# Autonomic Execution of Computational Workflows

Tomasz Haupt, Nitin Sukhija, Igor Zhuk
Mississippi State University
Center for Advanced Vehicular Systems, Box 5405
Mississippi State, MS 39762
USA
Email: {haupt, nitin, igorzhuk}@cavs.msstate.edu

*Abstract*—**This paper describes the application of an autonomic paradigm to manage the complexity of software systems such as computational workflows. To demonstrate our approach, the workflow and the services comprising it are treated as managed resources controlled by hierarchically organized autonomic managers. By applying service-oriented software engineering principles, in particular enterprise integration patterns, we have developed a scalable, agile, self-healing environment for execution of dynamic, data-driven workflows which are capable of assuring scientific fidelity despite unavoidable faults and without human intervention.**

## I. Introduction

Support for scientific workflows is now recognized as a crucial element of cyberinfrastructure, facilitating e-Science. Typically sitting on top of a middleware layer, scientific workflows are means by which scientists can model, design, execute, debug, re-configure and re-run their analysis and visualization pipelines.

There are many ways of implementing scientific workflows [1, 2]; however, with the advent of Grid and Cloud computing, most of the current efforts adopt Service-Oriented Architectures (SOA). Consequently, research on workflow management systems highlights methodologies of service composition and orchestration. To that end, this paper focuses on particular aspects of service-oriented workflow system development, namely, the scientific fidelity, fault tolerance, adaptivity, and management of complexity. The ideas presented in this paper are illustrated by an exemplary implementation of an adaptive computational workflow.

Scientific fidelity refers to a software system's ability to deliver reliable, trustworthy computational results; i.e., the end user can trust that the output is not distorted by erroneous information resulting from unreported failures of the workflow and/or its components. To achieve such fidelity, the system for executing the computational workflows must be capable of detecting faults and abnormalities and performing corrective actions, whenever feasible. The system can react to faults and abnormalities either by protecting against faults before they occur (possible when an abnormality has been detected), or by recovering after a fault has happened. In the latter case, the detection of a fault may also help detect an abnormality, which could then prompt a corrective action to prevent future failures of the same type.

In addition to a direct recovery from a point failure by automatic fixing the cause of the problem and retrying, it is desirable that the system has a capability to respond to an abnormality by adaptation. It may include use of an alternative service instance, correction of the request due to a change of the service interface, the selection of an alternative algorithm to be used by the service (or the code submitted by that service), or the modification of the workflow specification, i.e., the change of the execution path, perhaps using alterative or optional workflow nodes. Since the adaptations forced by the failures may be data-driven and thus unpredictable, enforcing the scientific fidelity is of critical importance.

Unfortunately, the enforcement of scientific fidelity adds to the complexity of the system; if not managed properly, this added complexity might actually decrease the reliability and maintainability of the overall system, thereby defeating its ultimate purpose. The situation is further complicated by the fact that the end user, a domain specialist that composes and runs the workflow, may not know or care about possible failure modes below the application level or the methods for remedying them. Conversely, an IT specialist maintaining the system typically has very little, if any, knowledge of the business logic of the workflow.

Herein, we address scientific fidelity, fault tolerance, adaptivity, and the management of complexity, applying (1) the concepts of Autonomic Computing, in particular self-management and self-healing, and (2) service-oriented software engineering, in particular exploiting the capabilities of the Enterprise Service Bus for dynamic message routing.

The remainder of this paper is organized as follows. In Section II we describe the concepts of Autonomic Computing (AC). In Section III we define dynamic computational workflows and explain the benefits of applying an AC paradigm to manage the complexity of the

This work has been supported by the U.S. Department of Energy, under contract DE-FC26-06NT42755 and NSF Grant No. 826547

DOI: 10.5176_2010-2283_1.4.97

system while assuring the scientific fidelity of the results. In Section IV we discuss the concepts of the Service-Oriented Software Engineering (SOSE), including Enterprise Service Bus (ESB) and Enterprise Integration Patterns (EIP) and their potential for enabling AC, and in section V we present our implementation of an autonomic workflow. Finally, in Section VI we offer our conclusions.

## II. Autonomic Computing

Autonomic Computing (AC) concepts [3, 4] have been effectively used to manage enterprise systems and applications; now they provide a promising approach to address the challenges of complexity management. Analogous to the human body, where the autonomic nervous system responds to stimuli by adapting the body to its needs and to the environment without involving the conscience, AC-driven complexity management is achieved by creating self-managing environments capable of dynamically adapting to unpredictable changes using only high-level guidance or intervention from the users. Following this concept, each element of a computational system is managed by its own autonomic control loop, involving monitoring, analysis, planning, and execution (M-A-P-E, cf. [1]), realizing a set of predefined system policies. These individual control loops will then collaborate, i.e., communicate and negotiate with other autonomic managers which control other aspects of the computations.

Furthermore, as Parashar expressed it, "the autonomic approach mimics nature's way of managing the complexity: complex patterns emerge from the interaction of millions of organisms that organize themselves in an autonomous, adaptive way by following relatively simple behavioral rules. In order to apply this approach, the organization of computations over large complex systems, computations must be broken into small, self-contained chunks, each capable of expressing autonomous behavior in its interactions with other chunks" [4]. The goal of autonomic computing, then, is to manage complex computations via sets of predefined, simple rules that define the system's responses to failures and unpredictable changes in the computational environment, thus providing means for recovery from faults and/or adaptation of the system without direct human intervention.

## III. Computational Workflows

A computational workflow is a sequence of computational and data management tasks in a scientific application. Organizing the scientific analysis into a workflow significantly reduces the complexity of the application: the monolithic and thus difficult to maintain application is decomposed into simpler, independent modules (workflow nodes) focused on specific aspects of the problem at hand. Individual components can be reused for different applications (workflows), and the business logic of the overall application can be tuned and improved by reconfiguring the workflow, i.e., changing the sequence of tasks.

Our goal is to provide a workflow execution environment with the capability to recover from faults of the workflow components and consequently to prevent erroneous data from failed components from entering the final result set ("scientific fidelity") or crippling the business logic of the workflow. Furthermore, we envision the workflow execution environment as capable of autonomous "self-healing," that is, correcting non-fatal failures without human intervention.

The autonomic execution of a workflow is even more important in the case of dynamic workflows in which the sequence of the components changes unpredictably (e.g., is data driven), and the same component can be invoked many times. The multistep design optimization (MDO) is an example of a dynamic workflow.

### A. Multistep Design Optimization

Many complex engineering systems are more readily optimized when they are decomposed into a number of subsystems with partitioned design variables and separate objective functions and design constraints. Following the Analytical Target Cascading (ATC) approach [5, 6], the resulting workflow has a layered architecture of decomposed systems, as schematically shown in Figure 1. The hierarchy can be expanded to include several levels, each containing multiple elements. This hierarchical architecture, applicable to integrated product-material design, offers autonomy to each element to optimize its own objective function according to an element-specific set of constraints, which are, in turn, based upon either inputs from lower-level elements and design targets or demands imposed by corresponding upper-level elements. Because the number of design variables in each element represents a fraction of the total set, the dimensionality of each element optimization problem is reduced. Hierarchically decomposed systems are naturally suitable for parallel computing and decentralized optimization approaches, but they require a careful coordination strategy in the ensuing iterative solution process to ensure satisfaction of system-level design criteria

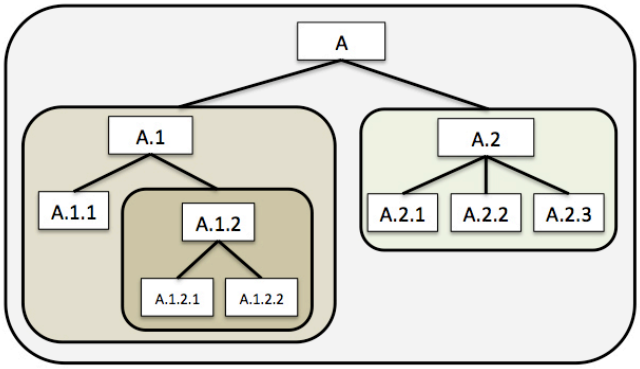


***Figure 1:*** *Idealized hierarchical workflow for multistep design optimization.*

and proper convergence to an optimum design.

### B. Idealized dynamic workflow

The details of ATC and its application for design optimization are beyond the scope of this paper. What is of interest here is the structure of the resulting dynamic workflow. The workflow comprises a number of nodes (cf., Figure 1), and each node implements the same pattern: given initial values, it performs an optimization of the subsystem by submitting a job to minimize its objective function. Depending on the results of the subsystem optimization, the children nodes are dispatched, or the results are returned to the parent node. This dependency on the optimization results at each level makes the overall computations dynamic: at the beginning of the process, it is unknown how many times each node will be invoked, and consequently, the sequence of job submissions is unpredictable.

ATC defines the rules for controlling the execution of the workflow, that is, the sequence of invoking workflow nodes and convergence criteria. However, these rules implicitly assume that all submitted jobs complete successfully and deliver trustworthy results. A failure of a single job may cripple the entire workflow, wasting all the results obtained before the fault occurred. Even worse, an unreliable result caused by an unreported failure may distort the end results.

### C. Job Execution Service

Since the core functionality of the workflow node is submitting a job, let us examine an example implementation of a Globus-based Job Execution Service (JES) [7], as shown in Figure 2. Given a job descriptor (a string in Resource Specification Language (RSL) [8]) as the service request argument, the service selects the target machine (e.g., site 1 or 2), performs data staging, and submits the job to the Globus Resource Allocation Manager (GRAM) [9] server at the selected site. If the submission succeeds, the job submission service enters the job id (returned by GRAM) to the job monitoring (JM) service, and responds with an acknowledgement. Otherwise, it responds with a job

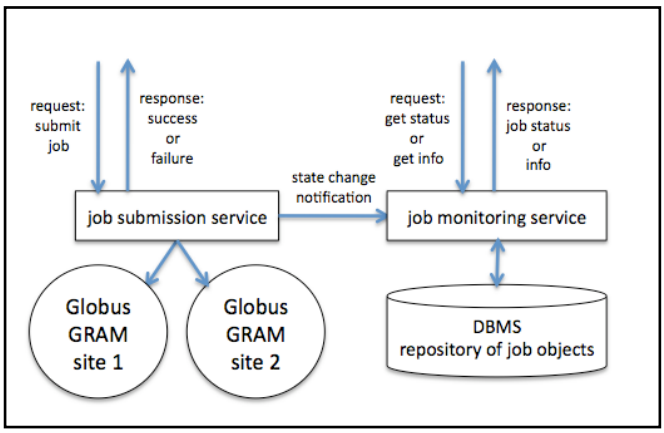


***Figure 2**: Job Execution Service*

submission failure message. All changes of the state of the job (pending, running, completed) reported by GRAM are forwarded to the JM service. The submitting client then polls for job status by sending requests to the JM service until the job is completed. At that moment, the client retrieves job information comprising of the actual location of the job stdout, stderr, and any other available output files.

### D. Failure modes

A job submitted through the JES may fail (i.e., no or unreliable results are produced) in many different ways. Following the patterns recognized in [10], we can group these failures into four categories or levels:

1. The service may not be responding to or reporting an internal error, that is, a service level failure.
2. The job submission may fail because of expired credentials, errors in RSL, shutdown of the target machine, or other specific job submission service level failure.
3. The job may crash (non-zero exit value) because of, for example, missing input data, insufficient memory, time limit, or other system level failure.
4. The job may complete with exit value=0 but still produce unreliable results, such as non-converged optimization or other application level failure.

Although demonstrated here for JES, this categorization is generic and can be applied to any type of service.

Many of these faults can be remedied programmatically. For example, in the non-responding service, a peer service can be invoked instead. Expired credentials can be refreshed; memory requirements or execution time limits can be tuned in a re-generated RSL; lack of convergence can be remedied by selecting another algorithm, changing the initial values, or modification of the constrains on the values of design variables.

Recovering from these failures could be incorporated into the workflow specification, but it would add unnecessary complexity to an already complex set of ATC rules. Furthermore, the domain expert who applies the ATC rules may not know or understand the failure modes and the remedies than could or should be applied, while the IT professional responsible for the deployment and maintenance of the services typically has little, if any, understanding of the ATC rules. It is thus desirable to manage the complexity of the ATC workflow (or any other computational workflow) by separating failure recovery from the business logic of the workflow, thereby designating fault recovery as a property of the execution environment and not of the workflow itself. This property, often referred to as self-healing, can be achieved by applying AC concepts.

### E. Autonomic execution of jobs

The complexity of computational workflow management due to unpredictable job failures can be addressed by treating jobs as managed resources. Following the AC approach, the job should be managed by its own autonomic control loop that would guarantee that the results generated

by the job meet criteria specified in predefined system policies. To achieve that, the JES must be augmented with additional functionality for assessing the quality of the results. To earn the qualification of being autonomic, the manger implementing the control loop to enforce the scientific fidelity of the results must be independent of the business logic defined in the workflow specification.

The taxonomies of failure modes help design monitors and analyzers of M-A-P-E autonomic managers, while the taxonomy of remedies allows design of the planners. Typically the planners would modify the service request (e.g., the job specification) and re-invoke the managed service (e.g., resubmit the job). These taxonomies will be necessarily open, as it is unreasonable to expect that all possible failure modes will be captured at the design time. Furthermore, the repertoire of remedies will grow as the knowledge of the system increases. Consequently, the design of the system must allow for adaptive runtime changes (defined by configuration files and/or policies) and learning.

The autonomic job manager envisioned here acts *reactively*: it responds to faults after they have actually happened. Such a manager should be complemented with *proactive* behavior: corrective actions taken before a predictable fault occurs (e.g., as in [11]). For example, the availability of the disk space could be monitored regularly (independently of whether a job is submitted or not), and if the available space is less than a predefined threshold value, some corrective action is taken so that when a job is submitted, it will not crash because of lack of disk space.

It follows that the AC paradigm requires adding a large number of new components: monitors, analyzers, planners, and executors. Therefore, if the system is not designed carefully, the complexity will move from the workflow's business logic to the execution environment, defeating one of our principal goals.

## IV. Service-Oriented Software Engineering

The Service-Oriented Computing (SOC) paradigm uses services to support the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications [12]. Services are autonomous, platform independent entities that can be described, published, and discovered. The visionary promise of SOC is that it is possible to easily assemble application components into a loosely coupled network of services that can create dynamic business processes and agile applications which span organizations and computing platforms [13].

### A. Enterprise Service Bus

The requirements to provide capable and manageable integration of services are coalescing into the concept of the *Enterprise Service Bus* (ESB) [14, 15], implementing Java Business Integration (JBI) [16] specification. An ESB is a software construct that provides fundamental services for complex architectures via an event-driven and standards-based messaging engine (the bus). With ESB, requestors and service providers are no longer interacting directly with each other; rather they exchange messages through the bus, and the messages can then be processed by mediations (e.g., message transformation, routing, monitoring). Mediations implement the integration and communication logic, and they are the means by which ESB can ensure that services interconnect successfully. As a result, the ESB acts as the intermediary layer between a portal server and the back-end data sources with which the data portal interacts [12].

### B. Self-managing of Service-Oriented Systems

During the last few years, the issue of self-management and support for adaptivity of service-oriented systems has attracted attention of many researchers [17-21]. Most of the proposed solutions to support this autonomic behavior place the service bus in the center of the architecture, taking advantage of dynamic routing features offered by most implementations of the bus.

For example, S-Cube [19] adopts a *publisher-subscriber* [22] pattern to manage the flow of messages in the bus. A central Service Adaptation and Monitoring (SAM) module subscribes to events fired by monitors of all managed resources. Based on the signature of the received event (context and runtime values) and adaptation strategies retrieved in real time from the Adaptation Manager, SAM automatically selects a suitable adaptation action and invokes it by firing an event (a one-way message) to be consumed by the adaptation gateway, which in turn, dynamically routes the message to a service capable of performing the corrective action. For the purposes of S-Cube, the adaptation strategy is an XML document implementing the router *slip pattern* [22], that is, it specifies the sequence of services to be invoked and message transformations needed in between.

The Ceylon autonomic system [11] exploits the flexibility of the *publisher-subscriber pattern* even further by implementing planners (in the M-A-P-E paradigm) capable of correlating independent but related events.

Many authors recognize the arising problem with developing such systems: heterogeneity of messages traveling through the bus and, associated with it, the increasing complexity of the dynamic routing. Multifactor-driven hierarchical routing (MDHR) [20] distinguishes three layers for message routing on an ESB: *the message layer* for standard ESB mechanisms for message delivery (content-based routing, itinerary-based routing, or static routing); *the application layer* that encapsulates legacy applications; and *the business layer* allowing for external mechanisms for message routing as defined by domain specific language of a business process. The virtualization of services supported by ESB is also exploited by the DRESR framework [21] to allow for dynamic changes in business and service processes.

The heterogeneity of messages and the resulting complexity of routers comes from different "types of service variability" [10] that require an adaptation of the system at one of four levels: (1) workflow composition (e.g., using

optional or alternative steps); (2) composition (e.g., alternative implementations to be bound at runtime); (3) interface variability (mismatch between actual and published service interfaces); and (4) logic variability (alternative business logic of a service). Handling the messages, which are carrying information about a system state change, an abnormality, or a failure at one of these levels, requires identifying a "signature" from a message and using it to select alternative services as defined in a registry and which are capable of modifying the workflow or service endpoints, applying a transformation, or changing a service configuration as needed.

The complexity of recognizing the message content needed to apply content-based routing seems to originate from the design feature that is common for the above-described implementations: the centralization of the adaptation control, leading to an unnecessary complexity of the autonomic environment. In this paper we propose an alternative approach, based on the foundations of AC. We propose a decomposition of the central complex decision maker, such as S-Cube's SAM, into a large number of small components implementing simple behavioral patterns, and use of the full power of a rich set of Enterprise Integration Patterns (EIP) [22] offered by ESB to integrate them into a dynamic, autonomic system. By mimicking biological systems, the inherent complexity of an autonomic system can be thereby reduced to a collection of easy to maintain and configure services, each following simple rules.

## V. Computational Workflow As a Managed Resource

Within the SOA paradigm, a workflow is a composite service, that is, a service that combines other services, where the 'constituent' services interact with each other through an exchange of messages.

A message traveling in the ESB is a Java object implementing javax.jbi.messaging.NormalizedMessage interface. This interface mandates, among other things, the message properties ("headers") and message content ("body"). A special case is a Fault message (javax.jbi.messaging.Fault interface that extends Normalized Message interface). A Fault message is created when the service cannot complete the processing of a request. It may happen for many reasons, such as missing or invalid data, insufficient resources, a bug in the implementation, or other unpredicted circumstances. This mechanism can be further exploited by introducing exceptions at the business level: if the result to be returned by the service does not satisfy requirements specified by a predefined policy, an exception is thrown. The content of the Fault message provides the details of the exception that triggered the service failure.

The Java objects representing the messages within the bus are converted to "wire-ready" messages (e.g., SOAP over HTTP) by ESB binding components when communicating with the external service requestor and provider (cf. Figure 4).

### *A. Service as a managed resource*

Catching exceptions by the service implementation itself is a form of service **monitoring**. The clear distinction between successful and fault messages can be used as a simple rule for content-based routing: unless the event message is of type Fault, the message is routed to the service specified in the routing slip (*Routing Slip* [22] pattern); otherwise, it is routed to an alternative endpoint (*Detour* [22] pattern). As a consequence, this simple router acts also as an **analyzer**, the second element of an autonomic manager. The intention here is straightforward: should a fault happen, the system makes an attempt to recover from it by applying a detour and, when the problem is resolved, the requestor gets a successful, trustworthy response without knowing that a corrective action has been autonomically performed. The detour results in forwarding the Fault message to a **planner**, that is, a dedicated service, which is capable of identifying the cause of the failure and of selecting one of a set of predefined but configurable corrective actions. The corrective actions are driven by a policy (e.g., articulated as XML documents) so that the planner service can translate the signature of the failure encoded in the content of the Fault message into a sequence of actions to be taken following the *routing slip* pattern. Since the planner must understand the signatures of failures, the functionality of the managed service and the planner are tightly correlated, and therefore each managed service should be associated with a corresponding planner. Should the planner fail to recognize the fault or devise a plan for corrective action (e.g., no policy defined), it throws an exception. In general, there is no reason to define a planner for the planner service; therefore, the router sends an "unrecoverable fault" message to the requestor: at this point, there is nothing that can be done to recover from the failure.

The planner should be a separate service because the monitor (i.e., the managed service itself) is capable of only identifying *what* is wrong, but, in general, does not have enough information to determine *why* the exception happened. For example, the service can easily recognize that the input data is invalid, but it is outside the service's scope to determine what steps need to be taken to correct the data. Furthermore, the determination of the corrective

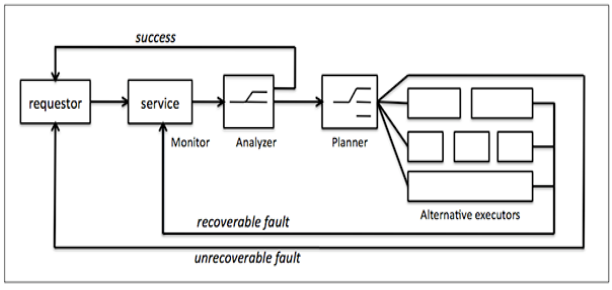


***Figure 3:*** *The concept of autonomic manager for a single service*

actions may require a correlation of information coming from several monitors.

The services defined in the routing slip serve as **executors** of the autonomic manager. The intent here is to remove the conditions that led to the fault of and then to re-invoke the managed service. For example, in the case of a job submission, invoking a sequence of services may be necessary to modify the job's RSL description and/or its input files, and then, to re-submit the job. Any already deployed service can be used as an executor, if its functionality happens to serve the purpose (e.g., RSL generator service); otherwise dedicated services must be developed and deployed. In addition, if applicable, a service acting as the executor of the autonomic manager may make an attempt to adapt the system to avoid the same type of faults in the future.

Finally, all actions need to be logged into the database (*message store* [22] pattern), the knowledge component of the autonomic manager. It is necessary to allow the end user to monitor the progress in real time (e.g., through an interactive GUI), and to indentify the sources of unrecoverable faults. Furthermore, the planners use the database to correlate responses from different services, such as monitors of the system state (e.g., is there enough disk space available?), or to break infinite loops or deadlocks if the sequence of the applied corrections does not converge.

To summarize, a service is managed by a M-A-P-E autonomic manager, schematically shown in Figure 3: should the service fail to complete successfully (the monitor functionality), the service response is detoured (the analysis functionality) to a planner service that determines the sequence of corrective actions to be preformed by executors. Once the conditions leading to the fault have been removed, the managed service is re-invoked. Should the planner or executors fail to recover from the fault, an "unrecoverable fault" message is returned to the service requestor.

We have realized this autonomic behavior using Apache ServceMix [23] implementation of the ESB. The requests from external requestors are received by a Binding Component. Binding components are standard JBI components that plug into NMR and provide transport independence to NMR and Service Engines (SE). The role of binding components is to isolate communication protocols from JBI containers so that Service Engines are completely decoupled from the communication infrastructure.

The routing decisions in our implementation of the standard org.apache.service.jbi.nmr.broker interface are based on three message properties: routing slip, return address and fault, following a simple algorithm shown in Figure 5. In the absence of a fault, the first element in the routing slip is resolved via registry to an endpoint of a Service Engine (e.g., JES). The fault messages are routed to the corresponding planner Service Engines with the endpoint defined in the service registry. If the routing slip is empty, or the endpoint of the planner cannot be resolved, the fault

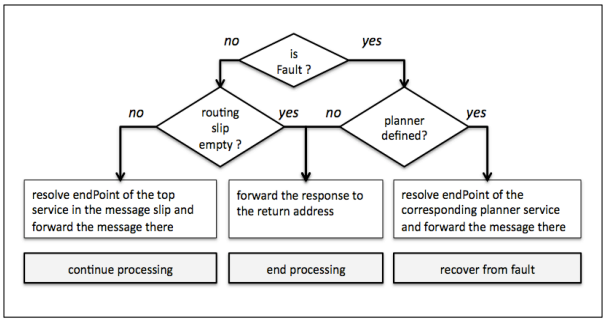


***Figure 4:*** *Routing algorithm*

message is returned to the requestor, using the return address embedded as the message property.

This implementation treats all services symmetrically, that is, the router is not aware of the business logic implemented by a service. In particular, it does not distinguish between the managed services and the various components of autonomic managers at different levels. The router only distinguishes between regular and fault messages, and it follows the routing slips created by invoked services and embedded as the message property. This approach reduces the complexity of the services, their managers, and the router to a set of simple rules.

Note that because of the symmetrical treatment of the services, the elements of the autonomic managers are also autonomically managed: if the event message generated by a planner or an executor is of type Fault, the router recognizes the failure and re-routes the message so that corrective actions can be taken. This feature is rarely discussed in literature.

### *B. Scientific fidelity of a service response*

A faultless completion of a service does not necessarily guarantee that the service's response satisfies criteria specified in a policy. An autonomic validation of the response must be performed as well. As an example, a job executed via JES may produce unreliable results (e.g., the minimization process has not converged). It would be a software engineering mistake to add the capability of detecting application-specific failures to the otherwise generic JES.

The validation of the results is therefore performed by a dedicated, validating service, which is automatically invoked after the service that produces the results completes. It can be easily achieved with ESB, exploiting its support for the virtualization of services. The original request (e.g., "run a job") is dynamically re-routed to a process manager service (implementing *process manger* [22] pattern) that inserts a routing slip to the message (in this example, run job, verify the exit value, and validate output). As a result, the sequence of services specified in the slip is autonomically executed: should the service's response not satisfy the criteria, the validating service fires a Fault message, which, in turn,

prompts the router to schedule a detour to the associated planner in order to initiate corrective action. Ultimately, the final response of the service to the original requestor is either trustworthy or it is an explicit fault message ("unrecoverable fault"), should no corrective action or other resolution be found.

### C. Workflow as a managed resource

Any workflow engine, e.g., a BPEL-based [24] one, will benefit from the autonomous execution of services, in particular, when the results produced by the services are autonomically verified. However, the complexity of dynamic workflows, especially those for which the determination of the subsequent actions can be defined by applying a set of rules (as is the workflow for hierarchical multistep design optimizations), can be reduced by applying the same autonomic approach as we have for a single service, that is, by treating the workflow as a managed resource. This creates a hierarchy of resource managers, with the workflow autonomic manager consuming "unrecoverable fault" messages generated by the individual services' autonomic managers.

Figure 5 shows the autonomic execution of a single node (node A.1 in Figure 1) of the idealized multistep optimization workflow. The execution begins with a planning activity based on the predefined rules (here, ATC). The planning is performed by a dedicated process manager service that updates the routing slip of the received message. There are three possible outcomes of this manager service: (1) the subsystem represented by A.1 node system is already optimized, and its optimized values are to be returned to its parent (here, node A); (2) the subsystem needs to be optimized by first optimizing its children (subsystems A.1.1 and A.1.2), followed by submitting a job to minimize objective function of subsystem A.1; (3) the optimization of the subsystem failed. The first case is handled by sending a message to the next service in the routing slip of the event that triggered this planning activity. The second case is processed by sending two messages, one with "process A.1.1" and "optimize A1," and the other with "process A.1.2" and "optimize A1" added to the top of the routing slip. The last case results in sending a Fault message, which triggers an autonomic recovery attempt. In each case, one or more messages are sent to the bus, and the router delivers them to the recipient following the simple algorithm shown in Figure 4.

Services labeled "process A.1.1" and "process A.1.2" are nodes in the workflow, and they are implemented in the same way as discussed for node A.1 (recursion). The "optimize A1" is actually a composite service: it aggregates (through updating of a routing slip) services for the creation of the job descriptor, the job input files, and job execution, which was discussed in detail above. Each of these services may fail, which results in sending a fault message that is appropriately routed for autonomic recovery (for clarity, this is not shown in Figure 5). The "optimize A1" is trigged by two independent events: successful completion of either "optimize A.1.1" or "optimize A.1.2" service (*aggregator* [22] pattern). In our implementation, the message store was used to correlate the events. When one of the children nodes sends the message, the store is searched for the message from the other child. If it is not found, the service exits without sending any events. For scientific fidelity, it is paramount that the messages from the children nodes are delivered to the "Optimize A.1" if, and only if, their results are trustworthy. Similarly, the decision whether or not subsystem A.1 has been optimized is based on the trustworthy result of "optimize A.1."

The system is easy to implement and can be deployed gradually, in small steps. The critical first step is to implement the ESB router capable of routing messages according to the routing slip embedded in the messages and of detouring the fault messages. Initially, this custom router preserves the original functionally of the system; for example, a fault the message is routed to the requestor unchanged since no corresponding planner is defined in the registry. Then the autonomic behavior can be added by deploying planners and process manager services, one by one, as experience with detecting failures and devising recovery procedures is accumulated. Furthermore, the self-healing property of the system can be progressively enhanced by adding "stand-alone" monitors which add to the message store updates on the status of other system resources that influence the reliability of the system.

It is important that the implementation of the services accommodate processing of policy documents that define the criteria for the determination of trustworthiness of results and specify the corrective actions. Following these guidelines enables updates of the policies at runtime that result in behavioral changes of the system, leading to a truly adaptive autonomic system.

## VI. SUMMARY

In this paper we have described an autonomic environment for execution computational workflows. This environment not only makes the best effort to recover from faults, it also guarantees the scientific fidelity of the results, in particular, that the final outcome of complex computations are not distorted by erroneous information

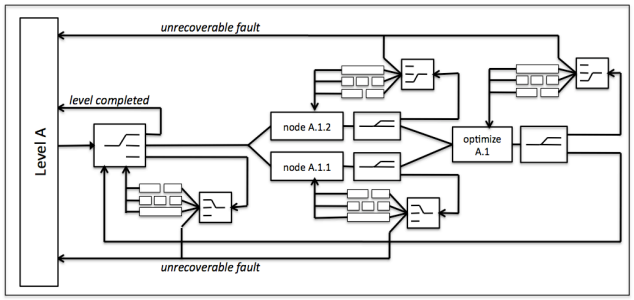


***Figure 5:*** *Autonomic execution of a multistep optimization workflow node*

resulting from unreported system- and/or application-level failures of the workflow and/or its components.

The autonomic behavior has been achieved by harnessing Service-Oriented Software Engineering, most notably, by employing the Enterprise Service Bus. By defining a set of very simple rules that apply to autonomous, loosely coupled services, we have generated a very complex autonomic behavior involving iterative, and possibly recursive, sequences of service invocations, thus mimicking biological systems. Scientific fidelity is achieved by enforcing service responses that meet criteria specified by configurable policies.

Failures to meet the criteria are caught as exceptions that result in firing fault messages. The custom ESB router, without any knowledge of the workflow's business logic of the workflow, detours all fault messages to specialized services that, based on the signature of the fault, plan corrective actions through inserting routing slips to messages. Those multiple planner services are independent of each other, each addressing specific problems and making decisions based on the policies defined in the configuration files that can be modified (adapted) at run time.

## REFERENCES


[1] Jia Yu and Rajkumar Buyya, A Taxonomy of Scitific Workflow Systems for Grid Computing, Special Issue on Scientific Workflows, SIGMOD Record, ACM press, Volume 34, Number 3, 2005.

[2] E. Deelman and Y. Gil. "Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges," Proceedings of the Workshop on Scientific Workflows and Business Workflow Standards in e-Science, The Second IEEE International Conference on e-Science and Grid Computing, Amsterdam, The Netherlands, December 4-6, 2006

[3] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing," Computer 36, 1 (2003), pp. 41-50.

[4] M. Parashar, "Autonomic Grid Computing: Concepts, Requirements, and Infrastructure," in "*Autonomic Computing*", M. Parashar, S. Harriri, (Eds), CRC Press 2007

[5] E.H. Miller, N.F. Michelena, M.K. Kim, and P.Y. Papalambros, "A System Partitioning and Optimization Approach to Target Cascading," Proceedings of the 12th International Conference on Engineering Design, Munich, Germany, 1999.

[6] M.K. Kim, N.F. Michelena, P.Y. Papalambros. P.Y., and T. Jiang, "Target Cascading in Optimal System Design," *Journal of Mechanical Design*, Vol. 125, pp. 474-480 2003.

[7] T. Haupt, A. Voruganti, A. Kalyanasundaram, and I. Zhuk. 2006. Grid-Based System for Product Design Optimization. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing* (E-SCIENCE '06). IEEE Computer Society, Washington, DC, USA, 46-52.

[8] Globus Toolkit 2.4, Resource Specification Language (RSL): http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1.html

[9] Globus Toolkit 2.4, Globus Resource Allocation Manager (GRAM): http://www.globus.org/toolkit/docs/2.4/gram/

[10] H.J. La, J. S. Bae, S.H. Chang, S. D. Kim, "Practical Methods for Adapting Services Using Enterprise Service Bus," ICWE 2007, LNCS 4607 (L. Baresi, P. Fratenalli, G.J. Houben, eds.), pp. 53-58, 2007

[11] Y. Maurel, A. Diaconescu, P. Lalanda, "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," in *Proceedings of the 2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems* (EASE '10). IEEE Computer Society, 2010, pp 3-11.

[12] M. Papazoglou, P. Traverso, S. Dustar, F. Leymann, "Service Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, Vol. 40 (2007), Issue 11, p. 38

[13] Leymann, F., "The (Sevice) Bus: Service Penertrate Everyday Life," 3rd Intl. Conf. on Service Oriented Computing ISCOC'05, Amsterdam, the Netherlands, Dec. 13-16, 2005, LNCS 3826 Springler-Verlag Berlin Heidelberg 2005

[14] D. Chappel, "Enterprise Service Bus: Theory and Practice," O'Reilly Media, 2004

[15] F. Leymann, "Combining Web Services and the Grid: Towards Adaptive Enterprise Applications," Proc. CAiSE/ASMEA'05, Porto, Portugal, June 2005

[16] Java Community Process, JSR 208 "Java Business Integration," http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html

[17] B.A. Christudas, "Service Oriented Java Business Integration: Enterprise Service Bus integration solutions for Java Developers," Packt Publishing, 2008.

[18] P. Martinez-Julia, D.R. Lopez, and A.F. Gomez-Skarmeta. 2010. The GEMBus Framework and Its Autonomic Computing Services. In *Proceedings of the 2010 10th IEEE/IPSJ International Symposium on Applications and the Internet* (SAINT '10). IEEE Computer Society, Washington, DC, USA, pp. 285-288

[19] L. Gonzalez and R. Ruggia. 2010. Towards dynamic adaptation within an ESB-based service infrastructure layer. In Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond (MONA '10). ACM, New York, NY, USA, pp. 40-47.

[20] X. Mi, X. Tang, X. Yuan, D. Chen, and X. Luo. 2009. Multifactor-Driven Hierarchical Routing on Enterprise Service Bus. In *Proceedings of the International Conference on Web Information Systems and Mining* (WISM '09), Wenyin Liu, Xiangfeng Luo, Fu Lee Wang, and Jingsheng Lei (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 328-336.

[21] X. Bai, J.Xie, B. Chen, and S. Xiao. 2007. DRESR: Dynamic Routing in Enterprise Service Bus. In Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE '07), Hong Kong, pp. 528-531.

[22] G. Hohpe, B. Woolf, "Enterprise Integration Patterns," Addison-Wesley, 2004.

[23] Apache ServiceMix, http://servicemix.apache.org/home.html

[24] Business Process Execution Language (BPEL), http://www.ibm.com/developerworks/library/specification/ws-bpel/



**Tomasz Haupt** got his Ph.D. in Physics from Jagiellonian University/INP in Poland (1985). Since 1990 he does research in Computer Science. His specialty is high-performance, distributed computing with the recent focus on Grid Computing, Cyberinfrastructure, Grid Portals, Service-Oriented Architectures and Autonomic Computing. Currently, he is a research professor at Mississippi State University leading the Cooperative Computing Group.



**Nitin Sukhija** received his Bachelor of Engineering in Computer Science from Institute of Technology and Management, India (2002), Master in Business Administration from San Diego State University (2009), and Master in Computer from National University, San Diego (2010). He is currently pursuing his PhD at Mississippi State His interests include autonomic computing and high performance computing for scientific and engineering applications.



**Igor Zhuk** graduated from Saratov University, Russia (1989). Since then he was involved in software development process as software developer, database and system administrator. Developed software for various business domains including billing systems, industrial automation, human-machine interface systems, word processors. His current occupation is Research Assistant in Mississippi State University where he develops distributed applications on the J2EE platform