furthermore, to cope with unforeseen syntactic differences, they should provide features to support the adaptation of the service interface the client expects to the one actually offered by the service.
- Because services might change unexpectedly and because of dynamic binding, their users (either humans or other applications/services) need to monitor real behaviors that might deviate from what's expected and plan for strategies to react to them.

The more we move toward dynamic and heterogeneous systems, and the more we stress their self-healing and self-adapting capabilities, the more we need new approaches to develop these applications and new ways to structure and program them. We must shift from the prevailing synchronous approach to distributed programming to a fundamentally more delay-tolerant and failure-resilient asynchronous programming approach.[7] Global behaviors emerge by asynchronous combinations of individual behaviors, and bindings and compositions change dynamically.

In such a dynamic setting, however, assuring the resulting application's desired quality is a difficult challenge.

Supporting Technology

The following are some existing standards, industrial products, and research prototypes that support, to a certain extent, the open-world assumptions.

Service-oriented technologies
- Jini: www.sun.com/software/jini/index.xml
- Open Services Gateway Initiative (OSGi): www.osgi.org
- SOAP: www.w3.org/TR/soap12-part1
- Web Service Description Language (WSDL): www.w3.org/TR/wsdl; also www.w3.org/TR/wsdl20
- Universal description, discovery, and integration (UDDI): www.uddi.org
- Business Process Execution Language (BPEL): www128.ibm.com/developerworks/library/specification/ws-bpel
- Web Service Security (WS-Security): www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
- Web Services Trust Language (WS-Trust): www-128.ibm.com/developerworks/library/specification/ws-trust

Publish/subscribe middleware systems
- Java Message Service (JMS): www.java.sun.com/products/jms
- Scalable Internet Event Notification Architectures (Siena): www.cs.colorado.edu/serl/dot/siena.html
- Java Event-Based Distributed Infrastructure (JEDI): G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Trans. Software Eng.*, vol. 27, no. 9, 2001, pp. 827-858.
- TIBCO Software solutions: www.tibco.com

Grid infrastructures
- Open Grid Services Architecture (OGSA): www.globus.org/ogsa
- gLite (lightweight middleware): http://glite.web.cern.ch/glite
- Globus toolkit: www.globus.org/toolkit

Autonomic frameworks
- The Anthill Project: www.cs.unibo.it/projects/anthill
- Recovery-Oriented Computing (ROC): roc.cs.berkeley.edu
- Autonomic Replication Management Service (ARMS): www.almaden.ibm.com/software/projects/arms
- Project AutoMate: http://automate.rutgers.edu

EXISTING SOLUTIONS

Promising new approaches and technologies including *service-oriented architectures* are now emerging that partially support these application domains.

As Figure 1 shows, in service-oriented architectures, components export the services they provide and clients can discover the services that fit their quality requirements. Once service requests match provisions, point-to-point interactions occur between clients and service providers.

In support of the matching process, a service-oriented architecture places service brokers between clients and providers to collect and advertise available services and facilitate their interaction. Services can thus be selected based on their functional capabilities and their promised quality of service (QoS) or nonfunctional features.

The "Supporting Technology" sidebar lists existing standards, industrial products, and research prototypes that support open-world assumptions.

Jini, the Open Services Gateway Initiative, and Web services are the most widely known implementations of these concepts. The first two approaches impose technology-specific programming environments and focus more on supporting interaction among services in a local area network.

Web services

At least in principle, Web services require neither particular programming languages nor dedicated supporting platforms. Developers encode interfaces and data in XML and exchange them through SOAP. Even if many XML-based proposals can tackle the different aspects of Web services, the available technology is based on the Web Service Description Language (WSDL) for service interface description, and UDDI for service discovery.

Web services, at least in principle, require neither particular programming languages nor dedicated supporting platforms.

Web services let designers integrate and remotely use services that different providers supply. The Web services can also be composed together to form more complex services. Designers typically use the Business Process Execution Language for this. BPEL imposes a workflow-based coordination of involved services and requires the identification of at least the structure of the WSDL interface of all parties at design time. In principle, designers can discover new services at different times.

This affects the degree of dynamism and flexibility embedded in these applications. If designers perform discovery at design time, they select the services and hard-code their addresses into the BPEL workflow. If they perform discovery at deployment time, they use a service broker to "configure" the application. In either case, the set of bound services doesn't change dynamically, and the binding between service requests and actual services is established once for all.

Discovery and selection can also occur at runtime. This enables a context-aware behavior of service compositions that can yield beneficial effects: The service broker can discover different sets of services, and the composition can dynamically select one of them. However, the current technology doesn't support expression, at design time, of the requirements and constraints to be fulfilled at runtime in the discovery and selection phase to identify the services to be bound. This drawback makes development of systems exploiting this runtime binding capability almost impossible in practice.

Publish/subscribe middleware

Researchers have proposed the *publish/subscribe paradigm* as a basis for middleware platforms that support software applications composed of highly evolvable and dynamic federations of components. According to this paradigm, components don't interact directly with each other. Instead, an additional "dispatcher" logical layer mediates their communications. Components declare the events they're interested in, and when a component publishes an event, the dispatcher notifies all components that have subscribed.

Publish/subscribe middleware decouples the communication among components and supports implicit bindings among components. The sender doesn't know the identity of the receivers of its messages, but the middleware identifies them dynamically. Consequently, new components can dynamically join the federation, become immediately active, and cooperate with the other components without requiring any reconfiguration of the architecture.

Available commercial products and advanced research prototypes support the publish/subscribe paradigm. They might differ in several aspects, such as the ability to cope with mobile environments or the policies adopted for event delivery. For example, event notification might or might not preserve the same order in which events were published.

Grid computing

Available technological solutions also support *grid computing*. They provide facilities to enable the sharing, selection, and aggregation of resources distributed across multiple administrative domains based on the resources' availability, capacity, performance, cost, and QoS requirements. In a grid, resources are virtualized and independent of the actual supplier and physical location.

Standardization helps diverse and heterogeneous resources make up a modern computing environment that designers can discover, access, allocate, monitor, and, in general, manage as a single virtual system—even if different vendors provide the single components and different organizations operate them. The Open Grid Services Architecture proposed by the Global Grid Forum (which recently merged with the Enterprise Grid Alliance) demonstrates the convergence of the different initiatives and technological solutions into an integrated and loosely coupled framework. Interestingly, this framework implements resources as Web services offering proper WSDL interfaces.

Autonomic computing

Finally, emerging *autonomic computing environments* offer features for self-configuration, self-healing, self-optimization, and self-protection to cope with the complexity, heterogeneity, and uncertainty of modern software systems.[8] In this context, among the other aspects, researchers are studying new interaction and cooperation paradigms inspired by biological systems. In particular, they're modeling the abilities of single

biological entities (for example, ants) to self-organize and behave autonomously and together achieve a common objective. Furthermore, they're coding these models into software systems. While we find this approach interesting and challenging, researchers need to thoroughly assess these behavioral models' advantages and applicability to specific autonomic problems.[9]

RESEARCH AGENDA

Software engineering research and practice has been moving to progressively remove the limitations of closed-world assumptions. Satisfying the goals of open-world software, however, requires defining a research agenda that builds on past achievements and identifies the new challenges.

Specification

Software engineering research has always been concerned with software specification. Researchers have made much progress with components' interface specifications, covering both syntactic aspects and functional semantics.

Providing formal specification of software components is mandatory today if components are to provide services that others can depend upon, retrieve, and use. Although component specifications have traditionally focused on functionality, in the open-world context they must specify other aspects, such as usage protocol, transactional properties, and nonfunctional properties.

The Semantic Web community, which uses ontologies to provide formal semantics to service descriptions, offers a first step in this direction. In the Web Service Modeling Ontology,[10] the ontology-based information included in a service description concerns a service's behavior, the way it interacts with its counterparts, and its QoS characteristics. We need more research in this direction, however. Researchers should explore how service specifications can support service classification and, hence, service retrieval. In particular, a notion of substitutability—to define when a given service might substitute another—seems important to support dynamic reconfigurations.

Verification

The services that providers make available should meet their specification. Service consumers, in fact, select services based on their specification and then rely on them to perform some relevant business functions or, in turn, to provide higher-level composed services.

Service producers' offline verification of services isn't enough in this framework.[11] Services need to be verified when they're published and even during runtime, since they might be changed without explicit notice.

Verification in the publishing phase might be done as part of an admission protocol, whereby the service undergoes a certification stage. Runtime verification protects users from unanticipated unacceptable changes.

New methods must be developed to support this new kind of verification. As an example, consider the idea of continuous testing. Antonia Bertolino and colleagues describe initial steps in this direction.[12]

Monitoring

A highly dynamic and open system necessitates runtime monitoring to watch for situations that might require suitable reactions to assure the desired level of global quality.

Software must be able to evolve dynamically in an open world.

Monitoring amounts to inserting probes in the system, collecting and analyzing data, and then reacting according to the results. Analysis compares the observations with the specifications. The problem here is to identify, select, and tune monitoring strategies, which can have various degrees of invasiveness.

Trust

In the open world, parties providing and consuming a service are easily exposed to cheaters. Monitoring can recognize such situations and trigger some recovery action. However, it should be possible to prevent the occurrence of situations that can harm the system.

Within the service-oriented domain, initiatives such as Web Services-Security, Web Services-Policy, and Web Services-Trust define protocols that allow some authority to guarantee other parties' trustworthiness. Others adopt a more "democratic" approach and base trust on the parties' reputation for interacting with others in the system. Clearly, neither approach is resilient to attacks, as discussed at O'Reilly's XML site (www.xml.com/pub/a/ws/2004/04/14/p2pws2.html?page=1).

The approaches must be clearly complemented with verification and monitoring, but in most cases they seem to work under the assumption that the majority of people—and the software components they build—are usually honest.

Implementation

Programming open systems requires new programming language features. Two features that bear investigation are introspection mechanisms to get runtime information about newly encountered services and reflective mechanisms to adapt client applications dynamically.

Self-management

Software must be able to evolve dynamically in an open world. Although developers can't foresee all changes, they must define the boundaries the system can evolve within,

Supporting Self-Reconfiguration

As part of the service-centric system engineering project (http://secse.eng.it), we are developing an approach to self-healing Web services. This approach is based on ideas coming from design-by-contract and assertions in the Eiffel programming language, which let the user set constraints on program execution and also identify possible reactions if they're not satisfied.[1] In our work, we propose monitoring rules and reaction rules to oversee the execution of Business Process Execution Language workflows.

The explicit and external definition of monitoring rules and reaction rules maintains a good separation between business and control logic, where the former is the BPEL process that implements the business process and the latter is the set of rules defined to control and modify the execution, if needed.

Separating these concerns lets designers produce BPEL specifications that address only the problem they need to solve, without intertwining the solution with awkward pieces of defensive programming. Different rules can be associated with the same BPEL workflow to tailor the degree of control to the specific execution context without reworking the business process. Moreover, a good separation of concerns allows precise management of monitoring and reaction rules, and it's an effective way to find the right balance between self-healing and performance.

Figure A summarizes our approach. More specifically:

- Designers may conceive rules either in parallel with the business process or just after it's designed. Monitoring rules are associated with specific elements of the business process—for example, invocations of external services. Reaction rules can be process-wide or associated with specific process elements.
- Users can select the rules to use with a specific execution anytime before the process is executed. BPEL2 instruments the original BPEL specification to allow execution of the proper rules. This instrumentation feeds the monitoring manager, a proxy service that is responsible for understanding whether a monitoring rule must be evaluated, and, if that's the case, for reacting as the reaction rules state.
- When the instrumented BPEL process starts its execution, it calls the monitoring manager whenever a monitoring rule must be considered. The actual evaluation depends on the values of the parameters associated with the rule and on the context in which it's executed. For example, one of these parameters is priority, and the context comprises the global priority set for the execution of the workflow at startup time. A rule with priority lower than the global one would be skipped, and the monitoring manager would call the actual service directly.
- A special-purpose interface interacts with the monitoring manager and changes its status. This happens when the designer wants to change the impact of monitoring at runtime without redeploying the entire process.
- If some constraints aren't met, the monitoring manager activates the reaction rules to try to keep the execution alive. Reaction rules can trigger and control various actions ranging from dynamic binding to negotiation of some service level agreement to a partial replanning of the process itself.

Reference

1. L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes," *Proc. Int'l Conf. Service-Oriented Computing* (ICSOC 05), 2005, pp. 269-282.

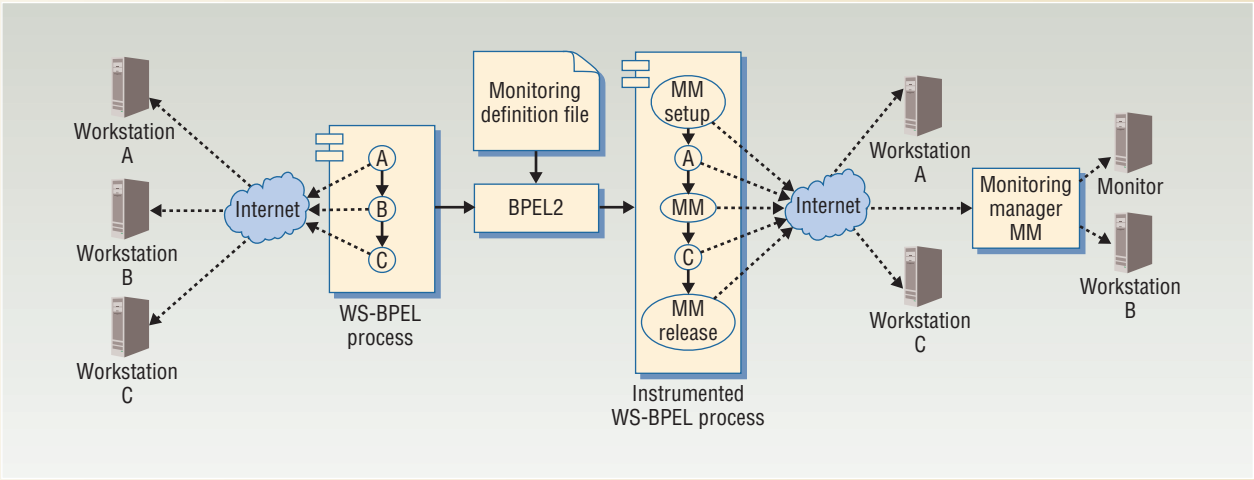

Figure A. BPEL workflow. Different rules can be associated with the same BPEL workflow to tailor the degree of control to the specific execution context without reworking the business process.

and the criteria it will consider in deciding how to evolve. Without defining these, self-managing systems would soon go out of control. As a result of monitoring, it should be possible to handle deviations from the expected behaviors and plan for a reconfiguration.

We investigated an approach to supporting self-healing Web services that relies on workflows that can detect and react to services' deviations from expected behaviors. The "Supporting Self-Reconfiguration" sidebar provides an outline of this work.

In the same direction, some autonomic computing platforms provide programming support to define autonomic rules that each component might execute to achieve an overall system change. As mentioned, some of these rules might be built in a way that resembles the behavior of existing biological systems. The research community is working on self-managing software, with significant results expected in the next few years.

The need for software that can continuously evolve in an open world is reaching unprecedented levels. Existing approaches to software development can't cope with these new challenges. Consequently, we must explore new research directions. The more we move toward dynamic and heterogeneous systems, and the more we stress their self-healing and self-adapting capabilities, the more we need new approaches to develop these applications and new ways to structure and program them.

References

1. W.W. Royce, "Managing the Development of Large Software Systems," *Proc. IEEE WESCON*, IEEE Press, 1970, pp. 1-9.
2. P. Naur and B. Randell, eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 1968.
3. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, Dec. 1972, pp. 1053-1058.
4. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.
5. K. Ducatel et al., *Scenarios for Ambient Intelligence in 2010*, tech. report, Inst. of Prospective Technological Studies-Seville, 2001.
6. D. Garland, J. Kramer, and A. Wolf, eds., *Proc. First Workshop Self-Healing Systems* (WOSS 02), ACM Press, 2002.
7. 2020 Science Group, *Toward 2020 Science*, tech. report, Microsoft, 2006; http://research.microsoft.com/towards2020science/downloads/T2020S_Report.pdf.
8. J.O. Kephart, "Research Challenges of Autonomic Computing," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 15-22.
9. G. Serugendo et al., "Self-Organisation: Paradigms and Applications," *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, Springer, 2004, pp. 1-19.
10. D. Roman, L. Holger, and U. Keller, eds., *D2v1.2. Web Service Modeling Ontology*, WSMO Working Group, 2005; www.wsmo.org/TR/d2/v1.2.
11. G. Canfora and M. Di Penta, "Testing Services and Service-Centric Systems: Challenges and Opportunities," *IT Professional*, Mar./Apr. 2006, pp. 10-17.
12. A. Bertolino et al., "Audition of Web Services for Testing Conformance to Open Specified Protocols," J. Stafford et al., eds., *Architecting Systems with Trustworthy Components*, Springer, 2006.

Luciano Baresi *is an associate professor of computer science at Politecnico di Milano. His research interests focus on dynamic software architectures, with special emphasis on service-oriented applications. He received a PhD in computer science from Politecnico di Milano. Contact him at baresi@elet.polimi.it.*

Elisabetta Di Nitto *is an associate professor at Politecnico di Milano. Her research interests are software architectures and middleware, process support systems, and Internet applications. She received a PhD in computer science and automation from Politecnico di Milano. Contact her at dinitto@elet.polimi.it.*

Carlo Ghezzi *is a professor of software engineering at Politecnico di Milano. His research interests are software engineering and programming languages. He is an IEEE Fellow and an ACM Fellow. Contact him at carlo.ghezzi@elet.polimi.it.*