Secure Scripting Based Composite Application Development: Framework, Architecture, and Implementation

Tom Dinkelaker Alisdair Johnstone Yuecel Karabulut Ike Nassi
SAP Research Center Palo Alto
SAP Labs, LLC
Palo Alto, USA
yuecel.karabulut@sap.com

***Abstract*—Dynamic scripting languages such as Ruby provide language features that enable developers to express their intent more rapidly and with fewer expressions. Organizations started using these languages in order to add enhancements to their existing applications or create composite applications. Current research has not yet addressed how security specification and enforcement can be done for scripting based application development. To fill this gap, we developed a framework for the design and facilitation of security. Our approach enables a business oriented application developer to add high-level security intentions to his business process model. The framework supports the automatic generation of security configuration and enforcement. As a proof-of-concept, we present an architecture and report the implementation status.**

Keywords: scripting, security, composite application.

I. INTRODUCTION

A composite application is an application making use of data and functions provided as Web services by service-oriented application platforms and existing packaged and custom-built applications. Composite applications combine these Web services into usage-centric processes and views, supported by its own business logic and specific user interfaces. In other words, composition enables prefabricated components to be reused by rearranging them in ever-evolving applications. Thus, composite applications enable business scenarios and/or user specific processes spanning multiple functional areas.

Application software vendors now offer Web service interfaces to their existing solutions and/or provide service-oriented application platforms, making cross-system composite applications much easier to develop. Some of these offerings also include tools for composite application development in languages like Java. While some organizations develop their composite applications by using these tools, other organizations, in particular small and medium companies, use dynamic scripting languages like Ruby and Python in order to add enhancements to their existing applications or create new composite applications. These languages allow building programs faster and more effectively in an agile environment than traditional strong-typed languages. Especially Ruby [28][29] is starting to take off. Its language capabilities like support for meta-programming, functional features and lambda expressions allow very effective programming also for the non-system programmer.

One of the main obstacles to building enterprise-quality (composite) applications in dynamic scripting languages is related to their missing security frameworks. Because dynamic scripting languages lack such language security features and application security models, script developers are forced either to ignore security or provide programmatic and implementation specific security development.

Our work was motivated by the following question: ”How can a business oriented application developer (a non-system programmer) easily integrate security features into his applications in development environments, where a) he uses a dynamic scripting language as the development language, b) the developer might not be security trained, c) the application needs to be developed in a very short development life-cycle, and d) the composite application has to enforce the business driven security policies of the company where the application is being developed?”

We present a security framework which enables a business oriented application developer to add high-level *security intentions* to his business process model. Intentions represent application security objectives such as business process authorization requirements, Web service Quality of Protection (QoP) requirements etc. The framework facilitates the automatic generation of security configuration and enforcement. Our approach associates high-level security intentions with extendible scripts that are provided as executable patterns. The security intentions and patterns are specified using internal security Domain Specific Languages (DSLs) [8].

The rest of the paper is structured as follows. Section II describes the problem domain and security requirements. We introduce the security framework in Section III. In Section IV, we show a specific architecture which discusses some framework blocks in detail. Section V summarizes the implementation status. Section VI is dedicated to related works. Finally, Section VII concludes the paper and discusses future work.

II. PROBLEM DOMAIN AND SECURITY REQUIREMENTS

A. Composite Applications & Service Composition

We observe three major reuse perspectives for a composite application: the reuse of data, business logic (including business processes) and user interfaces. In this paper we focus on the reuse of existing business logic. The notion of *service* is the fundamental abstraction for reuse of coarse grained business logic. We consider that there are three categories of services that a composite application development framework must contend, as illustrated in Figure 1: *backend* services exposed from backend enterprise applications (e.g., ERP); *external* B2B services provided by other organizations; and *local* services that are built into the composite as local components.

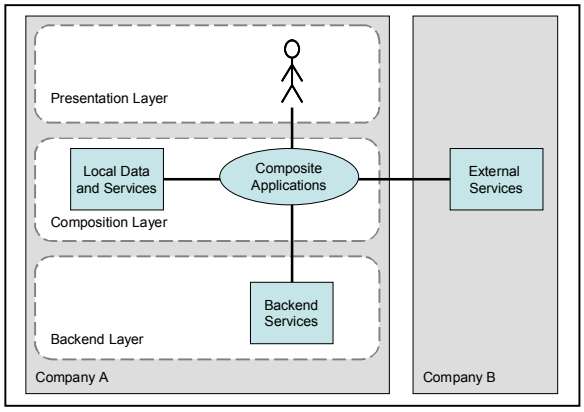

Figure 1. Composition Environment.

Local services need to be considered, since in many cases the developer might have to implement some business logic that is not captured by the existing services nor it can be captured by just orchestrating those services differently. Some local services may be built by composite application developer while others may be imported from other component providers and are installed and run within the composite.

The composite developer may ultimately wish for a composite or a local component he has built to be exposed as a Web service so that it may in turn be used by another composite.

B. Scripting Based Composite Application Development

In one of our related research projects we developed a scripting framework for composite application development. This framework provides an integrated modeling environment and scripting language that make it easier for a more business oriented developer to build new applications from existing or new building blocks (e.g., services). This framework follows a model-based scripting approach which supports the complete specification of composite applications in the form of one integrated model. Parts of that model describe the overall data model, orchestration of service calls, event management, user interface etc. as internal DSLs. One of the main goals of this scripting framework was to support the end-to-end development of composites by providing a family of DSLs. This means that all necessary logic and configuration to support the composite can be defined and deployed by one developer in the one toolset in as seamless a fashion as is possible. Thus, the developer is provided with a development and execution environment for which he doesn't need to use different tools and abstractions that are often used during the different software development phases.

Most general-purpose programming languages like Java and C# provide platform independence, but they still require the developer to render implementations of concepts in the problem domain using fine-grained constructs. This creates a gap between intent and implementation and causes additional complexity. In our scripting framework we provide a family of DSLs that offer abstractions closer to problem domains. Since the abstractions are more specific, the developer requires fewer constructs to describe the business logic.

The security framework we present in this paper extends the existing process scripting framework and demonstrates how security configuration can be easily handled by a business oriented developer.

C. Security Requirements

Security is one of the quality attributes that is of the greatest concern, especially when developing cross-organizational composite applications. We distinguish between two kinds security requirements: methodology related requirements and technical realization related requirements.

Methodology related requirements are following:

- From the usability perspective, the greatest challenge is to provide a simple security specification mechanism which should be in line with the simplicity of scripting based application development.
- Considering the complexity of computing environments on which composite applications are running, it is harder for developers to set up security properly. We need mechanisms which support a model-driven approach with the idea of generating security configurations out of high-level security intentions [30][2][3][4][6][7].
- From the secure software engineering perspective, we need to ensure that there is a close coupling between the business process model and its security requirements. This would create a consistent state across all changes in the business process and also bind the developer closer to the secure application development process [1][3][2].

Technical realization related requirements include the access control specification and enforcement, Quality of Protection (QoP) declaration and enforcement, and distributed policy management issues which might be required at different levels including the business process level, business process task level, and service level. These requirements are listed below:

- Specification and enforcement of authorizations and authorization constraints for individual business process tasks and business process, respectively.

- Specification and enforcement of QoP requirements for Web services. QoP requirements [21][22] define security/privacy requirements and technical security capabilities, similar to assertions and bindings in WS-SecurityPolicy [22] and policy intents in SCA Policy Framework [36].
- An automated policy configuration mechanism is required when interacting with backend services.
- Dynamic policy negotiation and policy enforcement are required when interacting with external services.
- Dynamic policy management is required to deal with policy changes during the operational phase in a cross-organizational composition scenario where multiple external service providers are involved, i.e., a change in the policy of a service being used by the composite is adapted without restarting the application.
- Standards compliant security services and policies are required in order to support interoperability in a distributed environment.
- Security APIs which provide an abstraction of low level Web services security standards.
- A unified usage of security mechanisms which provide enterprise level protection for all security aware applications.
- A unified design of business processes and security policies [1].
- Trust management infrastructure support for cross-organizational service interactions [24][13].

III. SECURITY DEVELOPMENT FRAMEWORK

We propose a framework for *designing* and *facilitating* security in scripting-based composite applications. In the following sections we discuss the framework building blocks.

A. Security Design

The framework aids the development of composite applications by defining certain development tasks to efficiently specify the security of a composite. Further, the framework also defines design-time protocols regarding which information and security artifacts the different participants must exchange with the other parties that are involved in the development process. Defining the dependencies between development tasks helps organize the design process.

Figure 2. shows the security design part of the framework. The overall process to model security in the framework is divided into 3 phases: a) the *definition phase* in which security objectives are identified, b) the *realization phase* in which means are provided in order to accomplish objectives, and c) the *declaration phase* in which security objectives for composite application or services are selected and attached as annotations.

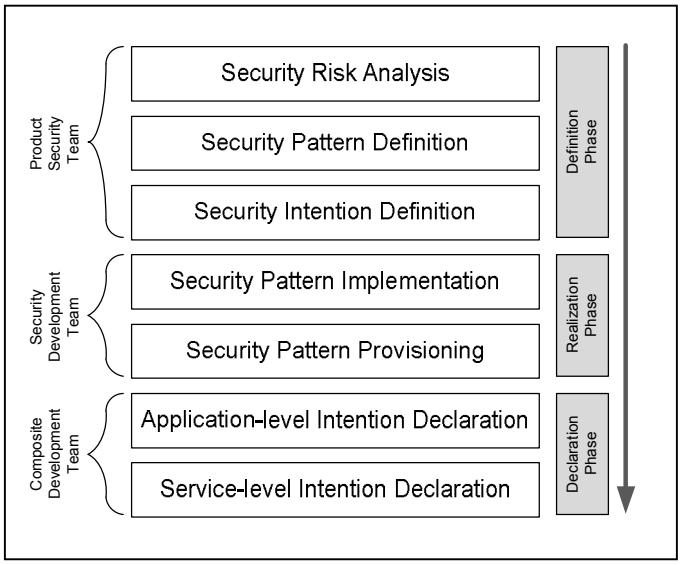

Figure 2. Security Design Framework.

In the definition phase, first, the product security team analyses the threats and identifies the associated risks in business scenarios and related business processes. A systematic analysis can be done by identifying service interaction patterns [34] in a service-oriented business application and performing threat analysis for individual service interaction patterns. In order to mitigate risks, the product security team proposes security solutions which can be cast in security patterns. Through definition of security patterns these solutions are made re-usable between different applications. This approach would also enable a unified usage of security mechanisms across all applications which need to be secured.

As a last step of the definition phase, the product security team defines a set of high-level security intentions which can be realized with a combination of security patterns. Security intention definition provides an intention ontology which aims to enable a unified definition of security objectives across all teams in the application development life-cycle.

In the realization phase, the security developer provides implementations for the patterns. Security pattern definition and implementation might utilize existing proposals [15]. However, domain-independent patterns often must be bound to a specific context. When re-using domain-independent patterns, the security development team follows company-specific rules to adapt their implementations.

The implemented patterns are made available through a pattern repository. This is supported by a security pattern provisioning process.

In the declaration phase, composite application developers declare security intentions to be followed by the application. Application-level intention declarations are used to capture the security requirements of the application. These declarations define the intentions applied by the composite in order to make interactions with the constituent parts (e.g., local components, process tasks, external Web services) of the composite secure. For example, when interacting with an external Web service,

intentions may specify the security requirement to secure all B2B interactions.

As stated above, the composite developer may wish for a composite or a local component he has built to be exposed as a service. He may want to specify QoP requirements by simply adding security intentions to his composite and local component before exposing them as services. This step is supported by the service-level intention declaration activity.

B. *Security Facilitation*

Enabling security for applications in the context of cross-organizational composites is supported by the framework through providing a set of runtime security protocols and pre-built security services.

Figure 3. presents the main protocols used to ensure the safe execution of composite applications. Protocols in the *start-up phase* ensure the basis for the protocols used in the *enforcement phase.*

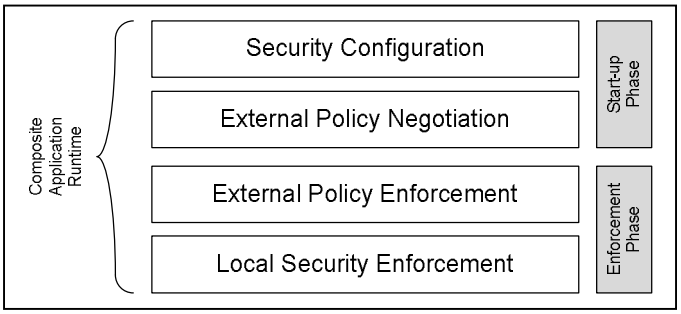

Figure 3. Security Facilitation Framework.

Security configuration happens before executing the application. The assigned application-level intentions assigned must be loaded and set up internally for runtime enforcement. When interacting with backend services, the corresponding security configuration must be set up. In order to execute the services on the backend successfully, sufficient authorization permissions must be in place. This requires a policy update protocol which includes generation of authorization policies and insertion of missing policies into the backend policy base. When interacting with external services, the required trust establishment can be achieved by exchanging authentication and authorization attributes, as discussed in WS-Federation [24]. Further, if necessary, trust negotiation can be done as proposed by [23].

For *local security enforcement*, the framework provides an access control mechanism to regulate access to local services and objects.

External policy negotiation is necessary when composites use external services in other security domains. A composite application (e.g., as service consumer) and an external service (e.g., as service provider) may define their individual security requirements (constraints) and security capabilities, with respect to token types, cryptographic algorithms and mechanisms used. Before engaging an interaction, both parties need to come to an agreement which specifies a common policy. This requires a policy negotiation process which supports merging of policies from two sources [22].

External policy enforcement takes care of enforcing the common policy for each interaction between both parties. This requires the modification of the exchanged messages, e.g., adding a security token to the message

C. *Evaluation*

We see our framework as an initial step towards developing secure scripting applications. The framework may need to be extended to add other important trust, security and contract management features which have been intensively studied in the TrustCoM project [13]. We didn't address the compliance related issues. The work in [16] can be adopted to address regulatory requirements as sets of compliance rules and their transformations to concrete policies. We didn't provide an in-depth study how to perform risk analysis and how to identify patterns and declare meaningful intents based on the analysis results. The elaboration of this topic is out of scope of this paper. For this purpose, the security quality requirements engineering methodology (SQUARE) [14] may be adopted. See [15] for a detailed discussion on security patterns.

IV. ARCHITECTURE

In Section III we presented a general framework for secure scripting based composite application development. In this section, we show a specific architecture which exemplifies selected building blocks of the framework.

The architecture in our view faces especially two design challenges: (1) providing a family of domain-specific languages that supports the efficient specification of application security policies and (2) runtime components for the enforcement and management of these policies. In the following sections we present solutions that address these challenges.

A. *Policy Specification*

The policies are specified by attaching intentions to the business process script. For the sake of understanding, we give an example of a concrete business process specification.

The business scripting language is designed to efficiently define the functional parts of composite applications. It is used to define a process, which consists of several *tasks* that may in turn include activities. Tasks may use local services, store local data in *variables*, and invoke external Web services or backend systems.

```
01 process Shipment
02
03 #Security Annotations
04
05   enforce B2BConfidentiality and B2BIntegrity
06   expose B2BConfidentiality
07   assign roles [manager] to select_carrier
08   constraint select_carrier before book_carrier
09
10 #Process Specification
11
12   variables
13     carriers as List
14     rates as Map
15     selected_carrier as Service
16     …
17
18   sequence
19     get_carriers => get_rate =>
20       select_carrier => book_carrier.
21
22   task get_carriers do
23     carriers = registry.get_services("carrier")
24   end
25
26   task get_rate do
27     carriers.collect { |carrier|
28       rates << {carrier => carrier.get_rate()}
29     }
30   end
31
32   human task select_carrier do
33     task_form.selection = rates
34     …
35     selected_carrier = task_form.result
36   end
37
38   task book_carrier do
39     selected_carrier.book_shipment()
40   end
41 end
```

Figure 4. Scripting-based Process Definition Language

Figure 4. depicts a simplified shipment process for selecting and booking a qualified carrier from a set of potential carriers. The selection is done based on the rates sent by carriers for a given shipment request. The process consists of a sequence of four tasks defined in lines 18-20. The "get_carriers" task selects a set of carriers from the carrier registry after checking their qualifications based on the shipment request details. During the "get_rate" task each carrier gets a request for quote (RFQ) and after evaluating the request, each carrier sends an offer back. The human task "select_carrier" implements the functionality to perform a manual selection of a carrier by a human user. Finally, the "book_carrier" task performs the booking of a selected carrier.

1) Security Intention Delaration

Security objectives are expressed together with the process specification in the form of *security intentions*. Figure 5. shows the conceptual model used to specify security in the process model. The definition phase (see Figure 2.) has provided an ontology of named intentions and the realization phase has provided a repository of patterns; each intention is associated with the pattern that implements the enforcement of the intention.

Intentions are used by declaring annotations in the process model, i.e., a process may declare a composed set of intentions to be enforced by using an annotation attached to the process script. For the sake of simple exposition, we only consider interaction annotations over atomic intentions which are composed to an *intention expression* together by using a logical conjunction. We consider three types of annotations:

a) ServiceInteractionAnnotation

A *ServiceInteractionAnnotation* is used to declare the external enforcement policies when using Web services. E.g., when messages are sent out they should be encrypted. This annotation has two sub-types: (1) *EnforceAnnotation* and (2) *ExposeAnnotation*.

The EnforceAnnotation **enforce** <service usage intention expression> statement declares a policy for interactions with Web services that are used by the composite. For example in Figure 4. line 5, we declare **enforce** B2BConfidentiality **and** B2BIntegrity (i.e., SOAP messages should be encrypted and signed). Therefore, when executing one of the tasks "get_rate" or "book_carrier" that call Web services, the SOAP messages sent will be encrypted and signed.

An ExposeAnnotation is attached by using the **expose** <service usage intention expression> statement. This expression declares policies when exposing the composite or a local component as a Web service. For example in line 6, the composite is exposed as a Web service which requires encrypted communication with any service consumer which may invoke the exposed service.

b) AssignAnnotation

An AssignAnnotation is attached by using the **assign** <role assignment intention expression> statement. This expression specifies which roles are allowed to execute the given tasks. For example in line 7 with the statement **assign roles** [manager] **to** select_carrier, the developer declares his intent that the task "select_carrier" is allowed to be executed by users that possess the role "manager".

c) ConstraintAnnotation

A ConstraintAnnotation is attached by using the **constraint** <execution order intention expression> statement. For example in line 8, we declare **constraint** select_carrier **before** book_carrier. This means that the task book_carrier must only be executed after task select_carrier is completed, i.e., the manager has selected a carrier. Additional constraint types such as separation of duty, binding of duty, and role seniority can be added, as discussed in [10].

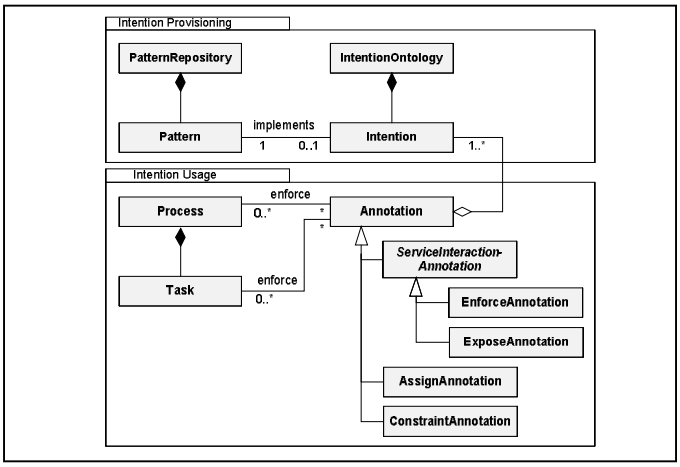

Figure 5. Policy specification on the process model.

2) Security Pattern Specification

The framework follows a pattern-oriented approach to implement security requirements. We associate security intentions with patterns. This means the enforcement of an intention is implemented by the associated pattern. Associating intentions with patterns is similar to associating security objectives with patterns [26]. When executing an application, the container (see Section IV) finds the corresponding pattern for a particular security intention and then follows the pattern implementation to secure the application.

In our approach, *security patterns* [15] are used to provide the technical details for the enforcement of an intention, i.e., how a certain security concern must be enforced. These patterns are provided as generic security components written by the container provider or by some other party as part of a *pattern library*, which is delivered with the infrastructure. If application-specific intentions are not covered, the pre-fabricated pattern implementations can be easily extended using abstractions provided by our DSL based pattern realization.

As pattern implementation should be modular and effective, we have designed a DSL for security patterns. Many patterns have a cross-cutting nature (see also [19]), therefore we decided to provide a DSL with support for aspect-oriented programming [18]. The idea is similar to providing an aspect-oriented DSL for transactions [20]. Furthermore, as pattern implementations should be effective, enforcement code in patterns is written in a scripting language.

```
pattern B2BConfidentiality {
  beforeServiceSelection { … }
  beforeServiceCall { … }
  afterServiceCall { … }
}
```

Figure 6. Example pattern.

In Figure 6. an excerpt of a security pattern is provided. In our framework, a pattern is a module that has several entry points through which the pattern can be invoked to enforce security. There are different types of entry points used to trigger a particular portion of enforcement implemented by the pattern. For example, "beforeServiceSelection" is the entry point at which the code in the curly brackets is executed, which happens before the service registry is requested. In the next section, we will further discuss how pattern implementations are invoked and what different types of entry points are used.

B. Policy Enforcement

The policy enforcement is carried out by a security monitor which is integrated into composite application container. It constitutes a design and execution environment for application development. Figure 7. shows the basic components of the container. As mentioned in Section II.B, the container has an integrated design time to develop composite applications in a business scripting language. When a business script is saved, the *script parser* processes the script to make it executable by the *execution engine*. While executing processes, the process uses *container services*, which includes a *service registry* and a *messaging service*. When executing human tasks, control is passed to the *tasklist UI* through which manual tasks can be completed.

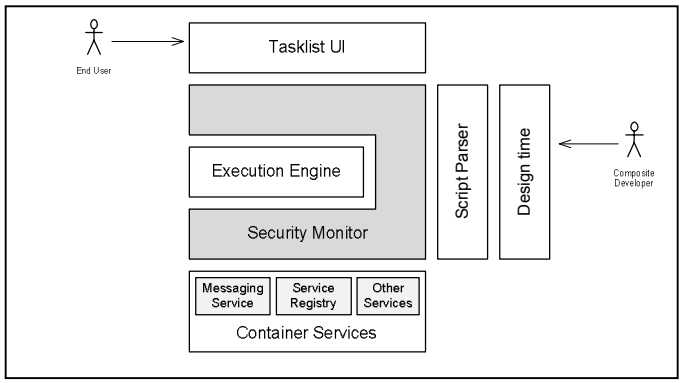

Figure 7. Composite Application Container Architecture.

For the integration of the security framework, the *script parser* was extended to support the declaration of security intentions on the process model. Further, a *security monitor* component was introduced. Several other components in the container have been enhanced in order to let the security monitor observe the execution of processes and interfere where necessary.

1) Security Monitor

The security monitor has the power to inspect and change the state and behavior of other container components (e.g., parser, messaging service). Behavior is observed by *events* produced by the components and the relevant state is accessible through the *context* of an event. The security monitor has access to the process model and its security intention declarations in order to know *what* must be enforced. In the same way, the monitor accesses the pattern repository in order to know *how* intentions must be enforced.

As one of its most important tasks, the security monitor observes process execution through hooks that have been introduced in the execution engine. Thereby the security monitor follows the concept of *total mediation*, i.e., every security-relevant event in process execution is intercepted by the monitor. Before the event actually happens, the monitor invokes selected patterns, which have the opportunity to check

and update the actual state or may alter the effect of the intercepted event to enforce security. For this the process execution engine must know which activities produce which events, and also the security monitor must know the composed set of intentions used for this process.

Each task of a process that is executed may produce several security-relevant events of a certain type. While executing a task, the process execution engine generates runtime events. The type of such an event corresponds to what currently happens in the task. Before the generated event becomes effective, it is deferred and delivered to the security monitor, which then may execute one of the runtime protocols presented in Section III and also may invokes all selected patterns at the entry point which correspond to the event type. We consider following event types:

a) Process model change event

This event is generated by the integrated design time when a process description or its security intention declaration is altered and saved. In systems that do not have an integrated design time, an analog event would be generated at deployment time. At this point, the container executes the security configuration protocol. In case of this event, there is no relating pattern entry point type to be invoked.

b) Service selection event

This event is generated when a task uses a service registry to select services of a certain category. This event is needed in order to be able to filter the service selection regarding security requirements. First, the container triggers the "beforeServiceSelection" entry points to generate a composed policy for the composite, which then is used for the service selection. Next, the container retrieves a list of all services of the selected category and uses the policy negotiation protocol from section III for each service in the list. If no agreed policy can be found for a service the service is removed from the list. Finally, the filtered list of services is return to the process.

c) Before service call event

Whenever a service call happens, an event is generated. As well, a context is set up which contains the message to be sent. Then, the container triggers the "beforeServiceCall" entry points in the patterns, which may alter the message content before it is finally sent out by the container.

d) After service call event

Whenever a service returns a result, an event is generated and a context is set up which contains the received message. First, the container triggers the "afterServiceCall" on the pattern in order to transform the message, before its data is further used in the process.

e) Human task execution

Before a human task is executed, this event is generated. The security monitor checks whether the user has enough permission to execute the task.

Because enforcement can only happen when an event is fired, enforcement is limited by what kinds of events are captured by the security monitor. However, the security monitor can easily be extended with new types of events.

V. IMPLEMENTATION

In this section, we present a prototype container that implements the security architecture. The implementation comes with a security infrastructure that provides security services that follow established Web service standard specifications. We implemented the container in Ruby. Due to space limitations, the paper will not discuss the realization of process execution engine in detail but only addresses the realization of security in the container.

In the current implementation, security intentions can be declared by providing a list of intention names with the process definition, which we will call "intention list" in the following. The "and" combinator is the only operator to compose intentions at the current time. More sophisticated combinators will be provided in the future.

For the current prototype, we assume that intentions composed do not have any negative interaction, i.e., they do not interfere with each other so that the enforcement of one intention contradicts with the enforcement of the other enforcement. If such negative interactions are possible, the implementation of the security intention is responsible to deal with it in such way that there is no negative effect on the enforcement.

A. Generic Security Services

In a composition environment, as discussed in Section II, each service may act as a consumer service and a provider service. In order to be able to address different security requirements each service will need a set of security services, each of which provides a well-defined security functionality. We implemented following security services for this purpose, as illustrated in Figure 8. :

- Backend policy generator is used to generate authorization policies which are used by backend policy enforcement.
- Backend policy updater is used to check whether a certain policy exists in a backend policy base and inserts the policy if necessary.
- Policy generator is used to generate WS-SecurityPolicy policies from service interaction annotations.
- Policy matcher matches compatible assertions between two WS-Security policies, resulting in an agreed policy.
- Policy registry is used to store and retrieve policies for external service interactions.
- Token Engine is used to embed tokens into a SOAP message and provides the token signature verification functionality.
- Security Token Service is used to generate a SAML token.
- CryptoEngine provides functionality for encrypting, decrypting as well as signing and verifying SOAP messages.

- Policy Decision Point is used to enforce access control policies encoded in XACML.

In the subsequent sections we will present how these services will be used by the runtime policy enforcement.

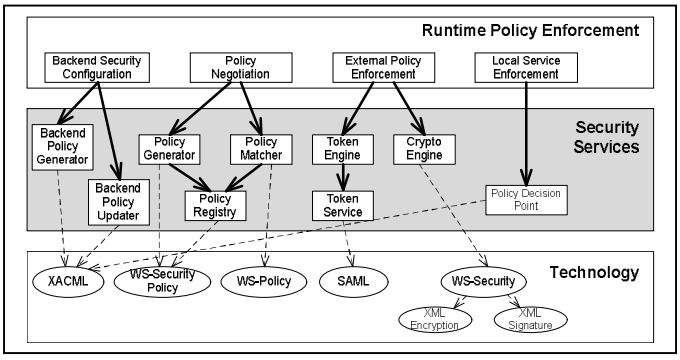

Figure 8. Genereic security services.

B. *Automatic backend and local security configuration*

In case of a "process model change" event (see Section IV), the backend security policies and local policies need to be updated based on the authorization requirements of the current process model. The backend policy update process works as follows. The authorization intentions are extracted from the process description and passed to the backend policy generator, which generates corresponding XACML [5] policies. Backend systems provide a separate "authorization policy updater" Web service interface for managing its authorization configuration. The policy generator passes each generated XACML policy to the policy updater service, which is provided by a backend system as a separate "authorization policy updater" Web service. The policy updater embeds the received policy into an XACML request, which is then sent to the Policy Decision Point (PDP) in the backend. The PDP then returns either an XACML "permit" response or "deny" response, depending on whether the received policy exists or not. In the negative case, the policy will be inserted into the policy base by the policy updater. We implemented the policy updater in Java and used Sun's XACML implementation for the PDP functionality [31].

The local policy update process works in a similar way except that the policies will be updated in the local policy base. These policies are mainly used to enforce authorization at the UI level.

For the sake of simplicity, we don't consider the revocation of assigned permissions.

C. *Runtime Enforcement*

The security monitor, in particular the code for security event handling, is embedded into the different components of the container, as explained below.

- The design time recognizes and handles the "process model change" events and triggers the "backend security configuration".
- The service registry (see Section IV) handles all "service selection" events and triggers the "policy negotiation".
- The messaging service (see Section IV) handles "before service call" events when sending SOAP messages or respectively handles "after service call" events when receiving results.
- The service call events are handled through triggering "external policy enforcement".
- When local service and data is accessed as well as when the UI processes a "human task execution" event, "local service enforcement" is triggered.

Figure 9. illustrates the runtime enforcement for a shipment process (see Section IV) with a given service interaction annotation "**`enforce`** `B2BConfidentaility` **`and`** `B2BIntegrity`" and an assign annotation "**`assign roles`** `[manager]` **`to`** `select_carrier`". The first annotation expresses that any B2B Web service interaction should be secured by encrypting and digitally signing the exchanged SOAP messages. The second annotation expresses that only users acting in the role of "manager" are allowed to select a carrier. In order to enforce these annotations, while executing each task of the process, the execution engine triggers events that are reported to the security monitor which will then take care of the required security enforcement activities.

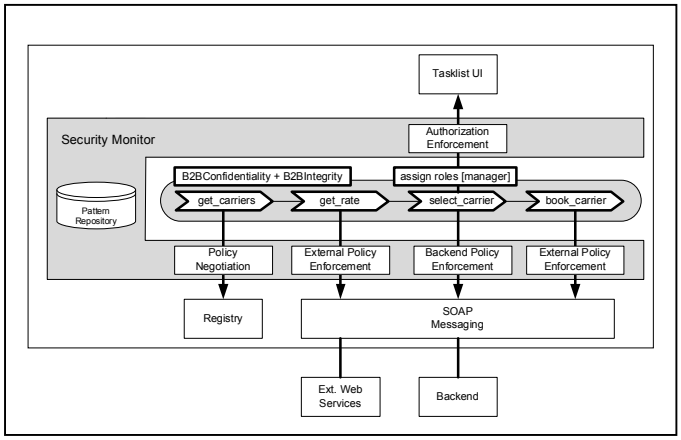

Figure 9. Runtime enforcement.

From the security enforcement perspective, the execution of the task "get_carriers" includes the selection of carriers, for which there is a mutual agreement between the shipment process and carrier services. This means that the QoP requirements B2Bconfidentiality and B2Bintegrity of the shipment process must match the QoP requirements of each selected carrier. As a first step, the policy negotiation process is performed according to the WS-Policy [21] specification. For the negotiation, we generate a WS-Policy policy that represents the high-level QoP requirements given by the intentions. As well, we update the carrier policies in the policy registry if necessary in order to cope with changing policies of invoked services at runtime. The generated policy is then intersected with all WS-Policy policies of the selected carriers. Policy intersection is the core function of the negotiation process in WS-Policy. The intersection identifies compatible policy alternatives (if any) included in both shipment process and carrier policies. Intersection is a commutative, associative function that takes two policies and returns a policy which still needs to be cleared of all invalid alternatives, as required by

WS-SecurityPolicy [22]. As a next step, the most appropriate policy, which is named as the agreed policy, is selected from all valid alternatives. The policy negotiation is accomplished by the policy matcher. It first requests available services from the service registry and then selects only the services for which a non-empty agreed policy has been produced. Speaking in terms of WS-SecurityPolicy [22] specification, a non-empty policy would include a confidentiality assertion and an integrity assertion. We also assume that an agreed policy includes a SAML token assertion, specifying the authorization requirement of a carrier service. The policy matcher is implemented in Java and utilizes Apache WS-Commons Policy [32].

The execution of the task "get_rate" requires Web service invocations, each of which needs to be regulated based on the corresponding agreed policy. Before sending out a request to an external Web service, the monitor retrieves required patterns for each intention listed in the service interaction annotation. The security monitor executes the patterns in the order as specified in the annotation by invoking the entry point "beforeServiceCall". The pattern code transforms the actual SOAP message into a secure message by encrypting and signing the message in order to fulfill the security objectives represented by the given intentions. The pattern code adds also a SAML token into the request, as needed by the agreed policy. Finally, the security monitor sends a request to the Carrier web service in the form of a WS-Security encoded SOAP message.

Upon receiving the service request, the carrier service performs required cryptographic operations on the SOAP message and verifies the SAML token. The PDP of the carrier service then evaluates the service request based on the token information. In the positive case, a rate will be calculated and embedded into a SOAP message. This will be encrypted and signed, and returned to the shipment requester.

After receiving the response of the invoked Web service, the monitor executes the pattern enforcement code of the entry point "afterServiceCall". This results in verifying the signature and decrypting the content.

From the security enforcement perspective, the execution of the task "select_carrier" requires an approval step which should be performed under consideration of the specified role assignment intention. Execution of the "select_carrier" task involves two activities: selection of the carrier in the UI and persisting the selection result in the backend. This requires a two-phase access control protocol. Enforcement in the UI guarantees that only carrier selection tasks can be seen and completed by the users acting in the role of "manager". Storing a selection result may require special permissions in the backend systems. Assuming that these permissions have already been updated as discussed above in this section, backend policy enforcement adds a SAML assertion token to the SOAP message which is then sent to the backend system. In our example, the SAML token encodes the user role "manager". Upon receiving the service request including the SOAP message and the token, the token manager of the backend system verifies the token and extracts the role information.

The last process step "book_carrier" calls the previously selected carrier service by performing a policy enforcement in the same way as it is done for the task "get_rate".

VI. Related Work

To the best of our knowledge, no other security frameworks for scripting based composite application development exist. However, some related work on secure software engineering, model-driven security, secure composition, and security DSLs in general is available.

In the area of secure software engineering [1] discusses security issues that arise in the interactions between software engineering and security. The authors in [1] emphasize the importance of unifying the design of systems and security policies.

The secure mediation approach presented in [30] is the main source of our inspiration for specifying intentions and generating policies from the intentions. [30] proposes the specification of access authorizations as annotated application schema declarations. Authorizations policies are generated from the annotated declarations.

Recent research in the area of UML based model-driven security followed a similar approach as presented in [30]. The work in [2] shows a model driven security approach which enables designers specify system models along with their access control requirements and use tools to automatically generate system architectures including authorization permissions and assertions for OCL constraints. The authors in [3] discuss a policy-driven approach to achieve effective management of security policies for applications. [4] presents an approach which specializes the concept of model driven architecture to model driven security by providing a so-called SECTET framework. Low level security requirements of a business application are modeled at a higher level of abstraction and merged with the business requirements modeled as platform independent models. Another UML based model-driven security approach is discussed in [6]. This work shows how security configurations for web services can be generated from the high-level security intents (security annotations like "integrity") attached to UML class diagrams. The work in [7] proposes a candidate profile for UML that presents security-related primitives (intents) as stereotypes that can be applied to UML elements when working with business stakeholders to capture security requirements. Like [6] we use security intents which are quite similar to the security primitives presented in [7] .

In the area of secure composition [9] presents an approach for modeling security constraints and a brokered architecture to build composite Web services according to the specified security constraints. In [10] the authors show an extension to the business process orchestration language WS-BPEL [33] in order to capture the specification of authorization information. [11] presents a formal model for consolidating the access control of composite applications.

Some existing research has also addressed the usage of DSLs for security policy specification and enforcement. [25]

presents SQLj, a domain specific extension to Java for developing secure service-based systems.

VII. Conclusion and Future Work

Our approach enables a business oriented application developer to add high-level security intentions to his business process model. The framework supports the automatic generation of security configuration and enforcement. Our concept work and implementation need to be extended to provide a more mature DSL for the specification of security intentions. There is a need to provide a more sophisticated set of intentions. This will require to extend our current pattern library. Last but not least, we also need to prove our approach in realistic scripting based software development environments.

Acknowledgements

This work benefited from discussions with Holger Mack, Murray Spork, Eric Solberg, and Juergen Schmerder.

References

[1] P. T. Devanbu and S. Stubblebine, "Software Engineering for Security: a Roadmap", International Conference on Software Engineering, Proceedings of the Conference on the Future of Software Engineering, Pages: 227 - 239 , Limerick, Ireland, June 2000.

[2] D. Basin, J. Doser and T. Lodderstedt, "Model Driven Security: from UML Models to Acccess Cotnrol Infrastructures", ACM Transactions on Software Engineering and Methodology (TOSEM), volume 15 , issue 1, pages: 39 – 91, January 2006.

[3] N. Nagaratnam, A. Nadalin, M. Honda, M. McIntosh and P. Austel, "Business-driven security: From modeling to managing secure applications", IBM Systems Journal, volume 44, no 4, 2005.

[4] M. Alam, M. Hafner, R. Breu. „Model-Driven Security Engineering for Trust Management in SECTET", Journal of Software, Academy Publisher, volume 2, no. 1, February 2007.

[5] OASIS eXtensible Access Control Markup Language (XACML), http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[6] Y. Nakamura, M. Tatsubori, T. Imamura, and Koichi Ono, "Model-Driven Security Based on a Web Services Security Architecture", In Proceedings of the 2005 IEEE International Conference on Service Computing (SCC'05), Pages: 7 - 15, July 2005.

[7] S. Johnston, "Modeling Security Concerns in Service-Oriented Architectures", http://www.ibm.com/developerworks/rational/library/4994.html

[8] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?", http://www.martinfowler.com/articles/languageWorkbench.html

[9] R. Bishop, B. Carminati, E. Ferrari, P.C.K. Hung, "Security Conscious Web Service Composition with Semantic Web Support", In Proceedings of the 1st ICDE Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07), Istanbul, Turkey, April 2007.

[10] E. Bertino, J. Crampton and F. Paci, "Access Control and Authorization Constraints for WS-BPEL", In Proceedings of the IEEE International Conference on Web Services (ICWS'06), pages: 275 – 284, 2006

[11] M. Wimmer, A. Kemper, M. Rits and V. Lotz, „Consolidating the Access Control of Composite Applications and Workflows", In the Proceedings of 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, pages 44-59, Sophia Antipolis, France, July-August 2006.

[12] A. Anderson, "An Introduction to the Web Services Policy Language (WSPL)", http://research.sun.com/projects/xacml/Policy2004.pdf

[13] The EU Project TrustCoM, http://www.eu-trustcom.com

[14] N. R. Mead, E. D. Hough and T. R. Stehney II, "Security Quality Requirements Engineering (SQUARE) Methodology", Technical Report, CMU/SEI-2005-TR-009, ESC-TR-2005-009, http://www.sei.cmu.edu/publications/documents/05.reports/05tr009.html

[15] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad, „Security Patterns: Integrating Security and Systems Engineering", Wiley Software Patterns Series, ISBN-10: 0470858842

[16] C. Giblin, A. Y Liu, S. Müller, B. Pfitzmann and X. Zhou, "Regulations Expressed As Logical Models (REALM)", In Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems, IOS Press, Amsterdam, pages: 37-48, 2005.

[17] A. Charfi and M. Mezini, "Using Aspects for Security Engineering of Web Service Compositions", IEEE International Conference on Web Services (ICWS), July 2005.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220–242, Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] J. Hannemann, G. Kiczales, „Design pattern implementation in Java and aspectJ". In ACM Object-Oriented Programming, Systems, Languages, and Applications 2002, Seattle, Washington, USA, November, 2002.

[20] J. Fraby, "Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code ", PhD thesis, Vrije Universiteit Brussel, 2005.

[21] WS-Policy, http://www.w3.org/Submission/WS-Policy/

[22] WS-SecurityPolicy, docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-cs.pdf

[23] J. Lee, K. E. Seamons, M. Winslett and T. Yu,. "Automated Trust Negotiation in Open Systems", In Secure Data Management in Decentralized Systems, edited by T. Yu and S. Jajodia, Springer, December 2006.

[24] WS-Federation, http://www.ibm.com/developerworks/library/specification/ws-fed/

[25] R. Bharadwaj and S. Mukhopadhyay, "SOLj: A Domain-Specific Language (DSL) for Secure Service-Based Systems". In Proceedings of the 11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'07), pages: 173-180, March 2007.

[26] K. Yskout, T. Heyman, R. Scandariato, and W. Joosen, "An inventory of security patterns", Technical Report CW-469, Katholieke Universiteit Leuven, Department of Computer Science, 2006.

[27] D. Koenig, "Groovy in Action", Manning Publications, New York, Januar, 2007.

[28] M. Schmidt, Enterprise Integration with Ruby, Pragmatic Bookshelf, ISBN-10: 0976694069, April 2006.

[29] Ruby Application Archieve, http://raa.ruby-lang.org/

[30] C. Altenschmidt, J. Biskup, U. Flegel and Y. Karabulut, "Secure Mediation: Requirements, Design, and Architecture", Journal of Computer Security, volume 11, number 3, pages: 365-398, 2003.

[31] Sun's XACML Implementation, http://sunxacml.sourceforge.net/

[32] Apache WS-Commons Policy, http://ws.apache.org/commons/policy/index.html

[33] WS-BPEL, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[34] A. Barros, M. Dumas and A. ter Hofstede, "Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection". Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.

[35] SAML, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

[36] SCA Policy Framework, http://www.oasis-opencsa.org/SCA-policy-framework