Empowering the business analyst for on demand computing

A. Arsanjani

The tools, methods, and techniques used to create business architecture are often quite different from those used in developing software architecture. This "impedance mismatch" or gap is aggravated by volatile business requirements that need to be satisfied in operational systems. Bridging this gap not only allows a more seamless transition and faster time to market, but also enables and empowers business analysts to contribute their deep subject matter expertise at many phases of the software-development life cycle, a critical aid in fruitful application development. This paper presents a case study of a project with the U.S. Patent and Trademark Office (USPTO), which explored the potential for reducing duplication of effort among patent offices by sharing work products. IBM provided an innovative method to support the analysis, the "business compiler," a tool that implements Grammar-oriented Object Design (GOOD). GOOD is a method for creating and maintaining dynamically reconfigurable software architectures driven by business-process architectures. The business compiler was used to capture business processes within real-time workshops for various lines of business and create an executable simulation of the processes used.

The idea of using a process language to encode a software process as a "process model," and enacting this by using a process-sensitive environment is now well established. Over the past decade, a variety of process languages have been defined and applied to software-engineering environments, but their use has been limited.[1] The same is true of the use of process languages to model business processes. This is because although business process modeling and IT (information technology) interact in practice, suggesting that modeling in those areas should also be done in parallel, surprisingly few works have addressed the issue of the integrated modeling of business processes and IT.[2]

Process modeling is often done with visual tools (Visio, WBI Indexer, Rational Rose, etc.) and UML** (Unified Modeling Language**). Processes can be represented using activity diagrams or swimlane

Figure 1
Grammar-oriented Object Design (GOOD) as the intersection of several disciplines in computer science

diagrams[3] (among other notations). In addition to the symbols used in these diagrams, various other symbols have been used by different vendors to depict processing elements. Efforts have also been made in the direction of simulation of the process models.[4,5] There are various standards in this area, such as BPML (Business Process Modeling Language) and ebXML (Electronic Business Extensible Markup Language). BPML[6] is a meta-language for the modeling of business processes, just as XML is a meta-language for the modeling of business data. BPML provides an abstracted execution model for collaborative and transactional business processes based on the concept of a transactional finite-state machine. "BPML represents business processes as the interleaving of control flow, data flow, and event flow, while adding orthogonal design capabilities for business rules, security roles, and transaction contexts."[2] ebXML[7] is a suite of specifications designed to standardize the way businesses exchange messages, establish and manage trading relationships, and define business processes.

In contrast with these approaches, the technique presented here combines traditional business-process modeling with service-oriented architecture (SOA), domain-specific languages, and the use of metamodels to create executable business process specifications. These specifications can initially be modeled using existing graphics-based tools, but can later be translated into an underlying business domain grammar that can be used not only at the business level but at the IT level as well. This helps bridge the gap between business and IT by having a consistent representation of the process that can be carried through into development efforts.

This technique augments the notion of use cases with a "use case grammar" that describes the flow and sequence of interactions within a use case by using an executable language. In this way, use cases are augmented with domain-specific languages that allow a semi-formal specification of the domain to be executed.

The GOOD (Grammar-oriented Object Design) technique helps create business-driven dynamically reconfigurable architectures, a significant departure from current conventional thinking. Firstly, a process is viewed as a domain-specific language in a business domain rather than merely a set of partially ordered activities that are drawn by using a graphical design tool. Secondly, emphasis is given to how process models are developed, used, and enhanced over the duration of the software development life cycle by using GOOD. In particular, the issue of composing both new and existing model fragments, consolidating modules of business processes that are found to be common (common productions) across business lines or between companies or organizations, the achievement of consolidation through the use of a metamodel, and process verification through simulation via execution of the metamodel are all central to our analysis and development approach. This paper outlines these features and gives the motivations behind them.

The development of GOOD started in 1984 as an attempt to formalize valid interactions between objects in an object model and to formalize collaborations. This led to the augmentation of use cases with use-case grammars, which was later developed into an augmentation of OOAD (object-oriented architecture and design) methods to support the alignment of business and IT through creating a dynamically reconfigurable architectural style.[8] Key to this paradigm are the notions depicted in ***Figure 1***, including bridging the gap between business and IT through a common specification by representing business processes in a domain-specific language, or metamodel. This specification enables dynamic reconfiguration through the externalization of *manners* (i.e., rules for how a model behaves) of *context-aware components* (CACs). CACs are components that understand the environment or context

in which they are inserted. They are context-aware rather than context-sensitive; they gain insight into their environment rather than being tied to or dependent upon it.

To illustrate the GOOD technique, we present a case study involving business processes for patent offices. The top three intellectual property offices, as measured by the number of patents granted, are the European Patent Office (EPO), the Japan Patent Office (JPO), and the USPTO (United States Patent and Trademark Office). A very high percentage of patent applications are submitted to two or all three of these offices, resulting in a great deal of duplicate processing of the same invention. With the escalating growth of filings in all three patent offices and the increasing difficulty in keeping up with the demand for timely processing, workload reduction may be achieved by processing a given application in one office and sharing the results with the other offices. (There are many legal issues that must be resolved before work products can be shared; this paper addresses only the technical issues).

Using this example, we discuss the experience gained in applying GOOD to the problem of finding commonality and variations in business processes, for the purpose of workload reduction across multiple geographic areas. In addition, we explore the spectrum of applications of this paradigm, including support for and realization of on demand computing.

Problems and forces in the problem space

At a sufficiently abstract level, the functions and processes in all of the patent offices mentioned previously are very similar. The work products that might ultimately be shared (patent applications and correspondence between patent examiners and applicants) and how they are processed, however, reflect a wealth of low-level details that are quite different from one office to the other. The three patent offices agreed on a common XML vocabulary for publishing patent applications and grants, as well as for filing patent applications. In 2002, the offices explored the use of XML for prosecution (i.e., the processes that take place between the filing of a patent application and the granting of a patent).

The USPTO identified more than 300 distinct standardized forms and components of correspondence between applicant and examiner. JPO reported nearly 300 forms, and EPO reported approximately 3,000 (including many forms in multiple languages). Determining which forms were the best candidates for workload reduction and exchange required some basic description of the examination process at the three patent offices.

Prior business process reengineering at the USPTO produced results in the form of process descriptions, which reflected the idiosyncratic details of the processing of patent applications at this office. These results proved inadequate as a basis for determining the commonalities of process among the offices. No corresponding process descriptions for the other offices were available. For this and other reasons, the project was ultimately abandoned. Nevertheless, a number of results were achieved in the course of applying GOOD to this project that have proven useful in other contexts. The process itself can be fully generalized and used in other contexts.

Solution approach

A method was needed to capture from subject matter experts (SMEs) a description of their functions (elements of business functionality) and processes (end-to-end activities aimed at a goal based on business rules and policies) independent of the details of any particular implementation. To avoid excessive use of SME time, a permanent, negotiated record of the descriptions was required. Finally, a method was needed for transferring this description to developers with minimal opportunity for misinterpretation.

The first step of the project was the activity of finding commonality between activities and artifacts used by various offices. In order to do this, a metamodel of the business processes had to be created. However, this had to be a "scoped" metamodel. We controlled the scope of the metamodel by limiting it to high-impact correspondence events (correspondence between an applicant and the patent office). We decided to use GOOD for this metamodel because we needed to capture the processes in a near-natural language in order to run and simulate the process for verification by many SMEs. Looking at a sheet of paper for verification of business processes and use cases was not feasible for the dynamic environment of a workshop, where the simulations were to be verified.

We used a tool (the "business compiler"[9] [BC]) developed by IBM and based on GOOD to address the challenges of this project. At the time, very few if any tools were present that met all these require-

```
AccountManagement = {Open, Transactions, Close}
Transactions = {Login, Transaction, Logout}
Transaction = {{checkFunds, debitAccount} | creditAccount | Transfer | queryAcctBalance}
Transfer = {fundTransferAllowed, checkFunds, debitSourceAccount, creditDestinationAccount}
```

Figure 2
A sample grammar for account management by the business analyst

ments in an integrated manner. The tool's graphical user interface (GUI) was customized based on the requirements and significant input of the USPTO, and functionality was added by using the tool's plug-in development facility to incorporate new requirements.

In the following sections, we give a brief overview of the theoretical foundations of GOOD, explore the BC tool itself along with its usage scenarios, and finally discuss lessons learned from the application of GOOD and the BC tool to this project.

DESIGNING AN EXECUTABLE METAMODEL WITH GOOD

GOOD is the application of business-domain-specific languages to objects, components, and services to implement context-aware software components that are business-driven and facilitate the creation of dynamically reconfigurable business-driven architectures (BDAs).[10] In many cases, IT systems often lose their direct connection and traceability back to the business models, processes, and goals that they support. As the business changes with no explicit or implicit connections to IT, maintaining the support of the IT systems for new processes and business capabilities becomes an overwhelming challenge to IT departments. By designing not just object-oriented or component-based systems, but those driven by a business language that can be defined and maintained in conjunction with the business analysts, we empower the business analysts to participate and contribute their knowledge to the software-development process. By having the business analysts maintain the business language defined by a business grammar, we can create executable models of business systems. The business grammar is defined by the business analyst and refined by the IT architect. The engine executing the grammar (and its input) serves as the role of the "controller" in the dynamic controller architecture. This controller is written in domain-specific language. The other roles of the MVC model, namely, the "view" and "model," follow more traditional means of general-purpose languages such as Java. These models of business domains can be generalized into metamodels that reflect the commonality and variations within a domain. They can be represented by a domain-specific business grammar and used to simulate IT systems.

A grammar describes a (potentially infinite) set of patterns in terms of a finite lexicon and a finite set of rules or constraints that specify allowable combinations of the elements in the lexicon. Similarly, a business grammar consists of the lexicon (key abstractions and relationships) for a business domain along with the policies and rules of how the lexicon elements should be combined into meaningful operational constructs that reflect business goals, processes, and imperatives. These combinations represent the semantics of the structure as well as the process flow of the operational systems within the domain. This paradigm is based on a combination of areas in computer science, including domain-specific languages and component-based architectures, service-oriented software computing, business rules, patterns, analysis and design methods, creation of metamodels, and compilers, as shown in Figure 1. Using GOOD, we empower the business analyst to develop a business grammar that then drives the specification and execution of the application.

As an example of a business grammar describing the processing of a business domain, consider account management in a bank. *Figure 2* shows a declarative way of specifying functionality that can be augmented with functional specifications. As shown in the figure, account management consists of opening accounts, performing transactions, and closing accounts. We can consider each of these functional elements as primitives of the language of banking account management or refine each one into a more detailed set of steps. Elements shown in the figure in leading lowercase, such as "checkFunds," or "debitSourceAccount," are "atomic" elements or

```
Account Mgt = {Login, Txns, Logout}
Login = {#displayLoginPage, login, #getUserIdAndPassword, #login, CheckLoginResult}
CheckLoginResult = {{success, DisplayMenu}      | {invalidUserIdOrPassword, #displayInvalidUserError, Login} |
newError, #displayNewError, Login}
DisplayMenu = {displayMenu, #displayMenu, #getTxnType}
Txns = {Txn, DisplayMenu, Txns} | end
Txn = { {accountInfo, AccountInfo}| {debit, Debit}| {credit, Credit}| {transfer, Transfer}}
AccountInfo = {getAccount, #getAccount, #displayAccountInfo(account)}
Credit = {getCreditParameters, #getCreditParameters, #performCredit(srcAccount, amount)}
Debit = {getDebitParameters, #getDebitParameters, #getCustomerType, CheckFund, CheckFundResult}
CheckFundResult = {{success, #performDebit(srcAccount, amount), CheckTxnResult}
            | {invalidAmount, #displayAmtError}
            | {insufficientFund, #displayFundError}
            | {error, #displayGeneralError}
}
CheckFund = {regularCustomer, #checkFund(srcAccount, amount)| platinumCustomer,
#checkCredit(srcAccount.getBalance()-amount)}
CheckTxnResult = {{success, #displayDebitConfirmation}| {invalidDebitAmount, #displayDebitAmtError}| {error, #
displayGeneralError}}
```

Figure 3
Architect adds more detail to the grammar for account management

terminals of the grammar; elements shown in leading uppercase such as "Transfer" are elaborated further in a subsequent (production) rule. "Transfer" is a non-terminal symbol, and appears as the "lefthand side" or definition of another declarative rule which indicates that before a transfer can be performed, the atomic elements "fundTransferAllowed" (to determine whether a transfer is allowed for this account) and "CheckFunds" (to determine whether there are sufficient funds for the transfer) must be used. If these conditions are met, the source account is debited and the amount is credited to the destination account.

A more detailed, executable, account management grammar with error handling added by the IT architect to the initial flow provided by the business analyst would look something like that shown in ***Figure 3***. The hash sign (#) represents a service call to invoke a service at that point in the grammar. Note that in the example shown in Figure 3, the architect would take the grammar provided as the initial specification by the business analyst and augment it with details pertaining to handling of errors, more detailed flow, and invocation of required services to provide input into the next step in the process. In other words, the "what" of the specification is developed by the business analyst, and the "how" or realization of the specification in the design is done by the architect, taking the same artifact and the grammar and adding more detail, instead of creating totally new and different representations for design and execution. Thus, the tools and methods employed converge and are elaborated rather than totally overhauled as we make the transition from business to IT.

GOOD[11] allows business analysts to capture the requirements of structure and flow of a business process. "Structure" refers to the composition of more static elements in the domain such as patents, examiners, first office action, etc. "Flow" refers to the sequence of activities within a goal-oriented business process, such as the patent examination process. In addition, business analysts can accomplish this process specification and execution not in a predefined executable language like Java,** but in a business-domain-specific language (that they themselves define) that helps convey requirements of the structure and flow of business processes for their domain (e.g., patent processing). ***Figure 4*** shows a segment of the USPTO grammar.

Structured prose from the business domain was used to craft the grammar shown in Figure 4. As a grammar involves the formation of basic linguistic units, a business grammar involves the formation of business functions. Thus a business domain grammar depicts the rules governing the formation of valid sequences of use cases. This has been referred to as a use-case grammar when applied to a single use case.[11–13] Use-case models are created as part of the analysis phase of a software project. Using GOOD, use cases are augmented with a grammar that

```
USPTO Process = {Filing, Pre-examination, Publication, Examination<and Correspondence>, Post-
examination }
Filing = {< details suppressed>}
Publication = {< details suppressed >}
Pre-examination = {OIPE(office of initial patent exam) or PCT(patent cooperation treaty)-Operations}
OIPE or PCT- Operations = {Assign Serial Number, Record Fees, Tentative Classification, Screen For
Security, [electronic word search], PICS Scanning, Licensing, Separate Regular From Security
Processing, Administrative Examination, Data Entry, Filing Receipt Mailed, Initial Data Capture, Initial
Preparation and Electronic Capture for Printing and Issue, Preliminary Amendments}
Assign Serial Number = {paper-based | electronic-using-ePAV}
<etc.>
```

Figure 4
Segment of USPTO grammar

defines the sequence, alternation, and repetition of the composition of use cases. This dynamic interaction among use cases is currently missing from use case analysis. The business processes are realized by a set of use cases whose dynamic behavior is now governed by the business-domain grammar.

A grammar is essentially a set of rules for describing sentences. More formally, a grammar G is a quadruple $\{N, T, S, P\}$ with the four components:

1. N—a finite set of nonterminal symbols
2. T—a finite set of terminal symbols
3. S—a special distinguished symbol, such as "goal" or "start"
4. P—a finite set of production rules or, simply, productions

This produces highly readable and understandable specifications that are, at the same time, executable on a virtual machine. The business-compiler tool debugging and execution environment allows the near-natural language capture of requirements that can be debugged to ensure that they are executable. The business compiler executes the grammar based on events that are manually or programmatically triggered during the execution of a program (or Web service) or the simulation of the processes. The business analyst can provide the initial grammar, and the IT architect can add more detailed design and implementation aspects (such as the user interface, access to existing systems, the invocation of external services, etc.) to render the grammar executable. Thus "grammar-oriented" refers to the fact that the grammar or vocabulary used by the business architect or analyst is the basis for an executable language that can be used in conjunction with object-oriented or service-oriented analysis and design to produce applications that are written in a combination of the traditional computer languages such as

- ***Sequence***
  - *Process = {DoThisFIrst, DoThisSecond, DoThisThird}*
  - **Example: USPTO Process = { Filing, Pre-examination, Publication, Examination, Post-examination }**
- ***Conditionals, Branching***
  - *Process = {DoThisFirst | YouMayDoThisFirst, DoThisSecond}*
  - **Example: Pre-examination = { OIPE Operations | PCT Operations}**
- ***Loops***
  - **Example: Txns = {displayMenu, Txn, Txns} | doc received**
- ***Comments***
  - **//This is a comment**

Figure 5
Syntax of EBDL

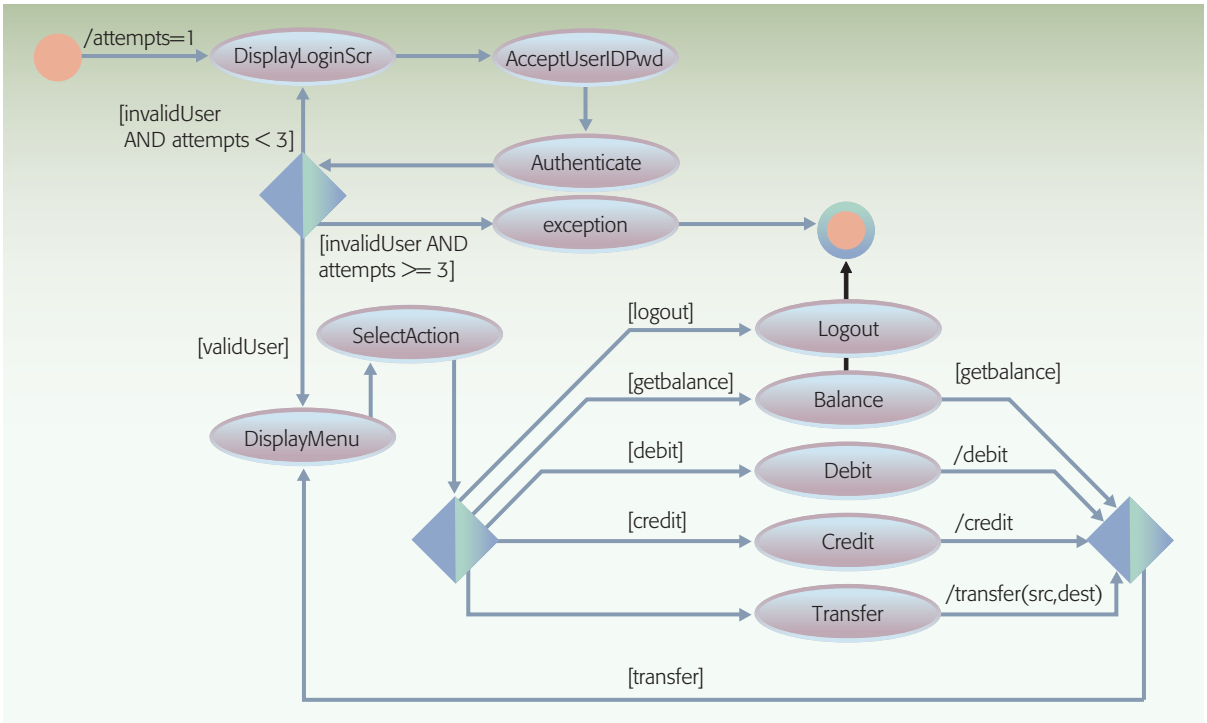


Figure 6
Using activity diagrams

Java and domain-specific languages that are designed for particular industry domains.

The syntax of the executable domain-specific business language (EDBL) used by the business compiler is simple and is shown in ***Figure 5***. Typically, business analysts are able to start writing specifications using their own EDBLs in a few days, with a 2-day workshop and some practice and assistance.

Although a picture is said to be worth a thousand words, sometimes pictures with hundreds of boxes and lines are less readable than plain text. To address this, there are two input formats to the BC, the textual language just described and an additional method using graphical input such as UML activity diagrams, as depicted in ***Figure 6***. The BC transforms the XML-based output of tools such as Rational* XDE (Extended Development Environment) and creates an intermediate representation that uses XSD (the XML Schema Definition language). This same representation is generated from the simple text input that can be imported as a regular text file by the BC to generate its code. The multiple input formats and the intermediate standard XML format are depicted in ***Figure 7***.

RELATION WITH TRADITIONAL APPLICATION ARCHITECTURES

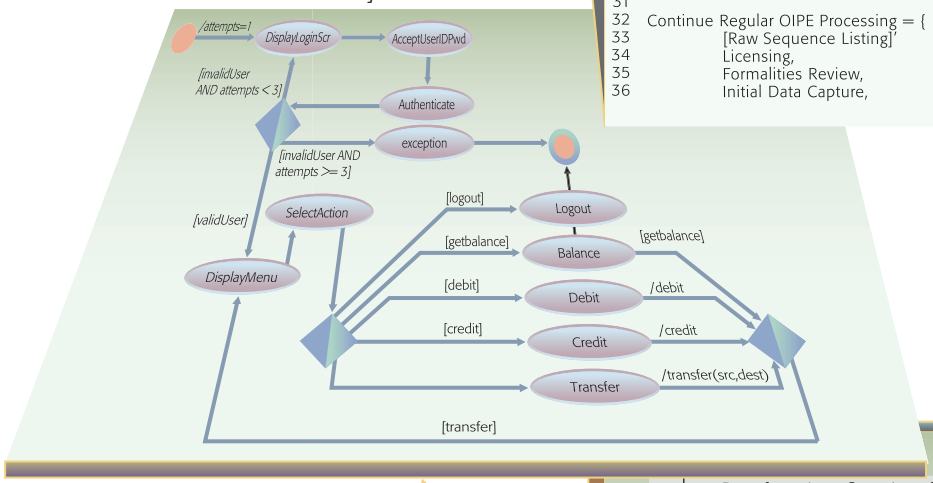
In a model-view-controller view of an architecture (see ***Figure 8***), the language used to build the view, controller, and model is a traditional third-generation computer language. Using GOOD, the controller, as depicted in ***Figure 9***, is written in a business-domain-specific language that has been defined by a business SME (a business analyst or architect). This introduces a novel method of empowering analysts to participate and contribute their knowledge on an ongoing basis throughout the application and architecture development and maintenance cycles. The initial flow is provided by the business analyst and then subsequently refined through cooperation with the IT architect.

In this new architecture, the controller is implemented at a higher level of abstraction in a business-domain-specific language. The business compiler assumes the role of a controller and executes the steps described in the business language similarly to steps or activities in a workflow. Alternatively, an enterprise component can have an embedded business compiler engine that can control the microflow within the component or act as an orchestrating

Textual Representation
of Grammar
[Easy to maintain]

```
initial symbol = Start

Start = {start, #initTestData, USPTO Process, Epilogue}

USPTO Process = {Filing, Pre-examination, Publication, Examination, Post-examination

Filing = {#msg(Filing), #msg([we don't have details.])  }

Pre-examination = { OIPE Operations | PCT Operations,

            Pre-examination
}
PCT Operations = {#msg(PCT Operations placeholder) }
OIPE Operations = {
            Manual Serial Number Assignment,
            Response From Applicant Arrives,
            Post Fees to RAM,
            Document Preparation,
            Scan and Index,  |
            Tentative Classification,
            Screen for Security Sensitive Contents,
            Separate Regular From Security Processing
}

Separate Regular From Security Processing = {
            Review Security Status,
                {notSecurityOK, make paper copy, destroy electronic documents, file secure
            |   {securityOK, Continue Regular OIPE Processing
                  }
}

Continue Regular OIPE Processing = {
            [Raw Sequence Listing]'
            Licensing,
            Formalities Review,
            Initial Data Capture,
```

UML Activity Diagram
[Easy to visualize and create first iterations]



XML Representation
of Grammar
[Appropriate for computer processing]

```
            <right>
                <symbol type="terminal">next</symbol>
                <symbol type="nonterminal">OIPE or PCT- Operations</symbol>
            <right>
        </production>
        <production>
            <left>
                <symbol type="nonterminal"> OIPE or PCT- Operations</symbol>
            </left>
            <right>
                <symbol type="terminal">next</symbol>
                <symbol type="nonterminal">Assign Serial Number</symbol>
                <symbol type="nonterminal">Record Fees</symbol>
                <symbol type="nonterminal">Tentative Classification</symbol>
                <symbol type="nonterminal">Screen For Security Sensitive
                Contents</symbol>
                <symbol type="nonterminal">PICS Scanning</symbol>
                <symbol type="nonterminal">Licensing</symbol>
                <symbol type="nonterminal">Separate Regular From Security
                Processing</symbol>
                <symbol type="nonterminal">Administrative Examination</symbol>
                <symbol type="nonterminal">Data Entry</symbol>
                <symbol type="nonterminal">Filing Receipt Mailed</symbol>
                <symbol type="nonterminal">Initial Data Capture</symbol>
                <symbol type="nonterminal">Initial Preparation and Electronic
                Capture for Printing and Issue</symbol>
                <symbol type="nonterminal">Preliminary Amendments</symbol>
            </right>
        </production>
        <production>
            <left>
                <symbol type="nonterminal">Assign Serial Number</symbol>
            </left>
            <right>
                <symbol type="terminal">next</symbol>
```

Figure 7
GOOD business compiler tools

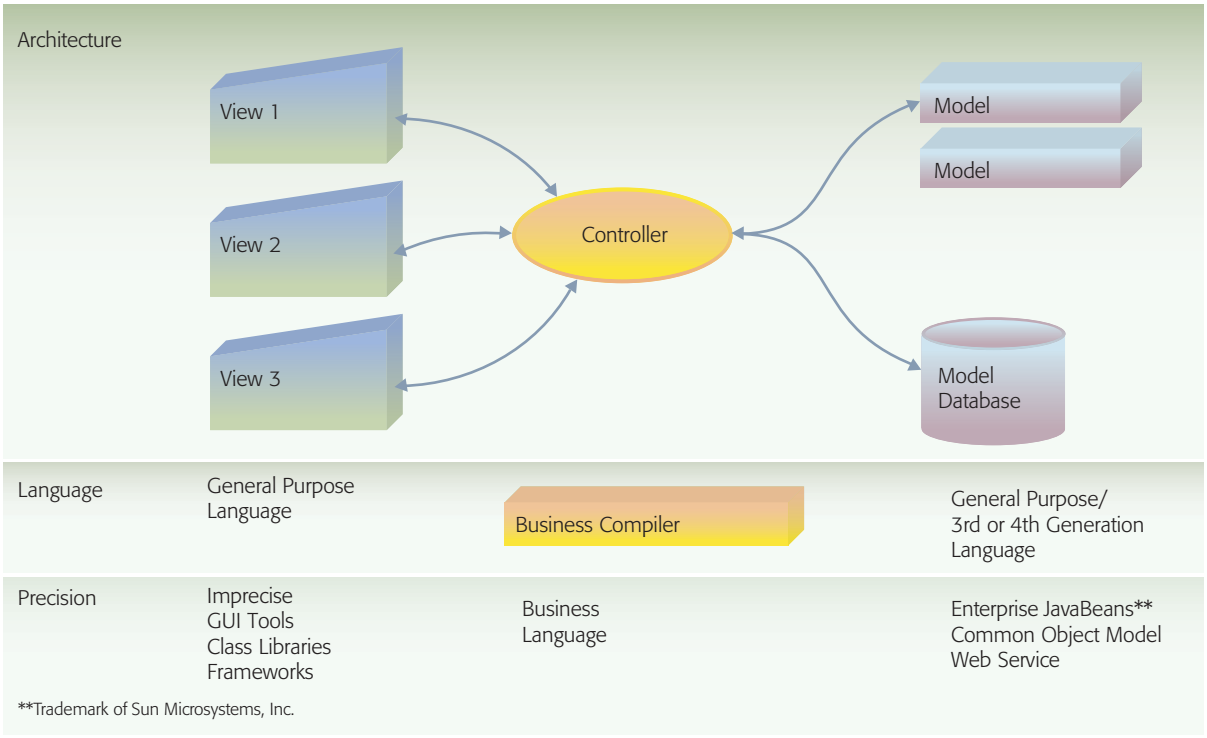


Figure 8
MVC Architecture

mechanism between components. The latter behavior is akin to that of a BPEL4WS[14] engine. In GOOD the grammar, or more precisely the parser, is the controller of the architecture. Changes can be made to the grammar, and the flow will thus be altered. The domain-specific language used can be translated into a BPEL4WS representation to be run on tools such as the IBM Process Choreographer. Prototyping is made possible by allowing the executable flow of the application to be defined at a high level of abstraction.

Usage scenarios for executable metamodels using the GOOD paradigm

GOOD and the business compiler can help address a spectrum of issues or problems ranging from business architecture applications to software engineering activities. Business architecture activities include the executable specification of business processes, their simulation, their re-engineering and consolidation, and the finding of commonalities and variations among them. Using GOOD, we can create a metamodel of multiple business processes across business lines or organizations to serve various objectives, including harmonization of different operational and business models to promote cooperation.

Another usage scenario involves assisting in software development. The BC was used to create a simulation of patent processing including bringing up screens of actual applications. This was found to have applicability in training new patent examiners and personnel in using existing systems as well as understanding the processes underlying them.

Solution implementation

GOOD methods were used to capture the business process flow in the interviews with SMEs in near-English prose, a process which was much faster and more participatory than diagramming the flow. The intent was to capture the flow of patent application processing which reflects the business processes involved and to record them in a manner that is not only readable by humans but also executable as a simulation and that can be integrated seamlessly into an application.

The business compiler supports many aspects of the business-modeling process and that of creating metamodels. This includes requirements gathering,

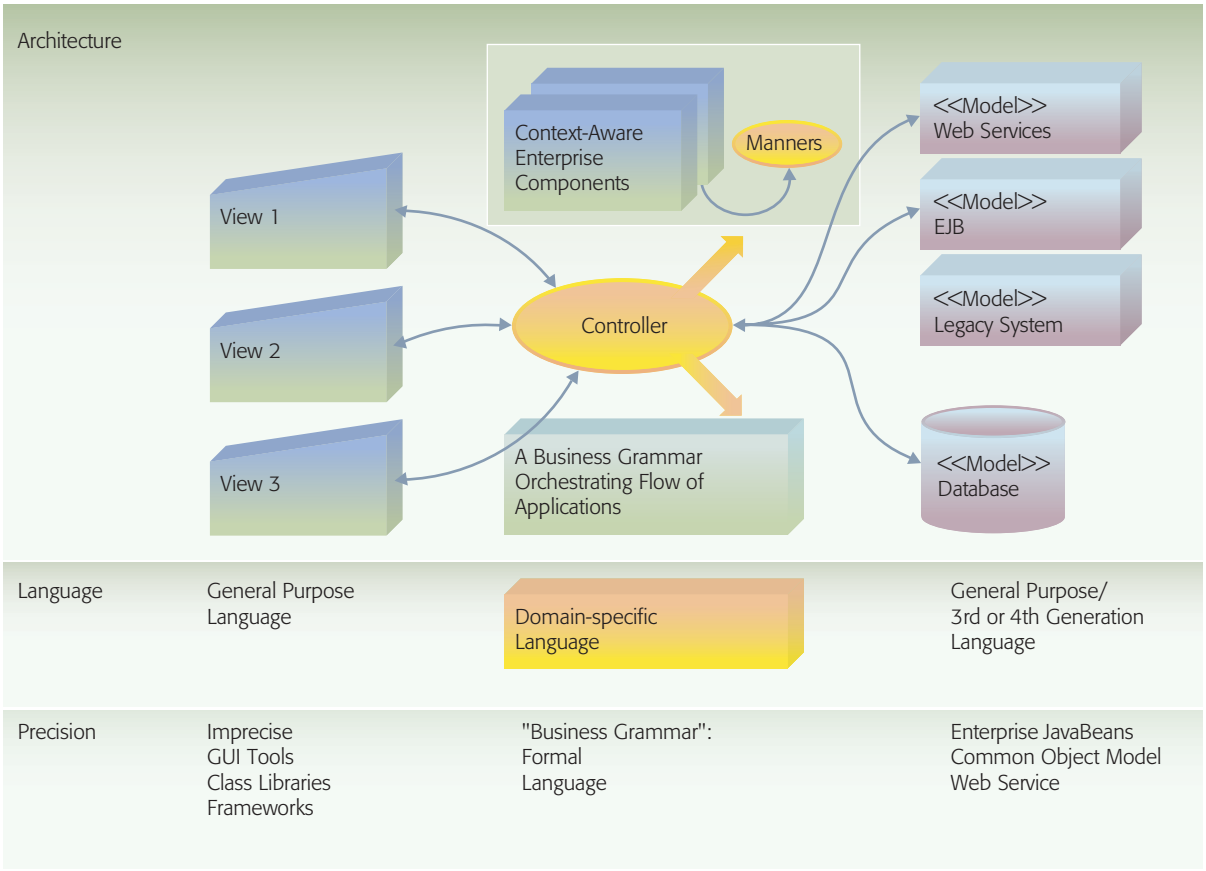


Figure 9
Dynamic controller architecture

simulation, verification, and testing of the business processes outlined within the scope of the metamodel mentioned previously. The solution approach (shown in ***Figure 10***) was to create a metamodel of business processes, uncover information on commonalities and variations, and feed this information back into the initiatives relating to standardization of DTDs (data type definitions) among patent offices. The initial set of elements of the DTD standards for correspondence between an applicant and the patent office would be matched against the business use cases coming out of the business-process modeling activities. These would serve to support the identification of commonalities by examining use cases which lead to the creation of common work products. This would in turn benefit the workload reduction and harmonization effort.

A METAMODEL OF THE GAP BETWEEN BUSINESS AND IT

The gap between business and IT is primarily due to the different terminology, levels of granularity, varied models, approaches, tools, and methods that each employ. In order to support the emerging on demand computing paradigm, this gap must be bridged, and common (or at least partially common) models and representations can aid in this effort.

The on demand paradigm requires an organization to be agile and adaptive, to "... respond to any opportunity or threat,"[15] and to maintain IT systems that support business process and model changes in "right-time" (i.e., not necessarily real time). For this goal, it is essential to acquire the ability to define the "configuration" of a business (as in its value net) and an IT system (as in the static and dynamic aspects of system operation). After this is done, it is necessary to represent the interdependence between the two, such that the IT configuration supports the business model and can change accordingly when that model needs to be changed. This adaptive capability entails the ability to reconfigure a business architecture's value net and the components,

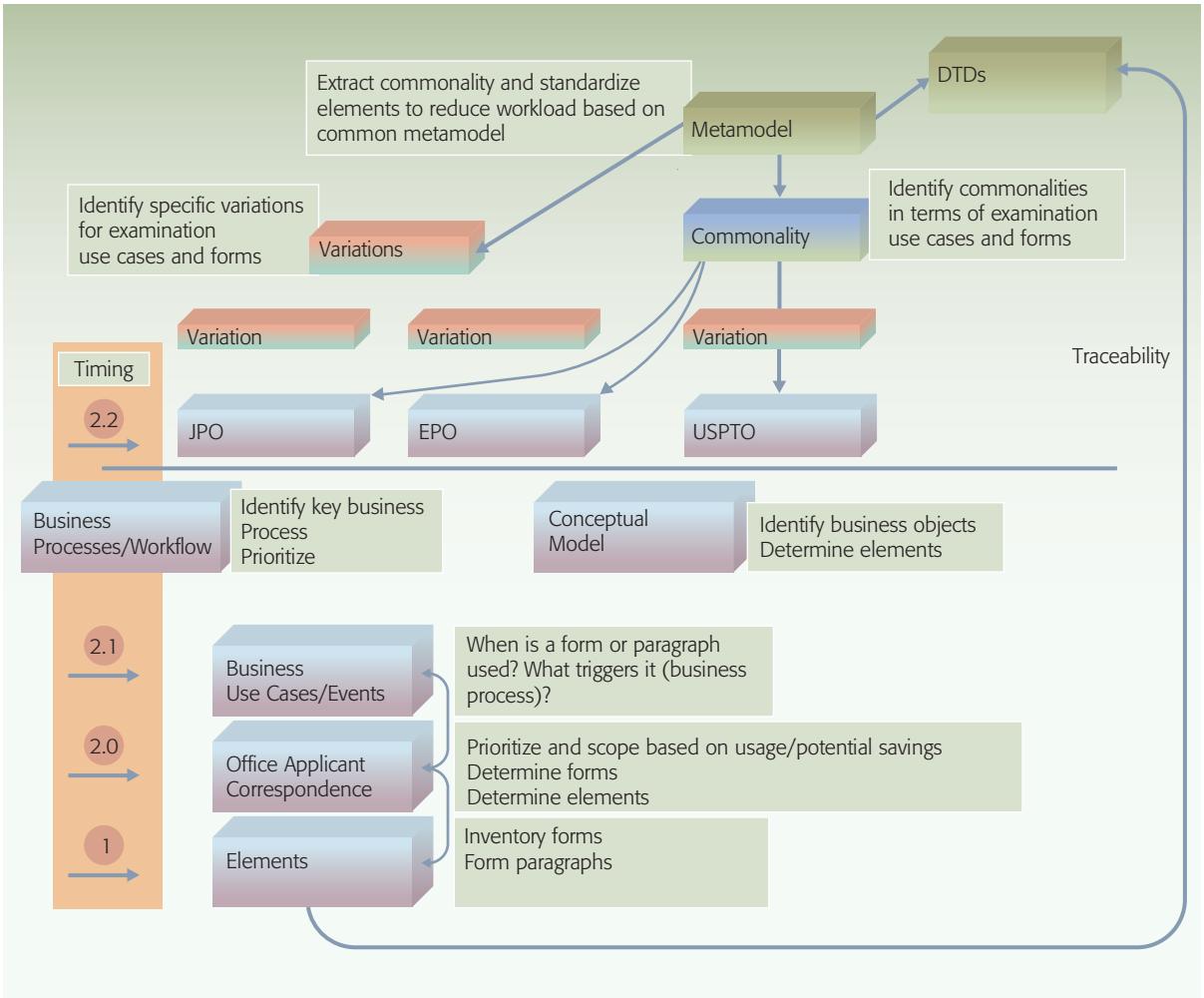


Figure 10
Proposed solution approach

services, composition, and flow of a software architecture.

Standards and their implementations, such as BPEL4WS, are a step in that direction. They allow the creation of composite service-oriented applications from a set of underlying published services. One key differentiator between GOOD and BPEL4WS is that in GOOD, the domain's language with its structural and functional ramifications remains a first-class construct. The structure and flow defined by a domain grammar drives the collaboration and choreography of services.

Figure 11 depicts a metamodel of the concepts relating to the problem and a solution to the gap between business and IT. Going from the top to the bottom of the figure, we see an emerging need for a new computing paradigm that offers agile responsiveness to changes in requirements. On the one hand (on the left), we have a business model or business architecture comprised of the business processes, their goals and objectives: the decomposition of the value chain into functional areas that work together within the context of a business process to bring value to the business. On the other hand, we have software architecture in which applications are compositions of components. Both business and software architecture need to be reconfigured as new opportunities, threats, mergers, and acquisitions occur. The applications offer services to provide functionality supporting the business.

The gap can be bridged by bringing these two traditionally diverging architectures into a conver-

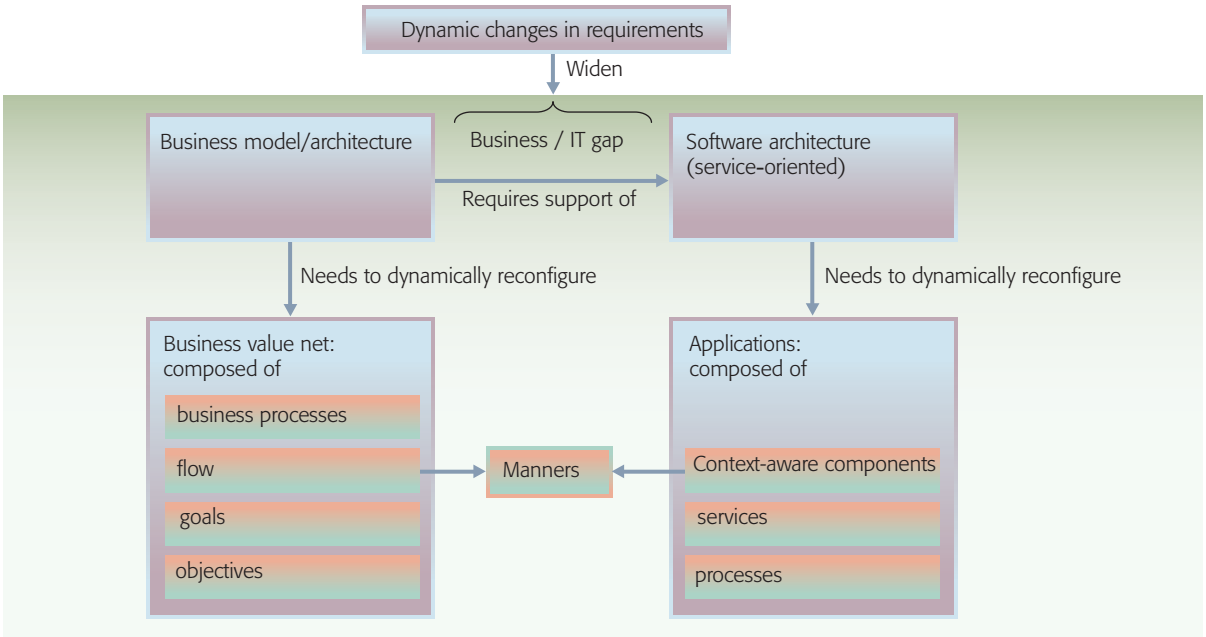


Figure 11
Dynamic reconfiguration, using GOOD, to bridge the business/IT gap

gent path. This can be accomplished through providing each with the ability to be dynamically reconfigured; that is, business value chains can be reconfigured to leverage new markets, forge new partnerships, and offer new products and services. This must be synchronized with the ability of the software architecture to adapt and reconfigure its composition and business rules to support these new partnerships, products, and services, and to do so under the constraints within which they must operate to be profitable. At the same time, the business must maintain the newly forged service-level agreements within this new value chain, delegating this responsibility to the software architecture and the applications, components, and services running on it.

To enable dynamic reconfiguration, the software architecture must handle variations in business context. It can do so through the design and implementation of CACs, whose functionality is self-optimized for the context in which they are used. To implement context-awareness without sacrificing wide-range applicability in multiple areas, the component must use a set of rules and policies to determine how it should behave in different contexts. These variations in context are externalized and stored by the designer in configurations for the component. *Manners* are properties of CAC services that can adapt to changes in their business context based on policies and externalized variations. The semantics of CACs and services are thus externalized in their manners.[16]

Figure 12 depicts the behavior of CACs. After the component's internal flow and external collaborations have been defined by using the business grammars, these can be externalized into the configurable profile as metadata that can be loaded and executed. The typical internal behavior of CACs is shown in Figure 12 as follows. When an event or message is received, the CAC checks its context and state (often passed to it with the event or message). An example of a message is, "An order for a platinum wholesale customer is being processed, and we are in a secure, authenticated, authorized transactional context." (Steps 1 and 2)

The appropriate policies ("rules about which rules to apply") are selected. For example, "For platinum and wholesale clients, we need to access the policies and select the appropriate rules, based on customer type, order type, account type, and transaction type." (Steps 3 and 4)

After the policy is selected and the rules to be applied are determined, access to the state of the incoming message is again needed to actually

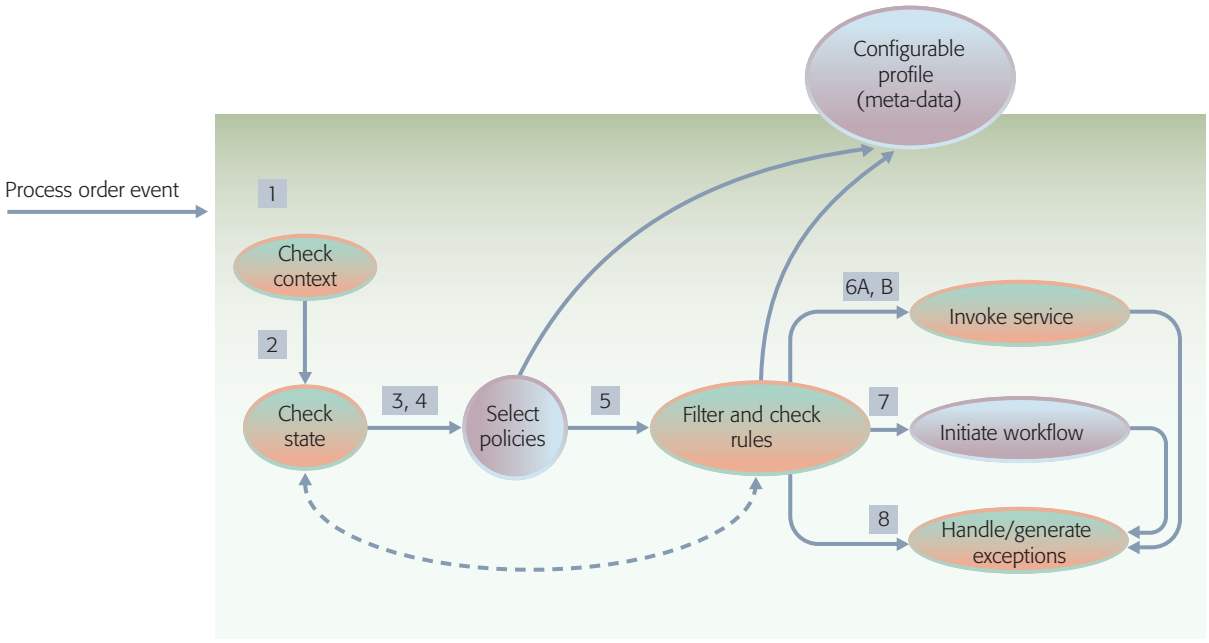


Figure 12
Behavior of CACs for order entry scenario

perform the rule check. The policies and rules are applied (Step 5). Based on the result, a service may be invoked and may participate in an orchestrated macroflow, or an exception may be generated through message-oriented middleware capabilities if an error condition arises. (This is shown in Steps 6A and 6B, which are conducted in parallel.) At this point the order has been received, the method of payment applied, and the processing of the customer's fulfillment will follow.

A workflow is then initiated to manufacture or pick and ship the products ordered, and a message is sent to the client (Step 7). If the result of the rule check is negative, an exception may need to be handled (Step 8). Also, after a service has been invoked or a workflow initiated, an exception may occur which needs to be handled.

CONCLUSIONS

In the USPTO implementation, using the business compiler entailed an initial learning period of a few days, followed by periodic interactions (several hours a week) for a few weeks. Applying GOOD through using the business-compiler tool helped capture business processes in a dynamically updatable manner, one which was amenable to larger workshop sessions where each SME contributed a small portion of the overall process. The process could be executed and the decision points evaluated by all participants. This led to the identification of the commonalities among business processes.

For many users, writing a business grammar in the form of an outline seems a less daunting task than learning an entirely new tool or notation, such as UML or WBI-Modeler. Others may prefer to rely on the business analyst for design and use tools like UML for maintenance purposes.

This paradigm allows participation of the business analyst in the development life cycle in such a way as to complete and augment the traditional roles of architect and developer through the provision of an executable specification of the business model and flow. This approach contrasts with runtime textual documents that typically cannot be interpreted and thus are vague and challenging to use as a basis for structuring IT solutions supporting changing business needs in an on demand world. When business processes are broken down, they ultimately decompose into messages communicated between knowledge workers. As a result, the standardization of those messages, such as in the current USPTO efforts, is critical.

ACKNOWLEDGMENTS

The author would like to thank Bruce B. Cox of the United States Patent and Trademark Office for his contributions and insight into this paper and his management, cooperation, and contributions to the project, without which this project would not have been implemented or refined. Thanks go to Thomas Moewe and David Ng as well, for their contributions to the project.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

CITED REFERENCES AND NOTE

1. B. C. Warboys, D. Balasubramaniam, R. M. Greenwood, G. N. C. Kirby, K. Mayes, R. Morrison, and D. S. Munro, "Collaboration and Composition: Issues for a Second Generation Process Language," *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (October 1999) pp. 75–90.
2. B. Curtis, M. I. Kellner, and J. Over, "Process Modeling," *Communications of the ACM* **35**, No. 9, pp. 75–90 (September 1992).
3. A *swimlane* is a visual region in an activity diagram that indicates the element that has responsibility for action states within the region.
4. W. J. Kettinger and J. T. C. Teng, "Business Process Change: A Study of Methodologies, Techniques, and Tools," *MIS Quarterly* **21**, No. 1, 55–80 (March 1997).
5. M. A. Ould, *Business Processes: Modeling and Analysis for Re-engineering and Improvement*, John Wiley & Sons, Chichester (1995).
6. BPML, http://www.bpmi.org/bpml.esp.
7. ebXML—Enabling a Global Electronic Market, www.ebxml.org.
8. A. Arsanjani, "Explicit Representation of Service Semantics: Towards Automated Composition Through a Dynamically Reconfigurable Architectural Style for On Demand Computing," *Proceedings of the International Conference on Web Services (ICWS 2003)* (2003), pp. 34–37.
9. A. Arsanjani, "Business Compilers: Towards Supporting a Highly Reconfigurable Architectural Style for Service-Oriented Architecture," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada (2002), pp. 287–289.
10. A. Arsanjani, "A Domain-Language Approach to Designing Dynamic Enterprise Component-Based Architectures to Support Business Services," *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* (2001), pp. 130–142.
11. A. Arsanjani, "Introduction to Special Issue on Enterprise Components and Services," *Communications of the ACM* **45**, No. 10, pp. 30–34 (2002).
12. A. Arsanjani, J. J. Alpigini, and H. Zedan, "Externalizing Component Manners to Achieve Greater Maintainability through a Highly Re-Configurable Architectural Style," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada (2002), pp. 628–639.
13. K. Levi, A. Arsanjani: "A Goal-Driven Approach to Enterprise Component Identification and Specification," *Communications of the ACM* **45**, No. 10, 45–52 (2002).
14. *BPEL4WS—Business Process Execution Language*, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
15. S. Palmisano, *The New Agenda*, IBM General Webcasts (October 2002), http://www-306.ibm.com/webcasts/WCPGateway.wss?jadeAction=WEBCAST_BROWSETIER2_HANDLER&WCP_NAV_ID_KEY=0403.
16. A. Arsanjani, "Grammar-Oriented Object Design: Creating Adaptive Collaborations and Dynamic Configurations with Self-Describing Components and Services," *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* (2001), pp. 409–414.

GENERAL REFERENCES

G. M. Giaglis, R. J. Paul, and A. Serrano, "Reconciliation of Business and Systems Modeling via Discrete Event Simulation (December 1999)," *Proceedings of the 31st Conference on Winter Simulation: Simulation—A Bridge to the Future*– Volume 2, pp. 1403–1409.

G. M. Giaglis, R. J. Paul, and Georgios I. Doukidis, "Simulation for Intra- and Inter-Organisational Business Process Modelling," *Proceedings of the 28th Conference on Winter Simulation*, pp. 1297–1304 (December 08–11, 1996), Coronado, California.

Accepted for publication September 6, 2004
Internet publication January 10, 2005.

Ali Arsanjani
IBM Global Services/Application Management Services, 1804 A Padmavani Lane, Fairfield IA 52556 (arsanjan@us.ibm.com).
Dr. Arsanjani is a Senior Technical Staff Member and Executive IT Architect with 21 years of experience in software consulting, development, and architecture. He is a chief architect in the IBM Global Services SOA and Web Services Center of Excellence. His areas of expertise include service-oriented and component-based software architecture, patterns, and methods. He has written extensively on patterns, service-oriented architecture (SOA), component-based development and integration, business rules, and dynamically reconfigurable software architecture. He is also an Adjunct Associate Professor of Computer Science at Maharishi University of Management in Fairfield, Iowa. ■