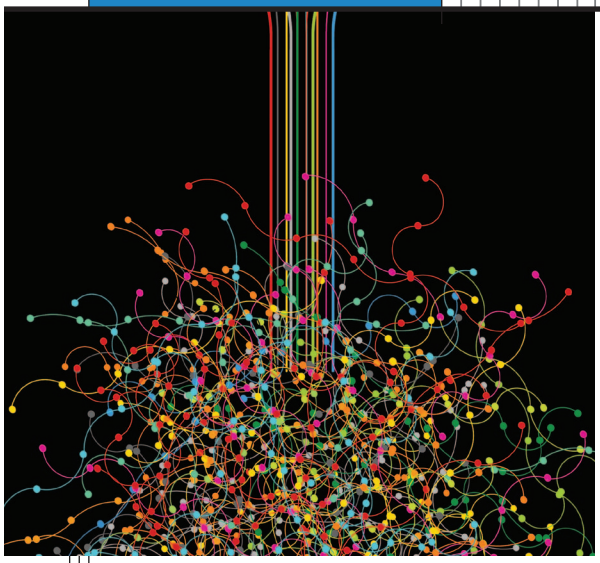

Service-Oriented Dynamic Software Product Lines

Luciano Baresi and Sam Guinea, *Politecnico di Milano, Italy*
Liliana Pasquale, *Lero—the Irish Software Engineering Research Centre*

An operational example of controls in a smart home demonstrates the potential of a solution that combines the Common Variability Language and a dynamic extension of the Business Process Execution Language to address the need to manage software system variability at runtime.

As software systems become increasingly dynamic and complex, configuration management must keep pace with a host of changing requirements and context-awareness demands. The cost of explicitly identifying and managing all possible feature configurations in such dynamic systems can easily become prohibitive, and designers cannot always pinpoint alternatives early on. Consequently, systems must be able to react to changes and adjust their behavior as they execute: in short, runtime adaptation is rapidly becoming essential to software system design.

Developers conceived software product lines to distinguish the various features of a system family and organize those features into meaningful configurations. Designers adopt a feature model to identify the software product line's common and variant features. This model describes feature constraints, such as if features require or exclude one another. Because many modern systems elicit actual requirements only at runtime, they select, deploy, and act on their features only while they are executing. This idea is central to the advent of dynamic software product lines (DSPLs), which extend software product lines to support late variability.

If the need for a feature materializes at runtime, the software system can use a DSPL to cope with that change by switching from one variant to another while executing. The DSPL feature model ensures that the system moves from one consistent configuration to another and that it satisfies the feature constraints. However, switching from one variant to another might not be enough. Functionality might need to vary to cope with unforeseen situations, and the feature model itself might need alteration to accommodate new variability; thus, the model itself must become a live runtime entity.

As the "Why Service-Oriented Architecture?" sidebar describes, SOAs have proven cost-effective in developing flexible and dynamic software systems,[1] and we believe there are significant mutual advantages in converging SOAs and DSPLs. The loose coupling in SOAs can provide DSPLs with the technical underpinnings of flexible feature management—underpinnings that are grounded in the considerable work to develop self-adaptive SOA systems[2,3] and in extensive studies of monitoring and adaptation techniques. DSPLs, in turn, can provide the modeling framework to understand a self-adaptive SOA-based system by highlighting the relationships among its parts. Software system architects can use these models to understand the implications of modifying a system's configuration at runtime.

To encourage this convergence, we are experimenting with how to enrich Business Process Execution Language (BPEL) compositions with dynamic variability management. Our solution is to use the Common Variability Language (CVL) to augment BPEL processes with variability, which makes it possible to easily generate a DSPL and use a dynamic version of BPEL to manage and run it. To

 Published by the IEEE Computer Society

WHY SERVICE-ORIENTED ARCHITECTURE?

In its simplest form, a service-oriented architecture (SOA) is a pattern that helps designers understand what qualities would be beneficial in a complex system.[1] SOA-based systems build on the notion of services—loosely coupled, self-describing, coarse-grained components accessed using well-defined standards, such as the Web Service Description Language and the Simple Object Access Protocol.

Services simplify the integration of complex systems and allow them to more flexibly accommodate change. Integration is simpler because there is no need to download and deploy the desired services: users merely access them remotely in a standard way. Loose coupling and the use of dynamic- or late-binding techniques make the most appropriate services available at runtime during any given situation.

The main SOA system development task is composition. A BPEL process establishes the order of message exchanges between a centralized entity, or BPEL engine, and its external partner services. An extended BPEL engine also executes business-oriented models after the proper automatic transformations.

Although initial acceptance has been high, time has shown BPEL to be fundamentally flawed in its ability to support change. Neither BPEL nor its execution engines support the modification of a process model or its instances after deployment.[2] Partner services can be substituted at runtime, but such a task becomes impractical or too complex when the change space involves thousands of services or when some services are unknown at design time.

In addition, BPEL does not support more complex changes, such as the substitution of entire process fragments. Consequently, although SOA-based systems have flourished, BPEL and the other well-known technologies can still only partially support changes.

References

1. M. Papazoglou, *Web Services: Principles and Technology*, Pearson-Prentice Hall, 2007.
2. F. Curbera et al., "Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, 2006, pp. 1219-1228.

validate our approach, we explored how to use it to automate the domestic systems, such as heating and lighting, in a smart home.

DYNAMIC BPEL

Our proposed solution is based on Dynamic BPEL (DyBPEL),[2] a tool that complements the widely used ActiveBPEL execution engine (www.activebpel.org) with runtime adaptation capabilities. DyBPEL exploits aspect-oriented programming[4] to dynamically change the features bound to variation points as well as to alter the variation points themselves. Replacing a feature does not simply mean substituting a partner service. A feature can be a complex fragment of BPEL code, and as such, it can aggregate—and interact with—remote services and access the process's internal variables. Thus, BPEL processes can cope with unexpected changes in requirements, resource availability, and execution environment.

Architectural overview

Figure 1 shows DyBPEL's architecture. The *Variability Designer*, which is based on the Eclipse CVL plug-in, provides a new configuration along with a *change request* to the coordinator, which oversees the migration of running instances and process definitions from one configuration (or version) to another. The *runtime modifier* is in charge of migrating the running process instances and embeds ActiveBPEL, which executes BPEL specifications. The *BPEL modifier* oversees the changing of process definitions and thus of all future process instances.

The *repository* contains the data required for other components to operate correctly and tracks the processes that the system manages, as well as the versions associated

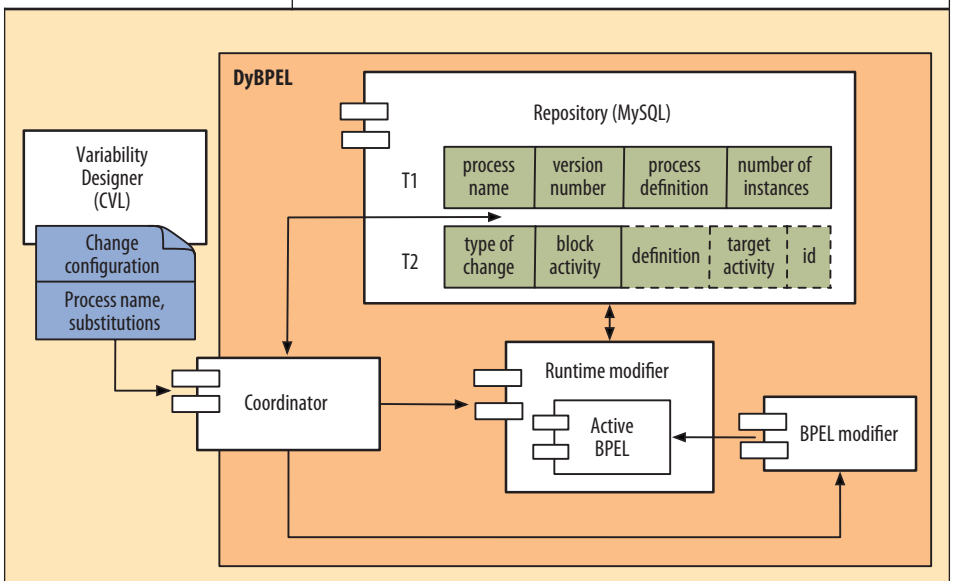

Figure 1. DyBPEL architecture. DyBPEL extends ActiveBPEL so that the service composition (process) can adapt to changes during its operation. From the Variability Designer, the coordinator receives a change request with the name of the process to be migrated and the set of substitutions to apply, which it then dispatches to the runtime and BPEL modifiers. The repository stores these changes and the data that other elements need to operate correctly.

with each process and the number of running instances that comply with each version (T1 and T2 in Figure 1). It also stores the changes that need to be applied to the executing instances when they need to be migrated to a new version (T2).

Managing change

Each change must define its type (such as add or remove), the point in the process in which the system should temporarily block execution to apply the change (block activity), and what will be added (definition) or removed (id). Elements to add or remove include single primitive/structured activities (for example, a complete sequence or flow), variables, and partner links. If a modification adds an element, there must also be a target activity specification that identifies where in the process the new elements are to go.

Block activities must not interrupt any conversation with partner services and cannot occur unless all internal state variables are in a consistent state.

These changes could introduce process inconsistencies, but DyBPEL is concerned only with providing the right means to enact the changes. State-of-the-art tools[5] are available to analyze the feature model and thus the soundness of foreseen changes.

Block activities must not interrupt any conversation with partner services and cannot occur unless all internal state variables are in a consistent state. Consequently, the system cannot perform changes while it is executing activities that are part of transactions. For example, the system can add or remove partner links only if it has not yet activated the scopes that encapsulate them. Also, the block activity should be immediately before the target activity and not contained in any transactional scope. Once the system has executed the block activity, it can no longer apply the change, and the process instance continues its execution untouched.

The runtime modifier intercepts the execution of the running process instances and applies the changes stored in the repository. To empower this component, we extend ActiveBPEL through aspect-oriented programming[4] and AspectJ (www.eclipse.org/aspectj). At the end of each activity, an aspect intercepts process execution, and the advice associated with the aspect checks if the repository contains changes that need to be applied at that point. If so, the runtime modifier operates the changes directly on the internal object representing that particular process instance (ActiveBPEL creates these objects). Another aspect intercepts the start and end of each process instance to update the repository.

The BPEL modifier handles the migration of process definitions to new configurations. It retrieves the latest process version from the repository and modifies it according to the substitutions defined in the coordinator's configuration change request.

Essentially, the BPEL modifier creates a new end point to distinguish the new process version from the previous ones and stores the new end point's definition in the ActiveBPEL's deployment descriptor file to redirect new requests to the latest process version. The repository also stores a record identifying the latest process version.

VARIABILITY DESIGN

To select a particular product or configuration and communicate to DyBPEL, we created Variability Designer, a tool that uses parts of CVL to represent the admissible variants characterizing a product line. A simple system to automate appliance control in a smart home illustrates the interplay between CVL and DyBPEL. Figure 2 shows the system's product variants and the configurations it can generate.

Base model

CVL choices provide a user-centric description of variability, representing both mandatory and optional features (variants). We use feature models to represent CVL choices since they can express a rich set of relationships among features. A feature model has a tree structure, and a group of child features can be either alternative or mandatory. A feature can also require another feature, or two features can exclude one other.

The system turns heating on (On) when the temperature is too low and switches it off (Off) when it becomes too hot. It also reduces the target temperature (Set temperature) if the energy consumption is peaking. The system switches the lights on when a person enters the room (On if present), and turns them off when no one is there (Off if not present). It can also trigger the alarm as soon as it detects motion from an intruder (Trigger alarm if present), but it excludes the option of turning lights on.

We use a base model as a starting point to define variability, which in this case is a BPEL process. The base model includes the process variables, activities, and partner links necessary to support the system's mandatory features—the partner services that monitor temperature (pt), control energy consumption (pe), and manage the heater (ph). When the temperature is too low, the process receives a lowTemp message from the pt service, and consequently invokes the ph service to switch the heater on. As soon as the system detects an energy peak, the process receives the energyPeak message from the pe service. If the energy peak is tolerable, the process invokes the pt service to reduce the target temperature

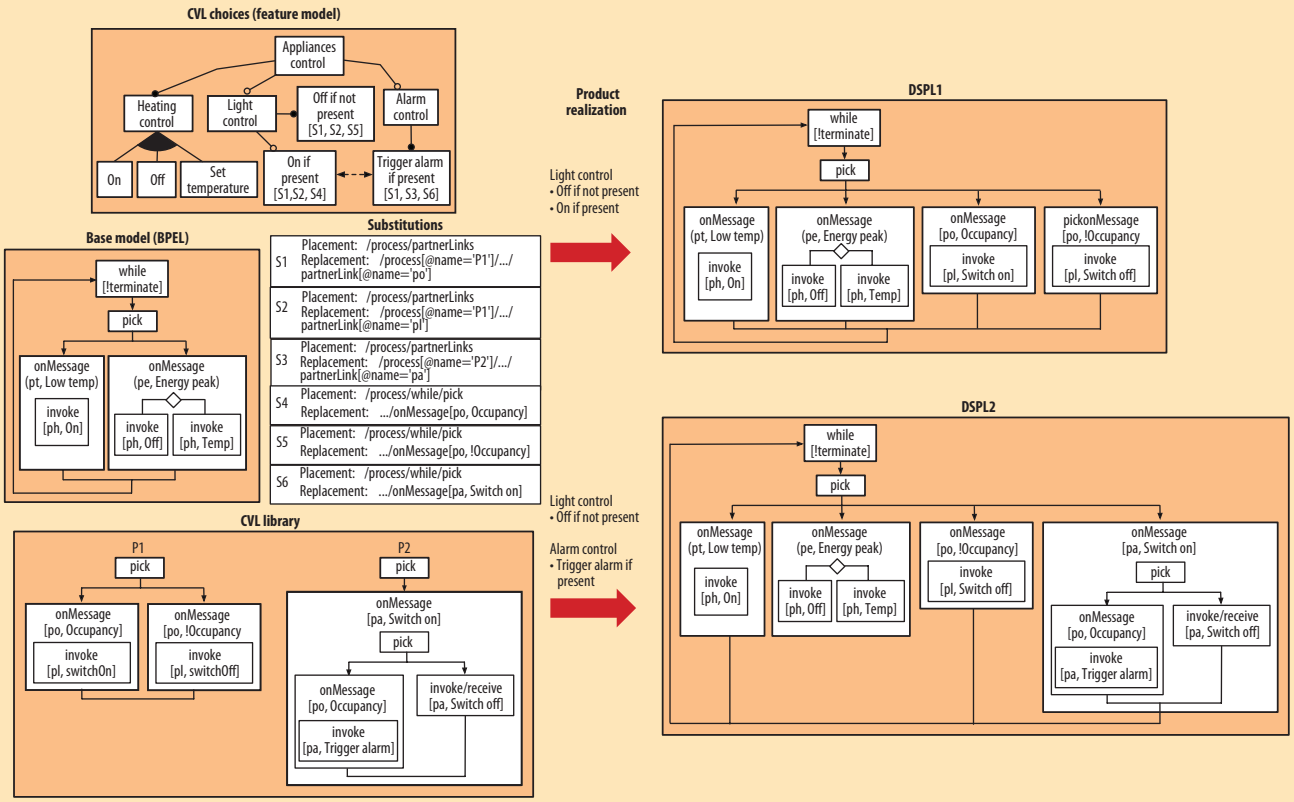

Figure 2. Automating domestic appliances in a smart home. In this CVL-based feature model, the feature variants (left) generate two DSPL configurations (right). Heating control is mandatory (dark dot); light and alarm controls are optional. The top configuration involves only light; the bottom configuration accommodates both light and alarm controls.

to temp; if it is not, the process invokes the ph service to switch the heater off.

Expressing variants

To define variants, we use a CVL library to represent additional process elements—from variables and partner links to structured activities. In our example, the library includes P1 and P2, process fragments that supply the features to implement light and alarm controls. P1 includes partner services for controlling lights (pl) and for managing the presence sensor (po). When the po service detects a person, it sends the process an Occupancy message. Subsequently, the process invokes the pl service to switch the light on. When po does not detect anyone for a certain period, it sends the process an !Occupancy message, and the process invokes pl to switch the light off.

P2 also includes po but adds a partner service to control the alarm (pa). Each time the homeowner turns on the alarm, the process receives a switchOn message. It then waits for one of two messages: a message from po that reveals the presence of an intruder, which signals it to trigger the alarm, or a message from pa that the system has switched off the alarm (switchOff).

Every model variant is expressed in terms of *substitutions* that it needs to incorporate into the base model to include that variant. A substitution comprises a *placement* and a *replacement*. A placement refers to an element in the process definition, and it indicates where in the process to add the replacement element, such as a variable or partner link or an activity. Table 1 shows the substitutions and their roles for the smart home example.

If a substitution has no replacement, the system removes the element that the placement identifies. For example,

Table 1. Substitutions in the smart home example.

Substitution	Role
S1	Adds service po to the partner links managed by the process
S2	Adds service pl to the partner links managed by the process
S3	Adds service pa to the partner links managed by the process
S4	Adds onMessage[po, Occupancy] to the activities managed by the pick
S5	Adds onMessage[po, !Occupancy] to the activities managed by the pick
S6	Adds onMessage [pa, Switch on] to the activities managed by the pick

Table 2. Time to migrate process definitions.

Specification	Average time (s)	Median (s)	Variance
Coordinator	0.1515	0.141	0.0035
BPEL modifier	0.1302	0.115	0.0043

Table 3. Time to migrate running process instances.

Process instance	Average time (s)	Median (s)	Variance
Retrieval	0.0463	0.045	0.00002
Change	0.0198	0.019	0.00001
Pick	0.2472	0.2485	0.0001

to apply "Off if not present," the system must perform substitutions S1, S2, and S5. S1 and S2 add the pl and po services, respectively. In both cases, placement points to the base model's partnerLinks element; replacement refers to the partnerLink elements in P1 that are associated with pl and po. S5 adds the need to insert the onMessage[po, !Occupancy] activity to the onMessage activities the pick activity manages.

Through CVL's product realization function, product configuration can include the process of selecting variants. An initial configuration includes only mandatory features, and it is impossible to generate a new product if selected variants are in conflict. After the designer identifies an admissible set of process variants, the system sends the information about the needed substitutions to DyBPEL. For example, if the designer selects the light control feature and all its subfeatures, the system sends substitutions S1, S2, S4, and S5 to DyBPEL.

The coordinator starts the change process by creating a new record in the repository for each partner link to be added. In this case, the block activity precedes the one containing the first use of the pl and po services. The replacements defined in S1 and S2 provide the definitions of these partner links.

The coordinator then creates a new record for adding the activities referred to in the replacements of S4 and S5. The block activity is one of the last activities of the onMessage sequences that the pick activity manages (invoke[ph, On], invoke[ph, Off], and invoke[ph, temp]). The S4 and S5 replacements refer to the necessary additional activities. The S4 and S5 placements identify the target activity. Finally, the coordinator also generates, deploys, and stores a new process definition in the repository so that future process instances can comply with it.

Another product configuration might add the "Start alarm if present" feature, which sets up a conflict with "On if present." If the system removes the latter feature to resolve the conflict, it must reverse the substitutions made to incorporate that feature in the first place. Thus, if a substitution adds a set of activities, variables, and partner links, the system must remove what was added. Likewise, if a substitution removes elements, reverting to the configuration before that substitution requires adding those elements back in.

When the homeowner selects new features, the Variability Designer sends the corresponding substitutions to the coordinator. The removal of the "On if present" feature causes S4 to be reverted. S1 and S2 are not reverted, since they are still necessary to support the "Off if not present" feature. To revert S4, the system removes the onMessage[po, Occupancy] activity and all its subactivities. The coordinator creates a record in the repository that indicates that removal: the id is the identifier of the activity to be removed and the block activity is the same one used in the previous configuration to apply S4. Applying substitutions S3 and S6 adds the "Start Alarm if present" feature. The coordinator creates a new record in the repository to add the pa service and onMessage[pa, Switch on] activity.

EVALUATING MIGRATION TIME

DyBPEL executes process specifications and migrates them from one configuration to another at runtime. Using aspect-oriented programming to define the entire DSPL can cause significant problems.[6] Typically, a feature has a higher abstraction level than an aspect; indeed, many aspects can represent a single feature. Thus, defining variability through aspect-oriented programming could become unmanageable and significantly degrade performance during production execution.

To avoid this problem, we rely on CVL to separate feature definition from the runtime mechanism that creates and executes the product line.

Table 2 illustrates the time the coordinator and the BPEL modifier take to migrate the process definition. Migration overhead stems primarily from the BPEL modifier, which must deploy the new process version and create a corresponding entry in the repository. This overhead is independent of the change management process; it is due mainly to the way ActiveBPEL works.

Table 3 gives the time to migrate running process instances. Retrieval is the time to retrieve a modification from the repository; change is the time to apply the changes. In our example, the time to dynamically modify the process is approximately 10 percent of the time needed to perform the pick activity (in which the modification takes place). The overhead of retrieving the changes from the repository is approximately 25 percent of the execution time.

Although this evaluation is preliminary, the results reflect the disadvantage of using aspect-oriented programming.[3] Our approach is clearly intrusive, since it interrupts the process execution at the end of each activity. However, using an optimized database design and in-memory solutions is likely to significantly improve both sets of mi-

RAISING THE ADAPTABILITY BAR

Emerging domains, new business needs, and novel applications require higher degrees of adaptability that traditional software programming languages cannot provide. In this context, some work emphasizes the convergence between DSPLs and SOAs,[1,2] but it typically resolves around the (simplistic) idea that features should map onto atomic services. For example, an extended version of Sassy,[3] a model-driven framework for self-architecting software systems, proposes the creation of a mapping between features and services. DSPLs need better support to describe their properties, such as dynamic variability and dynamic variation points, as well as the variants.[4]

A language for describing configurable reference models is a move in this direction, for example, to support the context-dependent configuration of processes, functions, and resources.[5] Unfortunately, the separation of concerns between base and additional features remains blurred, while CVL-based solutions foster it and thus help manage large, complex DSPLs.

The use of aspect-oriented programming to implement a DSPL is another option. This solution, also known as feature-oriented programming, fosters the implementation of features as aspects. For example, K@RT, an aspect-oriented and model-driven DSPL framework,[6] is based on runtime models that the system can modify during execution and check against constraints to ensure that reconfiguration is safe.

Another approach[7] uses dynamic aspects and runtime models to detect and solve context-dependent interactions among features. The designer models the reconfiguration of interacting features, and the runtime support ensures that the DSPL is delivered without conflicts. Although promising, the approach is still missing important features such as the possibility to add business rules, and thus change the dynamism of the system, during execution.

Finally, some solutions address dynamism by relying on context information. CAPucine[8] builds context-aware service-oriented product lines in which product derivations monitor their execution context and react by including appropriate software assets into the system. Provop[9] models a process family: while focusing on the relationships between variants and contexts, its runtime support can configure a specific process variant, but it cannot modify it at runtime.

References

1. R. Krut and S.C. Cohen, "Service-Oriented Architectures and Software Product Lines—Putting Both Together," *Proc. 11th Int'l Software Product Line Conf.* (SPLC 08), IEEE CS, 2008, p. 383.
2. P. Istoan et al., "Dynamic Software Product Lines for Service-Based Systems," *Proc. 9th Int'l Conf. Computer and Information Technology* (CCIT 09), IEEE CS, 2009, pp. 193-198.
3. H. Gomaa and K. Hashimoto, "Dynamic Software Adaptation for Service-Oriented Product Lines," *Proc. 15th Int'l Software Product Lines Conf.* (SPLC 11) workshop proc. vol. 2, I. Shaefer, I. John, and K. Schmid, eds., ACM, 2011, article 35.
4. S. O. Hallsteinsen et al., "Dynamic Software Product Lines," *Computer*, Apr. 2008, pp. 93-95.
5. M. Rosemann and W.M.P. van der Aalst, "A Configurable Reference Modeling Language," *Information Systems*, vol. 32, no. 1, 2007, pp. 1-23.
6. B. Morin, O. Barais, and J.-M. Jézéquel, "K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines," *Proc. 3rd Int'l Workshop Models@Runtime*, ACM, 2008, pp. 127-136; www.irisa.fr/triskell/publis/2008/Morin08e.pdf.
7. T. Dinkelaker et al., "A Dynamic Software Product Line Approach Using Aspect Models at Runtime," *Proc. 1st Int'l Workshop Composition: Objects, Aspects, Components, Services and Product Lines*, CEUR, 2010; http://tubiblio.ulb.tu-darmstadt.de/52434.
8. C. Parra, S. Blanc, and L. Duchien, "Context Awareness for Dynamic Service-Oriented Product Lines," *Proc. 13th Int'l Software Product Line Conf.* (SPLC 09), IEEE CS, 2009, pp. 131-140.
9. A. Hallerbach, T. Bauer, and M. Reichart, "Capturing Variability in Business Process Models: The Provop Approach," *J. Software Maintenance*, vol. 22, no. 6-7, 2010, pp. 519-546.

gration times. Even so, the migration of running process instances is not trivial and requires evaluating the tradeoff between delay and update criticality. Should the system postpone updates to complete running instances?

The problem becomes even more complex if we also consider process instances that cannot be migrated because their execution has already passed the point where the changes would need to be applied. Suitable rollback activities could restore the execution to a previous state, but the times would then be considerably higher than the figures in the table.

Converging DSPLs and SOAs provides a comprehensive adaptability solution, while DyBPEL contributes the machinery to manage and execute the resulting models. As the "Raising the Adaptability Bar" sidebar describes, we believe our solution has the potential for broad application because, on the one hand, it brings adaptability to DSPLs, and on the other, it provides BPEL process designers with the modeling support they need to understand and embrace more variability.

Our solution is not limited to BPEL; developers can exploit other business-oriented languages such as Business Process Modeling Notation to model the process and have BPEL be the hidden execution language (www.bpm.scitech.qut.edu.au/research/projects/oldprojects/babel/tools). Thus, a next step is to investigate modeling the product line with these different languages. Regardless of the language chosen, we have demonstrated the feasibility of combining the best of two worlds: CVL's flexibility and BPEL's structure.

Acknowledgments

The DyBPEL prototype is available free at http://home.dei.polimi.it/guinea/DyBPEL/DyBPEL.zip.

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero, and by the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7-ICT-2009-5, Project Indenica 257483.

References

1. D. Krafzig, K. Bamke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*, Prentice Hall, 2005.

2. H. Giese and B.H.C. Cheng, eds., *Proc. ICSE Symp. Software Engineering for Adaptive and Self-Managing Systems* (SEAMS 11), ACM, 2011.
3. L. Baresi and S. Guinea, "Self-Supervising BPEL Processes," *IEEE Trans. Software Eng.*, vol. 37, no. 2, 2011, pp. 247-263.
4. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming* (ECOOP 97), Springer, 1997, pp. 220-242.
5. D. Nebavides, S. Segura, and A.R. Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, 2010, pp. 615-636.
6. C. Kästner, S. Apel, and D.S. Batory, "A Case Study Implementing Features Using Aspect J," *Proc. 11th Int'l Software Product Lines Conf.* (SPLC 07), IEEE CS, 2007, pp. 223-232.

Luciano Baresi *is an associate professor of computer science at Politecnico di Milano, Italy. His research interests include distributed, dynamic, and mobile systems for service-oriented computing and the Web as well as software engineering. Baresi received a PhD in computer science from Politecnico di Milano. He is on the editorial board of* Transactions on Autonomous and Adaptive Systems. *Contact him at baresi@elet.polimi.it.*

Sam Guinea *is an assistant professor of computer science at Politecnico di Milano, Italy. His research interests include the application of software engineering principles to the design and runtime management of service-based systems. Guinea received a PhD in computer science from Politecnico di Milano. Contact him at guinea@elet.polimi.it.*

Liliana Pasquale *is a postdoctoral researcher in computer science at Lero—the Irish Software Engineering Research Centre. Her research interests include software requirements engineering, self-adaptive systems, service-based systems, security, and digital forensics. Pasquale received a PhD in computer science from Politecnico di Milano. Contact her at liliana.pasquale@lero.ie.*