

# A Platform for Service-Oriented Integration of Software Engineering Environments

Stefan BIFFL<sup>a,1</sup> and Alexander SCHATTEN<sup>a</sup>

<sup>a</sup>*Institute for Software Technology and Interactive Systems (ISIS)  
Technische Universität Wien, A-1040 Vienna, Austria  
{Stefan.Biffel, Alexander.Schatten}@tuwien.ac.at*

**Abstract.** In modern software application development, engineering systems and tools from several sources have to cooperate for building agile process environments. While there are approaches for the technical integration of component-based business software systems, there is only little work on the flexible and efficient integration of engineering tools and systems along the software life cycle. In this paper we introduce the concept of the “Engineering Service Bus” (EngSB) based on established Enterprise Service Bus concepts for business software systems. Based on real-world use cases from software engineering we show how the EngSB allows prototyping new variants of software engineering processes and derive research issues for further work.

**Keywords.** Systems integration, component-based systems.

## Introduction

In modern software application development, engineering systems and tools from several sources have to cooperate for building agile process environments. Software is in most cases no longer stand-alone and delivered as “shrink-wrapped package”, but embedded in larger contexts for building systems of systems: as part of a network, delivered as service in some *software as a service* (SAAS) context (i.e., not installed on client machines), or even as part of some infrastructure, in which hardware and software components have to cooperate seamlessly. This situation changes the game for software engineering (SE) as the embedding in larger contexts raises challenges in the following aspects.

**Need for better integration of engineering models.** Currently there is a rather weak integration between SE models and non-SE models that provide crucial requirements and design elements to software development, e.g., models of architects and mechanical engineers. As the correct function of the product depends on the systematic checking and updating across all relevant models, better model integration is a key to achieving better software and system quality more efficiently.

**Weak integration of engineering tools across domains.** Application life cycle models and tools are mostly focused on supporting specific engineering roles. However, the integration between tools of engineers beyond domain boundaries (other engineer-

---

<sup>1</sup> Corresponding Author.

ing disciplines, project management, and business roles) is fragile, inflexible, or involves tiresome and error-prone human chores.

**Frequent deployment cycles.** In many domains, e.g., in SAAS environments, release cycles occur frequently, up to several deployments a day. Frequent deployment cycles require tight integration between deployed instances and SE tools [4].

**Multi-source tool integration.** Each project has its specific requirements for the SE environment, which are rarely well supported by a single-vendor tool set, in particular, if organizational units cooperate, which already have made large investments into different tool landscapes. For building component-based systems the cooperation of systems and tools from a range of sources (typically component vendors and system integrators) is often necessary to combine best-of-class components and services [6].

In heterogeneous SE environments capabilities for the effective and efficient integration of engineering systems [17] and the semantic integration of engineering knowledge [1] are key enablers for engineering process automation and advanced quality management. Service-oriented engineering approaches promise solutions for effectively and efficiently integrating the tools and systems for software development, operation, and maintenance. While there are approaches for the technical integration of component-based business software systems, there is only little work on the flexible and efficient integration of engineering tools and systems along the software life cycle.

Certain application life cycle integration tools are already today part of SE best-practice like *continuous integration* (CI) tools that aim at integrating components along the processes of build automation, testing, reporting, and deployment on regular basis [11]. However, many of the available integration tools are rather monolithic and hard to extend or integrate into a more complex tool landscape.

In this paper we introduce the concept of the *Engineering Service Bus* (EngSB) to bridge technical and semantic gaps between engineering models and tools for quality and process improvements for SE and engineering disciplines that interact with SE [4]. The EngSB approach applies proven concepts from the “Enterprise Service Bus” (ESB) approach in the business IT context [7] to software systems engineering. However, most ESB implementations make design assumptions like: services will always be online and resources (computing, network bandwidth, memory) are no major design concern. Unfortunately, these assumptions are not sustainable in distributed mobile SE environments. Thus the EngSB has to be designed more lightweight and support the following capabilities.

**Tools as components in SE processes.** As tools encapsulate engineering models and project data, a major goal is their integration into SE processes, which may run for hours up to days, e.g., when converging to a major release. Particular challenges come from the need to integrate (1) both backend and frontend (i.e., user-centered) tools, (2) mobile tools that may go offline and reappear, and (3) run-time environments to ease the use of run-time measurements in combination with design knowledge in engineering models.

**Domain-specific services.** The ESB works on the level of routing messages between service endpoints. While this is an important basis for flexible systems integration, we see the need to work towards domain-specific services, i.e., services on the level of SE processes. On this level we can identify a manageable number of tool types that occur in many SE projects. Providing services for tool types rather than for specific tools allows to keep the SE processes stable even if tool instances evolve.

**Flexible software process prototyping.** The vision is to efficiently set up the environment for a new project, which takes a combination of technically heterogeneous

tools and systems (i.e., tools use a variety of platforms, protocols, and data formats). In such an environment the flexible prototyping of software-relevant processes should be supported by enabling the simple combination of existing tool functionality and provide a platform for designing new components and advanced services based on their access to data in other tools, such as tracing of requirements to design and test elements. A core question for designing the interaction of systems beyond sending single messages is which conversation styles, e.g., event-driven, service-oriented, or process-driven, are best suited for a given process and how to design the integration infrastructure for implementing the chosen mix of conversation styles. Typical current tool integration strategies choose one communication strategy for all kinds of communication, although a mix may be better suited. In this paper, we will demonstrate how these three techniques have different advantages (and disadvantages) for SE processes and how combining their strengths can provide a powerful solution to a wide variety of integration issues that currently pose challenges in the integration of development tools.

Based on real-world use cases from SE in the context of the “continuous integration and test” process [11], an elaborate standard process that is implemented unnecessarily rigid in modern SE environments, we show how the EngSB allows prototyping new variants of SE processes and derive research issues for further work.

Major results were: Even initial stages of an EngSB implementation can bring significant benefits to a heterogeneous engineering environment as this integration is the basis for flexible SE process prototyping, better awareness in the team on relevant changes in the project environment, process data collection and analysis, and quality assurance. Advanced stages of an EngSB implementation facilitate a global view on tools and systems along the life cycle as basis for the optimization of the engineering processes.

The remainder of this work is structured as follows: Section 1 discusses related work on software and systems integration in software engineering. Section 2 introduces the EngSB concept. Section 3 illustrates use cases for the EngSB, discusses results from exploratory prototypes, and develops research issues; and Section 4 concludes and provides an overview on further work.

## 1. Systems Integration in Software Engineering

In this section we summarize related work on approaches to integrate technically heterogeneous systems in general SE and on concepts towards better integration of multi-vendor systems engineering environments. Technical systems integration provides mechanisms to exchange messages between systems that rely on different platforms, communication protocols and data formats.

**Issues from coupling tools and data in SE.** In current practice, the integration of tools and processes in SE follows a range of strategies, often script-based point-to-point integration, e.g., using batch files. Specific build tools like *Ant*, *Maven* or *Rake* are used as a basis for build automation [19] and more integrated solutions following the concept of continuous integration [10, 11] that enables short development cycles. However, integration of user-centered development tools, such as Integrated Development Environments (IDEs) like Eclipse, has to be designed with specific plug-ins for each system; for non-SE tools even plug-ins are hardly available.

Data integration is usually conducted with a server-based source code management (SCM) repository, such as *Subversion (SVN)* or *GIT/Mercurial*. A common data repository works well for a homogeneous group of developers but poses difficulties if several teams need to work together. Partly, because other repository systems, such as databases, might be in use and complicate keeping consistency, partly, because internal SCM systems in commercial software projects can not always be open to external partners due to security reasons and conflicting business interests.

Additionally, collaboration via common repositories often requires the same tool set shared by all engineers, e.g., the same UML modeling tools to ensure compatibility of the data formats. However, even within a fixed tool set certain tools may not easily share data. Therefore, a common repository will be used for certain types of data, e.g., source code, but most likely not for other (non-text based and not standardized) file and data formats, e.g., engineering models. Moreover, as mentioned above, coupling of tools and data is rather tight, which is not desirable for project scenarios that need to minimize the impact of tool changes.

An example has been explored with the service-and-repository-based approach in *Eclipse ALF*<sup>2</sup>, an open-source Application Lifecycle Management Framework concept to integrate multi-vendor SE tool sets with a mandatory central repository. However, the tight coupling between tool integration and data repository makes the concept hard to integrate into existing environments.

**Enterprise application integration concepts for SE.** In enterprise integration scenarios, where backend systems need to be integrated to flexibly support business processes like financial services [9], tools and standards have been established to solve similar, but not identical problems of technical integration: separation of business logic from technical integration logic to ease the independent evolution of business processes and technical solutions; building from decoupled components an overall testable and robust business process with required functions and qualities of service. Established technical integration approaches are (1) message-based middleware and (2) service-oriented integration using variants of the Enterprise Service Bus concept [7, 14].

**Message-oriented middleware (MoM)** uses technology-independent message formats to connect applications in a loosely coupled manner but requires low-level application coding that intertwines integration and application logic [7]. Communication is no longer based almost exclusively on synchronous request-response communication patterns (such as RPC/RMI) but the emphasis is on asynchronous events and messages [12].

**Services** in service-oriented environments are autonomous, platform-independent entities offering a well-defined interface for interacting with them without the need to know how they are implemented [8]. Services can be described, published, and discovered in a loosely coupled manner [2]. In this context, interoperability refers to the ability of a service to use information and/or functionality of another service by adhering to common standards [24].

**SE tool integration needs.** For tool integration Thomas and Neimeh [23] distinguish four aspects that have to be taken into consideration and partly go beyond typical enterprise integration scenarios: (1) Presentation integration: integration of different tools into a common user interface and integration of end-user applications (like modeling tools and IDEs). (2) Process integration: Definition of processes beyond individual tools and integration of several tools into a process. (3) Data integration: tools use

---

<sup>2</sup> <http://www.eclipse.org/alf/>

different data formats and semantics that may need translation. (4) Control integration: tools can use each other to perform specific tasks (see also [16]). In this work we focus on process and data integration aspects. Beyond these aspects of tool integration and important question is how to describe commonalities of tools in a family of frequently used types of tools in order to elicit an abstract description of tool functionality that eases the evolution and exchange of tool instances and versions.

**Approaches for message-based communication.** In our recent work [5] we considered tool integration based on message-based integration following three paradigms, which put the communication of individual messages in a larger context: (1) Event-based mechanisms including complex event processing tools and rule-engines [12, 13], (2) service-oriented request/reply patterns [3], and (3) process-driven patterns for activities that follow clear described (long-running) workflows [9]. The request/reply and process-oriented styles are top-down approaches, i.e., the caller more or less decides who is invoked. The event-driven approach on the other hand works rather bottom up, i.e., the sender does not necessarily know who the consumers of certain events will be and the potential consumers (or middleware rules) decide what events are interesting to them.

The event-based approach has the advantage to provide a high level of decoupling between the involved systems. This allows a high degree of flexibility if individual tools change or new scenarios have to be covered. The synchronous request/reply pattern is needed particularly for tools that work directly with a user; and process engines support long-running development processes based on, e.g., conventional SE processes. Typical current tool integration strategies choose one of these communication strategies for all kinds of communication, although a mix may be better suited depending on the specific situation.

**Basic concepts of the Enterprise Service Bus.** The Enterprise service bus (ESB) provides an integration backbone for enterprise application integration [7, 21]. The ESB allows the implementation of all three architectural styles for message-oriented communication and additionally opens up the opportunity to link the development tool chain with other IT systems (e.g., business systems) or external systems (e.g., development tools in other companies).

The ESB provides a distributed integration platform and a clear separation of business logic from integration logic. The ESB offers routing services to navigate the requests to the relevant service provider based on a routing path specification. Routing may be based on an itinerary, on content, and on conditions that are defined manually or dynamically [7]. An inherent concept within the ESB is the container model [20], which describes how a tool in a particular technology context interfaces with the technology-independent ESB. In the run-time implementation the so-called container makes the service's functionalities and non-functional properties public to the external world and establishes connectivity and message exchange patterns [7]. Further, the container provides support and facilities such as transactions, security, performance metrics, dynamic configuration, and services discovery [26]. In addition, the container performs data and protocol adaptation, and monitors the internal behavior and state of services. In many cases, the container also defines a component model and a component lifecycle model that allows deploying, configuring and maintaining components on the ESB.

**Limits of ESB systems for SE tool integration.** Typical commercial ESB systems target enterprise integration scenarios and are typically expensive heavy-weight

systems that cannot easily be bundled for deployment with individual solutions, e.g., for use in a laptop application during field work.

Additionally, enterprise integration scenarios often focus on the integration of (rather heavy-weight) backend or client/server systems. Interaction with users is usually delegated to either dedicated (web-) applications or the client parts of the backend systems. Integration of engineering systems is typically not considered on the ESB side but on the client side of (also heavy-weight) applications like UML modeling tools, IDEs, and ontology editors. These applications are often stand-alone systems that store data in files; and use proprietary data formats and functions. Moreover, user-centered desktop applications (as opposed to backend systems) are often not designed with integration capabilities in mind. Yet the integration of engineering tools should consider both backend applications and heavy weight client applications.

A further issue (that is hardly covered with current systems) is that desktop applications are usually not permanently online (again, as opposed to backend systems), hence synchronization features have to be considered that allow conflict resolution in case of asynchronous changes to common data structures due to offline work of specific clients.

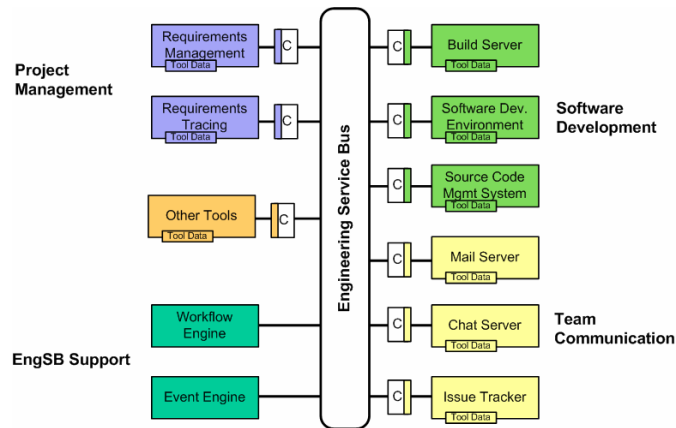
Based on the needs for tool integration in SE processes and the strengths and limitations of existing integration approaches, we introduce the concept for the Engineering Service Bus (EngSB), which focuses on the integration of tools and systems along the SE lifecycle.

## 2. The Engineering Service Bus Concept

Engineering tools in the SE life cycle can be viewed as components that already contribute to the engineering process independently by making the single engineer more productive and could support the engineering team even more effectively when working together seamlessly as part of the SE process.

**Goal of the EngSB.** The goal of the Engineering Service Bus (EngSB) is to integrate heterogeneous software engineering components similar to the enterprise service bus (ESB) in business IT while addressing the particular requirements of SE processes, systems, and tools [4, 15]:

- Capability to integrate a mix of user-centered and backend systems
- Mobile work stations that may go offline and reappear
- Flexible and efficient configuration of new project environments and SE processes
- Stable team process even if tool instances (and their data formats) change
- Stepwise migration from existing environments towards fully integrated EngSB platform

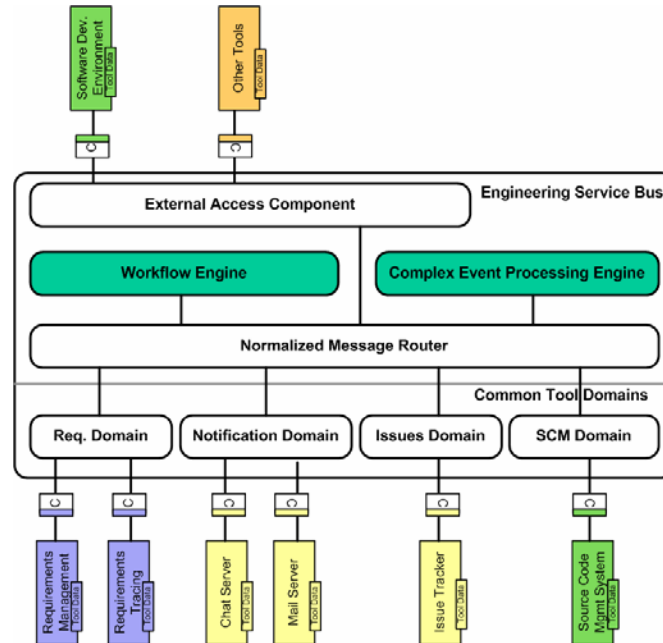


**Figure 1.** High-level view on tool connections with the EngSB.

**Elements in a SE environment scenario.** Figure 1 provides a high-level view on an illustrating EngSB scenario that consists of a collection of systems around software development. Typically a SE environment consists of several types of tools. *Project Management* includes tools to administrate a software project and the product requirements. *Software Development* comprises the well-known types of SE tools such as Software Development Environment, Source Code Management Systems, and Build Servers. *Team Communication* consists of tools for synchronous and asynchronous communication and notification in the team about relevant events such as changes to the system. These tools can generally support tracking process steps (e.g., with engineering tickets) that can not be sufficiently automated by direct tool integration. *EngSB support*: Once an EngSB platform has been established in a SE environment, components can be designed to build on the services provided by the connected SE tools. In order to define work steps beyond single process steps, a workflow engine and an event engine provide functions to describe rules for integrating the communication between tools on the engineering team level.

*Other Tools*: A SE environment can consist of many tools, so the EngSB must stay easily extensible. For example, UML modeling tools may become relevant to add to the EngSB for tasks like model validation.

A connector, which implements the container concept of the ESB, bridges between the local (technology-specific) application program interface (API) of a component, i.e., an engineering tool instance, and the (technology-neutral) EngSB. The connector defines how to map functionality between the bus and the tool instance. Each tool instance could be connected in its very own way; fortunately, a connector has to fulfill only the basic interface contract with the EngSB but has few other design constraints. A connector could work with *Java Message Service* (JMS), web service, or, if necessary, with scripts to integrate the tool instances. Thus the local technology and tool expertise can be applied without the need to learn more than the basic ESB interface technology concepts.



**Figure 2.** Detail view on selected EngSB architecture elements.

**Architecture elements of the EngSB.** Figure 2 extends Figure 1 to provide a more detailed view on the architecture elements of the EngSB. The EngSB core design is based on the concept of the *Java Business Integration* (JBI) framework [22]. JBI is a platform-independent standard for an ESB, described in a Java Community Process and documented in the Java Specification Request 208 (JSR-208). The specification itself is beyond the scope of this work. In this context it is sufficient to summarize the four most important points for choosing the JBI framework to support the EngSB prototype architecture.

1. *Normalized message format.* The JSR defines a normalized message format for the Normalized Message Router (NMR) allowing to send well-specified messages within the router.

2. *Lightweight service description.* All services available via the NMR are described using the Web Service Description Language (WSDL) and allow to clearly define interfaces and services. The model is very similar to the concept of Eclipse ALF but does not require heavyweight web service standards.

3. *Standard message patterns.* The eight most important, and well-known Message Exchange patterns (MEPs) in the WSDL 2.0 specification are supported, which allow both “send and forget” (in-only) messages and service calls similar to web services (in-out).

4. *Component life cycle definition.* The JSR further specifies a life cycle for components connected to the bus, which allows to load and unload components at run time, which is crucial to support mobile components.

**Integrating frontend and backend tools and systems.** Integration of engineering tools should consider backend applications and heavy weight client applications alike. This can be done in a series of steps: Initially, when proprietary applications are in



place that can not easily be extended, or the effort to write plug-ins seems too high for a “quick-win”, integration can be done by using notification systems that include the user. For instance, initially an engineering tool was not directly integrated into the engineering process due to an insufficient tool API. However, as a change in a particular part of the source code could affect an engineering model, the system can detect (1) the change in the module and (2) who the responsible persons are and then notify them relevant roles via e-mail or instant messenger while creating an engineering ticket in the issue tracker at the same time. Tickets have the advantage to provide space for links to relevant documentation for a task, which can be made understandable for humans and machines alike.

**Tool types provide common functions in the SE domain.** The requirement to integrate a variety of tools in a SE domain is reflected in the Application Lifecycle Management (ALM) systems on the market. ALM systems support connecting tools based on the knowledge they have about the applications that should be integrated. The EngSB has a similar goal but acknowledges, in contrast to most ALM systems, that many companies already have considerable investments into the tools they require for their processes. Therefore, the EngSB approach introduces the concept of tool types that provide interfaces for solving a common problem, independent of the specific tool instance used. This seems possible since different tools, developed to solve the same problem have, more or less, similar interfaces.

For example, the source code management (SCM) tools *Subversion* and *CVS* both provide similar functionality, which allows describing these tools as instances of the SCM tool domain. Figure 2 illustrates the SCM tool domain and other possible domains in the context of the EngSB.

We call the concept of tool types in this work “common tool domains”. This concept allows the EngSB to interact with a tool domain without knowing which specific tool instances are actually present. Note that tool domains do not implement tool instances but provide the abstract description of events and services, which have to be provided by concrete connectors of tool instances to the EngSB.

**Features of common tool domains.** The concept of “common tool domains” has important features beyond providing an abstract service and event interface to a range of specific components.

*Data Mapping.* Some domains require mapping data between the connectors and the EngSB. For example each issue tracker uses its own model to uniquely identify individual issues. If more than one issue tracker is used in an environment, there must be a mechanism to provide globally unique issue IDs and map them to the local issue IDs in each tracker instance.

*Data Enhancement.* Domains can enhance data from the tools to the EngSB or from the EngSB to the tool instances. For example a test domain can record, according the outgoing messages of a test tool instance, how often a specific test had been executed and enhance the message to the EngSB with statistical data about each test case. In the other direction process-related information can be linked to model elements, e.g., the process status of a model element, e.g., which elements need permission to change after customer approval.

*Functional enhancements.* This feature allows extending the functionality of tool instances similar to the interceptor pattern or aspects in aspect orientated programming. This feature allows profiling or adding additional logging and monitoring for tool instances.

**Pre-configuration of tool types.** In many industry and open source companies we have identified the need for a commercial off the shelf (COTS) solution for new technology on team level, i.e., a preference for convention over configuration by providing a ready-to-use infrastructure like the EngSB. Since the tool types in the SE process are quite stable, we expect to be able to describe most required domains and therefore be able to fulfill the requirement for a COTS solution.

**Integration of custom tools.** However, there will always be tools, which are unique for specific use cases and may not warrant a new tool domain. Further, some tools do not provide services or events to the EngSB but simply consume them. The EngSB addresses this challenge with the concept “External Access Component”. This component (1) provides all services of the EngSB domain as web services and (2) maps business events to JMS queues. This allows external tools to use services on the EngSB and to listen to events. An example for a consume-only tool in Figure 2 is a “Software Development Environment” (or Integrated Development Environment (IDE)). This does not mean that an IDE, such as Eclipse, does not provide services nor produces events. Actually, such tools are better seen as control centers for developers and not as active components. The “External Access Component” does also allow the EngSB to register to external events or services which could be integrated directly into the EngSB logic. This concept therefore allows to integrate tools which are not described by any tool domain at all.

In summary, the “External Access Component” allows to extend the EngSB with regular ESB and enterprise application integration (EAI) concepts. However, when working at this level most advantages of the EngSB are not available, i.e., the advantages of the tool domains and process design support.

**Designing processes and conversations.** Services and events received from tool instances via tool domains or external tools appended to the EngSB via the “External Access Component” have to be integrated at some point in order to support a larger process or as part of a conversation between tools [13]. The EngSB offers two mechanisms for this task.








A *Process Engine* (see Figure 2) that supports long-running processes. Although the EngSB is designed to handle SE rather than business processes, a Business Process Management (BPM) engine could be used for certain processes, like any other component on the EngSB. In this context, the EngSB process engine can be seen as a wrapper for a BPM engine converting messages from the NMR to a format understood by the wrapped BPM engine and back. This allows defining processes in a language similar to Business Process Execution Language (BPEL) [18] with the benefit that processes can be called by other processes (as usual in BPEL) and also by events.

*Complex Event Processing.* We added a component for Complex Event Processing (CEP) with the *CEP* component (see Figure 2), which supports event-driven patterns and executes short-running rules typical for event processing. This *CEP* engine works with rules describing which events should invoke which actions. Since rules depend only on events, but not on tool domains or other tools, it is possible to describe the default behavior and logic on SE domain level rather than on tool-specific level. Therefore, the CEP engine in combination with the process workflows could be seen as the internal knowledge base for a specific SE project instance on the engineering team level.

Processes for the EngSB can be described in the Business Process Modeling Notation (BPMN) language [25]. One of the reasons was that we require a standard to allow engineering process experts to describe their processes in a language they can under-

stand well without requiring deep technology knowledge. Further, the BPMN can easily be converted to most BPM implementations, which allows to quickly transfer BPMN process descriptions into the format of the internal BPM implementation. This allows separating concerns without need for additional graphical user interface tools. Management staff could design the engineering processes, which engineers can implement independently. Further, the process could easily be reverse engineered, to check whether the implementation was correct. Table 1 illustrates how BPMN can be used in the context of the EngSB: This table explains how each BPMN symbol is interpreted in the context of the EngSB BPM engine.

**Table 1.** Mapping of BPMN components to a BPM engine in the EngSB context.

Name	Symbol	Description
Pool		A so-called pool groups the activities of a tool domain. The pool is named after the tool domain.
Activity		Activities represent the operations within a domain.
Decision node		Decision nodes describe alternative flows within a tool domain annotated with Boolean conditions. More than one flow can be followed.
Process start event		Starts a process within a tool domain.
Process end event		Ends a process (or part of a process) within a tool domain.
Event		Shows the receiving (empty triangle) or sending (bold triangle) of an event. Receiving/sending an event starts/ends the process part.
Sequence flow		A sequence arrow connects the activities in a tool domain in chronological order.

### 3. Use cases and prototype “continuous integration and test”

In this section we illustrate expected benefits and limitations of the EngSB with the detailed use case "continuous integration and test" (CI&T) and an EngSB prototype. The prototype is completely implemented in Java, since there is a rich offer of Java open source products available, which allow prototyping a lightweight ESB-based solu-

tion that may be published as open source. We used the JBI implementation *Service-mix*<sup>3</sup>, which supports connecting to a variety of protocols.

**Use case CI&T.** The CI&T use case illustrates a key part in an iterative software development process: if part of a system or engineering model gets changed, the system has to be re-built and tested in order to identify defects early and to provide fast feedback on implementation progress to the project manager and the owners of the changed system parts.

This part is actually done by Continuous Integration (CI) servers (such as *Continuum*<sup>4</sup> and *Hudson*<sup>5</sup>, two non-commercial CI servers). CI is a very important part of the software engineering process, yet, a CI server is at first glance quite a simple thing: the CI server checks out the source code from a source repository (such as *Subversion*) and starts the build scripts, which are defined in the source code. For a typical Java project a *Maven*<sup>6</sup> or *Ant*<sup>7</sup> script will guide the CI process, which consists out of the following steps. 1. Build the source code, 2. Test the built source code, 3. Package the compiled source code. Every build result will be published by the CI on a project web homepage and if there are any errors, a notification mail gets sent to a configured list of recipients. This means that (1) the build process is fully defined by the build scripts and (2) this definition has to be repeated for each new project. It is possible to extend the process of, e.g., Hudson with plug-ins. However, writing tool-specific plug-ins will lock your work with the tool. If you have to change to another build server, all your tool-specific work will get invalid. Therefore, designing a flexible CI tool is a challenging experience.

**EngSB implementation of CI&T.** The CI&T use case can also be implemented in the EngSB to show how this application can support a state-of-the-art process but with better flexibility. Figure 3 shows the full process designed in BPMN. As the CI process is a long-running process the design can be executed by the “EngSB Workflow Engine”. Figure 3 shows the required domains that have to be designed: SCM, build system, test system, mail service, and reporting.

1. *SCM tool domain.* The SCM requires at least the possibility to register a check-in event and create a “commit-done” event. Before this event is thrown a check-out is done by the SCM domain and the path to the folder, other components have to work on is embedded in the event.

2. *Build tool domain.* The build tool is actually a *Maven* instance simply executing a *Maven* build script on the code base after receiving a “commit-done” event. This process has to work on a copy of the source, since it is not sure, whether another process is also working on this code concurrently. The build tool domain has to support at least a “build-successful” event in case of a successful build and a “build-failed” event if an error occurred. For further steps both event messages need to carry all required data about the build.

3. *Test tool domain.* The test tool domain requires only a service to start the tests and has to produce a “build-successful” event on success or a “build-failed” event on failure.

4. *Mail service tool domain.* The mail service domain consists of one service, “distribute report”, which is used to notify all registered persons.

---

<sup>3</sup> <http://servicemix.apache.org/>

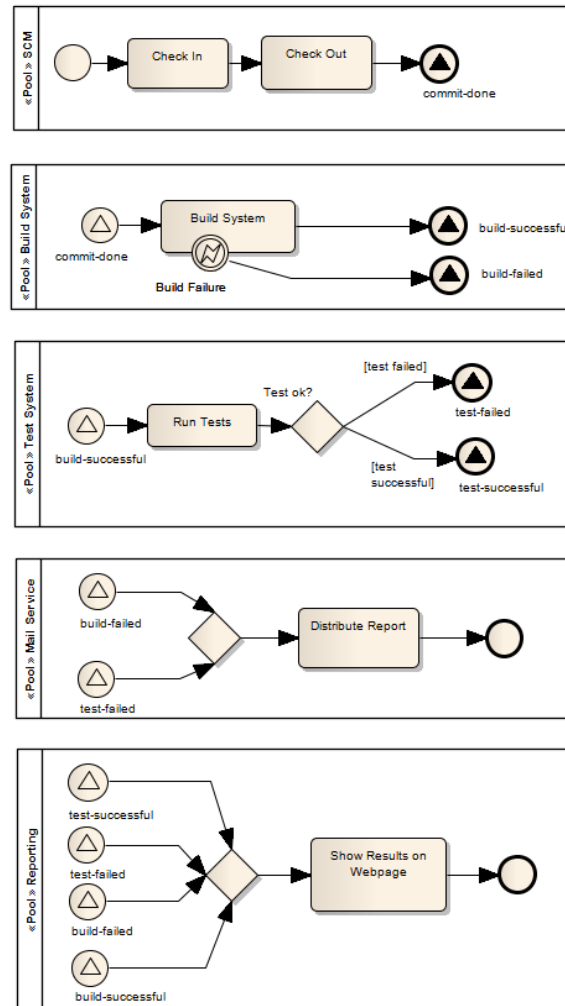
<sup>4</sup> <http://continuum.apache.org/>

<sup>5</sup> <https://hudson.dev.java.net/>

<sup>6</sup> <http://maven.apache.org/>

<sup>7</sup> <http://ant.apache.org/>

Reporting is not handled as an extra tool domain as it could be simply seen as an external application listening to the “External Access Component” for some events and providing them on a homepage.



**Figure 3.** CI process in BPMN design for in the EngSB.

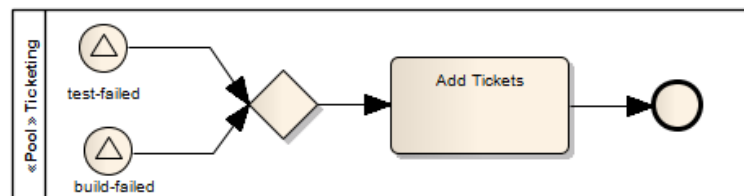
Finally the tool domains get wired together by a long-running process in the EngSB Workflow Engine. After a check-in has been done, the tool instance behind the SCM tool domain publishes a “commit-done” event. This event starts the CI workflow in the workflow engine. First the build service in the build tool domain is called. If no error occurs during this call, the test service is invoked by the process engine. If any error occurs during the build process, the process will stop. Notification on failure should be seen rather as a part of the SE domain than of the CI process itself. Therefore a CEP rule is added which calls the notification service if a “build-failed” or a “test-

failed” event occur. Both are shown in Figure 3 in the Mail Service Pool. Reporting is handled by an external application listen for all relevant events in the bus (“build-successful”, “build-failed”, “test-successful”, and “test-failed”) and present them to some user in the known way.

Up to this point we demonstrated the ability to reproduce the process as performed by CI servers like Continuum or Hudson. Now we can start to tackle some of the weaknesses of the traditional CI&T process.

**Designing flexibility into CI&T with the EngSB.** Actually changing the kind of notification, e.g., by switching from mail notification to chat notification requires editing the configuration files for each individual project, or editing each project directly at the server (which might not be appropriate as notification could be differ by context). In the EngSB this process is more flexible and yet simpler: by changing the tool instances behind a domain all projects can get similar benefits at once. Further, this approach also allows to simply add new kinds of notifications. If a chat notification is required beside a mail notification, the tool instance could be simply added to the process and would start working with the next service call to the notification tool domain.

Adding tools that were not envisioned by the developers of CI servers is a pain with current implementations. Adding such a tool requires writing additional plug-ins to a server to add the required logic. The problem with this is the reuse of such plug-ins in another context than the one planned. Further, changing the CI server instance can become very costly. The event-driven model in the EngSB makes such changes easy to handle. Figure 4 shows the workflow for an additional component, which creates issues in the case of build failures. We added an additional domain, the “issues tool domain” and connected the *Trac*<sup>8</sup> tool instance to it. Afterwards, a CEP rule has to be added which calls the “create-issue” service on the issues tool domain with the relevant data in the events.



**Figure 4.** CI workflow for ticketing reacts to "failed" events and creates issues.

Adding a statistics component to track the state over *several* projects at once is feature for project managers and quality personnel. Figure 5 illustrates the design of the statistics component similar to the reporting component: a separate application stores data gathered from events. This application is then connected to the JMS topics of the “External Access Component” and harvests all relevant test and build events required for a full statistical report.

<sup>8</sup> <http://trac.edgewall.org/>

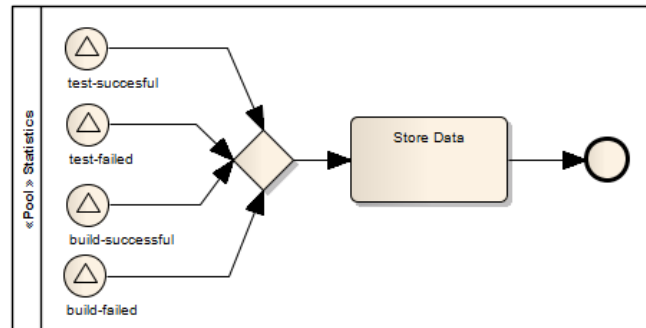


Figure 5. Statistics CI workflow reacts to all CI events on the EngSB.

**Use case conditional build failure.** Enriching the workflow of a standard CI server for measuring profiling data or for wrapping results of activities is quite a hard task. Some CI servers allow writing some kind of interceptors around their function calls, but this is not supported by all CI server implementations. (Automated) software testing sometimes creates the situation where tests have to be committed, which does not work if the tested code is simply not finished at the moment. For example, in case a developer finds an issue and commits a test to reproduce it. Or in the situation that requirements are defined in (integration) tests. Theoretically it is not a conflict to create a (snapshot) build now, since these tests never had been successful before. However, regular tests, which do not succeed, unnecessarily make a complete build process fail. We could simply extend the logic for all tool instances at tool domain level. The test tool domain intercepts all return values for the test component and can therefore store the results of the tests. If a “build-failed” result is returned by the tool connectors the tool domain analyzes if the failed tests were successful before. If they have not the domain creates a “build-successful” event with special warning flags and copies the result into this event.

**Lessons learned.** Implementing a series of scenarios with an initial EngSB prototype showed the feasibility to “reproduce” the state of the art of build automation and continuous integration. Further, we show how the EngSB allows prototyping new variants of software engineering processes more open, flexibly, and transparent than standard CI tools. More elaborate tool integration scenarios will include backend tools as well as tools that interact with end users. The actual process can be easily adapted and integration of external tools becomes easier and more flexible. Some key benefits reflected in the EngSB implementation for this use case are:

*Lightweight EngSB implementation.* The current EngSB prototype runs well in resource-constrained environments such as a laptop.

*Tools as components.* Treating tools as components on the EngSB worked well, in particular, as most tools were backend tools. The events that frontend tools send are also easy to integrate. A limitation comes from legacy frontend systems that do not provide a sufficiently mature API to deliver events to them and thus need a work around, e.g., with engineering tickets that notify the user of events that need human interaction.

*Tool domains simplify changing tool instances.* Because of the tool domain concept used by the EngSB we were able to simply replace any tool within a domain with another tool from that domain, e.g., the mail notification with a chat server notification

instance or one specific issue tracker with another, or even use different issue trackers at the same time for different projects or aspects of a project. Furthermore we could also extend the notification domain as well with a mail server as a chat server.

*Flexible process definition with BPMN and CEP rules.* CI&T standard tools are often designed with a fixed workflow, which may not be usable for a specific situation or SE domain. The EngSB allows connecting tools according to the workflow required rather than a fixed predefined workflow. BPMN process designs and rules worked well to capture expert knowledge and organizational culture explicitly as foundation for process automation.

*Flexible workflow extension.* The use case presents how easy it is to extend an otherwise very rigid process with additional tools adding error handling and statistic capabilities to the CI process.

*Flexible tool instance logic extension.* All tool instances of a specific tool domain can be extended with additional logic at once as shown in the use case. This is comparable to the interception pattern, i.e., logic on the bus could make modification to messages or data within a specific domain, hence add functionality before or after a specific tool is invoked.

*Limitations.* Major limitation of the EngSB is the added complexity to the tool environment as a new middleware layer that needs configuration and administration, which is justified for sufficiently complex heterogeneous environments. However, most modern SE projects have such a kind of environment. Further, the development and adaptation of the core EngSB needs advanced skills, i.e., support is necessary to make the design of EngSB applications simple for typical skill levels in the target environments.

#### 4. Conclusion and Further Work

In this paper we introduced the concept of the “Engineering Service Bus” (EngSB) that integrates components in office-like design environments based on the concept of the Enterprise Service Bus concept. Based on real-world use cases from SE we showed how the EngSB allows prototyping new variants of software engineering processes, and discussed strengths and limitations of the EngSB concept.

Even the initial prototype was able to demonstrate significant benefits in a heterogeneous SE environment.

- Standards-based design of process automation at engineering team level based on a common abstract infrastructure for communication between tools and systems
- Flexible and efficient configuration of SE processes
- Stable team process even if tool instances change

Advanced stages of an EngSB implementation can bring a global view on tools and systems in the software systems life cycle for optimization of the engineering and operation processes like analysis of cross-linked data from several sources; secure access to data in automation systems.

**Further work.** From the experience with the prototype in this paper we derive the following research issues.



*Configuration on engineering/domain level.* ESB and MoM infrastructures can be seen as a "low-level" infrastructure in the automation systems engineering domain. The goal is to provide a higher-level component description that allows tool vendors to easily connect their development tools and their run-time environments as well as other information systems to the EngSB. These partners should not have to deal with, relatively speaking, "low-level" protocols like Web services or "low-level" component models like Java Business Integration (JBI). Thus we will investigate methods and tools to map the rather high-level engineering component description of the EngSB more efficiently to "low-level" bus concepts like message topics, filters, routers or process engines.

*Gateway for collaborating EngSBs.* The EngSB approach seems well suited to integrate the systems and tools in a project or work group. An important issue is how to design a gateway that allows several work groups to connect their EngSBs to selectively share their engineering knowledge. The integration of mobile workers who may connect to a range of EngSBs over time and to other business systems raises issues regarding data synchronization and conflict management; security and privacy.

## Acknowledgments

We want to thank our industry partners and our research colleagues in the Engineering Service Bus team, in particular, Andreas Pieber, for their helpful comments and inspiring discussions.

## References

- [1] L. Aldred, W. v. d. Aalst, M. Dumas, and A. t. Hofstede, Understanding the Challenges in Getting Together: The Semantics of Decoupling in Middleware, BPM Center, Eindhoven, The Netherlands, 2006.
- [2] G. Alonso and F. Casati, Web Services: Springer, 2004.
- [3] A. Barros, M. Dumas, and A. t. Hofstede, Service Interaction Patterns: Towards a Reference Framework for Service-Based Business Process Interconnection, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 2005.
- [4] S. Biffl, A. Schatten, and A. Zoitl, Integration of Heterogeneous Engineering Environments for the Automation Systems Lifecycle, IEEE Industrial Informatics (IndIn) Conf., 2009 (to appear), (2009).
- [5] S. Biffl, A. Pieber, and A. Schatten, A Service-Oriented Prototype for Continuous Integration with an Engineering Service Bus, Technical Report TUW-QSE-2009-07, (2009)
- [6] K. K. Chan and T. A. Spedding, An integrated multi-dimensional process improvement methodology for manufacturing systems, Comput. Ind. Eng. 44 (2003), 673-693.
- [7] D. Chappell, Enterprise Service Bus - Theory in Practice: O'Reilly Media, 2004.
- [8] A. Dan and P. Narasimhan, Dependable Service-Oriented Computing, Internet Computing, IEEE 13 (2009), 11-15.
- [9] U. Dayal, M. Hsu, and R. Ladin, Business Process Coordination: State of the Art, Trends and Open Issues, 27th Very large databases conference, (2001).
- [10] P. Duvall, S. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk: Addison-Wesley, 2007.
- [11] M. Fowler, Patterns of Enterprise Application Architecture: Addison-Wesley Professional, 2002.
- [12] G. Hohpe, Programming without a call stack--Event-Driven Architectures, [www.enterpriseintegration-patterns.com/docs/EDA.pdf](http://www.enterpriseintegration-patterns.com/docs/EDA.pdf), (2006).
- [13] G. Hohpe, Workshop Report: Conversation Patterns, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
- [14] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions: Addison-Wesley Professional, 2003.

- [15] M. Huhns and M. P. Singh, Service-Oriented Computing: Key Concepts and Principles, Internet Computing, IEEE (2005), 75-81.
- [16] IEEE, IEEE Recommended Practice for CASE Tool Interconnection: Characterization of Interconnections. vol. IEEE Std 1175.2-2006, (2007), c1-36.
- [17] V. Issarny, M. Caporuscio, and N. Georgantas, A Perspective on the Future of Middleware-based Software Engineering, 2007 Future of Software Engineering, International Conference on Software Engineering, Washington, DC, (2007), 244-258.
- [18] D. Jordan and J. Evdemon, Web Services Business Process Execution Language Version 2.0: OASIS Standard, (2007).
- [19] M. Loukides and R. Romano, Maven the Definitive Guide: O'Reilly, 2008.
- [20] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, Service-Oriented Computing: State of the Art and Research Challenges, Computer 40 (2007), 38-45.
- [21] T. Rademakers and J. Dirksen, Open Source Enterprise Service Buses in Action: Manning Publication, 2008.
- [22] R. Ten-Hove and P. Walker, Java™ Business Integration (JBI) 1.0: Sun Microsystems, Inc. , 2005.
- [23] I. Thomas and B. A. Nejme, Definitions of tool integration for environments, Software, IEEE 9 (1992), 29-35.
- [24] F. B. Vernadat, Interoperable enterprise systems: Principles, concepts, and methods, Annual Reviews in Control 31 (2007), 137-145.
- [25] S. A. White, Business Process Modeling Notation (BPMN): Business Process Management Initiative, 2004.
- [26] J. Yin, H. Chen, S. Deng, Z. Wu, and C. Pu, A Dependable ESB Framework for Service Integration, Internet Computing, IEEE 13 (2009), 26-34.