WS-Policy4MASC – A WS-Policy Extension Used in the MASC Middleware

Vladimir Tosic1,2,3, Abdelkarim Erradi1,4, Piyush Maheshwari1,4
1 School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
2 Empirical Software Engineering (ESE) Program, NICTA, Sydney, Australia†
3 Department of Computer Science, University of Western Ontario, London, Canada
4 IBM India Research Lab, New Delhi, India
vladat@computer.org, aerradi@cse.unsw.edu.au, pimahesh@in.ibm.com

Abstract

WS-Policy4MASC is a new XML language that we developed for specification of monitoring and control (particularly, adaptation) policies in the Manageable and Adaptable Service Compositions (MASC) middleware. It extends the Web Services Policy Framework (WS-Policy) by defining new types of policy assertions. Goal policy assertions specify requirements and guarantees to be met in desired normal operation. Action policy assertions specify actions to be taken if certain conditions are met or not met. Utility policy assertions specify monetary values assigned to particular situations. Meta-policy assertions are used to specify which action policy assertions are alternatives and which business value-driven conflict resolution strategy should be used. WS-Policy4MASC also enables detailed specification of additional information necessary for run-time policy-driven management. We evaluated feasibility of the WS-Policy4MASC solutions by implementing a policy repository and other modules in MASC. We examined their usefulness on a set of realistic scenarios.

1. Introduction and motivation

Policy-driven management [1] has caught considerable attention in the area of management (monitoring and control) of networks and distributed systems. A policy can be defined as a collection of high-level, implementation-independent, operation and management goals and/or rules expressed in a human-readable form. Policies can be viewed as decision-making guidelines for operation and management of a system. A policy-driven management system refines these high-level goals and rules into many low-level, implementation-specific, actions controlling operation and management of particular system elements. For example, a policy could be used to: ensure compliance, configure behavior, or achieve adaptability. Several classifications of policies exist. We find the classification from [2] particularly useful. It differentiates action policies (describing actions to take in a state), goal policies (describing desired states of the system), and utility function policies (defining value for each possible state).

In spite of recent significant progress on developing technologies for Extensible Markup Language (XML) Web services, there are still many open issues related to management of Web services and their compositions. Management is needed to achieve correct operation, recover from faults, optimize performance, increase security, perform accounting, and achieve maximal benefits from the managed systems. One of the prerequisites for performing management of Web services and their compositions is existence of a machine processable and precise format for description of monitored requirements, guarantees, capabilities, and control actions. Many languages have recently appeared to address some aspects of this need, usually specification of quality of service (QoS) requirements/guarantees. Some of these languages are accompanied by corresponding management middleware. Some examples are the Web Service Level Agreement (WSLA) [3], the Web Service Offerings Language (WSOL) [4], the Web Services Agreement Specification (WS-Agreement) [5], the Web Services Policy Framework (WS-Policy) [6, 7], the Web Service Constraint Language (WS-CoL) [8]. However, the past results have predominantly addressed (simpler) monitoring or QoS-based selection of Web services and less (more challenging) control (e.g., adaptation). Further, they mainly concentrated on management of individual Web services, while challenges of management of Web service compositions are relatively under-explored. In addition, the current solutions are almost exclusively focused on optimization of technical QoS metrics (e.g., response time) and provide only a very simple treatment of tangible business metrics (e.g., profit) without examining intangible business metrics (e.g., customer retention).

Policy-driven management could offer solutions to these challenges. We determined a number of requirements for policy-driven management of Web services and their compositions (see [9] for detailed requirements for a policy language, [10] for a brief discussion of wider issues) and concluded that none of the past languages and middleware tools fully addresses them. Therefore, we develop a novel policy-driven Web service management middleware – Manageable and Adaptive Service Compositions (MASC) [11, 12, 14]. Compared to related work, MASC has several distinctive characteristics. It provides monitoring of both Web services

† NICTA is funded through the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council (ARC).
and their compositions, a wide range of dynamic (run-time) adaptation mechanisms (handling business exceptions, versions, faults, performance problems), coordination of adaptation actions between the SOAP messaging layer and the process orchestration layer, and the capability to select between alternative control actions to maximize tangible and intangible business values in various ways. Further, it leverages and extends the power and flexibility of the recently published Microsoft .NET 3.0 platform [15] (the Windows Workflow Foundation – WF and the Windows Communication Foundation – WCF), for which the previous Java-based solutions are not fully appropriate.

This paper presents the WS-Policy4MASC language, which is our novel language for description of policies used in the MASC middleware and, thus, for its automatic configuration. As we will explain, WS-Policy4MASC is domain-independent extension of the WS-Policy [6, 7] industrial specification. While we have examined extending other Web service management languages (e.g., WS-Agreement) and other policy-driven management languages, we have found that WS-Policy is most suitable for the MASC middleware and is widely used in practice. WS-Policy4MASC is a powerful general language for specification of various Web service management policies. However, we put a particular emphasis on supporting aspects that differentiate MASC from other Web service middleware. Particularly, our language enables detailed specification of information for various types of adaptation of Web service compositions and for selection between alternative control actions using strategies for maximization of various types of business value. Consequently, WS-Policy4MASC provides a number of solutions that are not present in past related works.

In this section, we have briefly introduced the area of policy-based Web service management and motivated our development of WS-Policy4MASC. The following section summarizes the main related works, putting particular emphasis on WS-Policy. An overview of WS-Policy4MASC is given in Section 3. Then, Section 4 briefly presents how we evaluated WS-Policy4MASC by using it in the MASC middleware. At the end, we summarize conclusions and outline our ongoing and future work.

2. Related work

The main related work for our project is the Web Services Policy Framework (WS-Policy) [6, 7], an industrial specification standardized by the World Wide Web Consortium (W3C). It is a general framework for specifying various Web service properties in a way that complements the Web Services Description Language (WSDL) and the Web Services Business Process Execution Language (WSBPEL). In the WS-Policy model, a policy is a collection of policy alternatives, each of which is a collection of policy assertions. WS-PolicyAttachment defines a generic mechanism that associates a policy with subjects to which the policy applies, such as WSDL elements or Web service registry information. Various policy subjects are possible, e.g., service, endpoint, operation, message, or message part. A policy scope is a set of policy subjects to which a policy may apply.

WS-Policy has many good features. For example, it is simple, extensible, and flexible (e.g., policies can be specified inside and outside WSDL and WSBPEL files). However, it is only a general framework, while details of specification of particular categories of policies are left for specialized languages – domain-dependent extensions of WS-Policy. Currently, only standard extensions for security, reliable messaging, and a few other management areas that were not the focus of our project had been published. WS-PolicyAssertions specification (now deprecated) could be used for formal specification of functional constraints, but the contained expressions can be specified in any language. It is not clear whether and when some standard specialized languages for the specification of QoS, prices/penalties, and other management information will be developed. Some unification and standardization of common elements (e.g., expressions) of various WS-Policy languages would reduce the overhead of supporting this framework. Further, WS-Policy does not detail where, when, and how are policies monitored and evaluated. Since many policies have to be monitored and controlled during run-time, WS-Policy needs better support for management applications, including explicit specification of such management information. Consequently, we had to develop a new domain-independent WS-Policy extension, which we named WS-Policy4MASC.

The WS-Policy specification is currently evolving within the W3C standardization process [6], so a few solutions in WS-Policy4MASC (notably, dependencies between policy assertions) are not completely aligned with the current WS-Policy guidelines. We have decided to work with the past stable WS-Policy version described in [7], but plan to align WS-Policy4MASC with new versions once they are stable and to provide input into future evolution of WS-Policy.

Another important related work is Web Service Constraint Language (WS-CoL) [8], a domain-independent WS-Policy extension for specifying client-side monitoring policies. At deployment time, WS-CoL constraints attached to a process are translated into WSBPEL invoke activities that call a monitoring manager component to evaluate monitoring policies and detect anomalous conditions. This approach is similar to ours in that monitoring policies are specified externally rather than being embedded into process specification. It achieves the desired reusability and separation of concerns. However, it only provides support for monitoring and focuses mainly on security. It does not provide full support for diverse adaptation and business-driven management that are distinctive characteristics of our research.

The Web Services Policy Language (WSPL) [16] is an early version of the XACML Profile for Web Services (WS-XACML) standardized by the Organization for the Advancement of Structured Information Standards (OASIS). It is suitable for specifying a wide range of policies, e.g., acceptable and supported encryption algorithms or privacy guarantees. Its key strength is the ability to support negotiation of mutually acceptable policies that represent intersections of two source policies. While current version of WS-
3. Overview of WS-Policy4MASC

3.1. Main constructs in WS-Policy4MASC

WS-Policy4MASC extends WS-Policy by defining XML schemas with new types of WS-Policy policy assertions. These are not domain-specific policy assertions because they can be used for representing functional constraints, QoS, adaptation, security, prices, and other information. Compared to an a posteriori combination of several single-domain policy solutions, a true integration of different domains (as done in WS-Policy4MASC and a few older works, e.g., WSOL) leads to better specification of inter-domain dependencies, increased interoperability between domains, and lower total run-time overhead [4]. WS-Policy4MASC mandates no changes to WS-Policy constructs (e.g., <Policy>, <All>, <ExactlyOne>), so they can be used with WS-Policy4MASC in the same way as for other WS-Policy policy assertions. (The impact of specification of WS-Policy4MASC constructs in the compact form instead of the canonical form of WS-Policy is discussed in [9].) WS-Policy4MASC policy assertions can be attached to WSDL constructs (e.g., service, endpoint, operation, message) and WSBPEL constructs (e.g., process, sub-process, activity).

Another recent research trend to address management issues is augmenting Web services middleware with autonomous behavior capabilities such as self-healing and self-configuring [17, 18]. Notably, IBM’s Policy Management for Autonomic Computing (PMAC) [18] is a policy driven framework intended to simplify creation, storage, distribution, and execution of policies. It uses the Autonomic Computing Policy Language (ACPL) as the underlying policy language. ACPL is a strongly-typed XML-based language for specifying policy rules and expressions. Its key concepts are: scope, condition, decision, and business value. The scope specifies the policy subject. The condition expresses when a policy is to be applied. The decision describes observable behavior or desired outcome of a policy in the form of a management action or a configuration profile. The business value expresses utility functions to make economic trade offs between policy alternatives. We found ACPL tightly coupled to PMAC and hence more suitable for configuring resources and managing applications. It offers limited support for monitoring and adaptation of Web service compositions. Furthermore, the policy subject is specified within the policy definition and this hinders reusability and maintainability of policies. On the contrary, WS-Policy offers more flexibility and has better acceptance by industry. Our work belongs to the autonomic Web services research area, but has unique characteristics, outlined in the previous section.

Over the last decade, multiple approaches have been suggested for specifying policies for different application domains. Most of these proposals target network management, security, privacy, and trust. Our work is closest to a few recent works for monitoring and adaptation of Web services and their compositions. For example, the proposed policy-driven Web service transactional frameworks (e.g., [19]) only address coordination of activity termination and possibly compensation to ensure that participating Web services reach consistent states after a failure. Our work complements them with suitable repair policies. However, we have only studied policy-driven local repair and adaptation strategies.

There is a body of work on policy refinement and policy conflict detection and resolution. Our MASC middleware supports basic policy refinement by mapping WS-Policy4MASC assertions into calls to MASC middleware management interfaces. For run-time policy detection and resolution, MASC uses WS-Policy4MASC meta-policies that describe how to maximize business value. However, we left design-time analysis of policies to detect and resolve policy conflicts for future work. We plan to reuse and/or adapt existing solutions in this area, e.g., [20].
some time, but also to state transitions, execution/non-execution of action policy assertions, satisfaction/non-satisfaction of goal policy assertions, and other events. However, the meaning of the WS-Policy4MASC term “action policy assertion” can be considered the same as the meaning of the [2] term “action policy”. They both represent a set of actions to perform if particular conditions are met.

In addition to these 4 new types of "real" policy assertions, WS-Policy4MASC enables specification of additional information that is necessary for run-time policy-driven management. Some of this information (e.g., which party performs evaluation/execution of a policy assertion, which party is responsible for meeting a goal) is specified in attributes of the above-mentioned "real" policy assertions. Much more information is specified in additional WS-Policy4MASC constructs, specifying monitored data items, states, state transitions, schedules, events, scopes, and various expressions (Boolean, arithmetic, arithmetic with units, string, date/time/duration). These "other" constructs are also implemented as WS-Policy policy assertions from the syntax viewpoint (although they are not policy assertions from the semantic viewpoint), to support reusability and ease automatic code generation (this is elaborated below). Probably the most important of them is the <When> construct that specifies when something (e.g., evaluation of a goal policy assertion or execution of actions in an action policy assertion) should take place. The emphasis in this paper is on the 4 types of "real" policy assertions, while details about the other WS-Policy4MASC constructs can be found in [9].

Simplicity of automatic code generation from XML schemas into C# classes and reusability of specification elements significantly influenced the design of our language. For example, we avoided the <choice> element in XML schemas for WS-Policy4MASC because it resulted in C# classes difficult to understand and handle. A drawback of our approach is that many WS-Policy4MASC supporting constructs (e.g., event definitions) are specified as WS-Policy policy assertions, leading to specifications that are somewhat verbose. However, we hope that, in the future, WS-Policy4MASC files will be generated by graphical tools, so this verbosity of WS-Policy4MASC files will not be a strain for humans.

Some relationships between the main WS-Policy4MASC constructs are shown in the UML ((Unified Modeling Language) diagram in Figure 2. Due to the space constraints, the figure does not show all existing constructs and relationships. Goal policy assertions, action policy assertions, utility policy assertions and meta-policy assertions are subtypes of the abstract policy assertion construct MASCPolicyAssertion. It defines common attributes, such as policy assertion ID and party that performs execution/evaluation. The 4 policy assertion types have additional attributes and elements. The former 3 types of policy assertions reference a <When> element that contains information about one or more states in which something occurs (not shown in Figure 1), one or more events that can (each individually) trigger this occurrence, and an optional filtering Boolean condition to be satisfied. A goal policy references a Boolean expression with the condition to be evaluated – only if the given expression evaluates to “true” the goal was satisfied. An action policy assertion references a group of to be executed. A utility policy assertion contains an arithmetic with (currency) unit expression that determines associated business value. A meta-policy assertion does not reference a <When> element, but a set of mutually conflicting policy assertions and a conflict resolution strategy (this will be explained in Subsection 3.4).

While most WS-Policy4MASC policy assertions are intended for guiding internal monitoring and control processes, they can be, in principle, also advertised externally and matched between two parties, using the existing WS-Policy policy assertion matching algorithms. For example, such matching of utility (but also other) policy assertions can be used during Web service selection. We recently started research on extending the Universal Description, Discovery, and Integration (UDDI) with WS-Policy4MASC information and plan to research all aspects relevant for such matching.

As already noted, the main goal of WS-Policy4MASC is formal specification of policies used by run-time Web service management middleware, particularly MASC. This goal has influenced many design decisions. For example, we could have built into WS-Policy4MASC richer support for specification of semantics to aid in determining conflicts, consistency, and completeness of policy assertions within a policy. Integrating into WS-Policy4MASC some existing partial solutions to this problem (known as very difficult for many years) would certainly help humans using WS-Policy4MASC, but would also significantly complicate im-
3.2. WS-Policy4MASC Examples

Figure 2 illustrates some of the WS-Policy4MASC constructs on a simple example. (Many additional examples, including various adaptation of Web service compositions, are given in [9].) A weather report Web service has one operation: `Integer weatherTemperature(String postCode)`.

Figure 2 shows how WS-Policy4MASC can be used to specify the post-condition that the result is temperature in Celsius degrees between -70C and 50C. WS-Policy4MASC policy assertions and other constructs are specified within a WS-Policy element (namespace attributes are omitted from Figure 2 for brevity). First, the `<MonitoredDataCollection>`, `<MonitoredDataCollection>`, and `<ActionGroup>` constructs specify that the message part "weatherTemperature" is monitored and expressed in Celsius degrees. (For the previously mentioned reasons, this specification is verbose.) Definitions of states and events follow, but they are omitted from Figure 2. Then, a `<When>` construct referring to the state "Executing" and the event "MessageToBeSent" is defined. The subsequent action policy assertion specifies that when this event happens in this state, monitoring of the message part "weatherTemperature" is performed by the provider Web service. This action policy assertion is used to configure MASC monitoring modules. Definition of the Boolean expression "LimitsOfValidWeatherTemperature" is omitted from Figure 2 for brevity. This is a complex expression that specifies that values of the monitored data item (the message part "weatherTemperature") must be between -70C and 50C. The above-mentioned `<When>` construct and Boolean expression are referenced in the definition of the subsequent goal policy assertion, which is also used to configure MASC monitoring modules. This goal policy assertion specifies that when the event "MessageToBeSent" occurs in the state "Executing", then the provider Web service should evaluate the mentioned Boolean expression. It also states that the provider is responsible for meeting this goal.

In a separate file, shown in Figure 3, the standard WS-PolicyAttachment element defines that this policy (referred in the `<PolicyReference>` element) is applied to the WSDL message specified in the `<AppliesTo>` element.

3.3. Supported Adaptation Actions

Specification of Web service requirements/guarantees for monitoring activities was enabled by a number of past languages, such as WSLA, WSOL, and WS-CoL. In this area, WS-Policy4MASC offers only minor advantages. However, one of the distinctive characteristics of WS-Policy4MASC is built-in support for a diverse range of common Web service composition adaptation actions. We focused on actions handling frequent adaptation needs, such as customization (including versioning) and corrective adaptation for fault management. Most of these adaptation actions are executed by the process orchestration layer of MASC [14], but some are executed by the SOAP messaging layer of MASC without an intervention by the process orchestration engine [11].
Actions supported by the WS-Policy4MASC language and the MASC middleware can be grouped into: 1) monitored data collection (using logging, measurement, calculation); 2) monitored data transfer (using special push or pull operations); 3) cancellation of previously scheduled actions or events; 4) middleware configuration adaptation (e.g., changing parameters of used WS-* protocols); 5) messaging adaptation (data flow adaptation at the SOAP messaging layer, e.g., message transformation); 6) process instance structure adaptation (e.g., replacing a sub-process); 7) process instance execution adaptation (e.g., process termination); 8) activity instance execution adaptation (e.g., skipping an activity); 9) policy assertion adaptation (e.g., deactivating a policy assertion). We have also considered supporting some other actions, but left them for future work. Due to the space constraints, we will list only the actions for the last 4 groups. Details about how some of these actions are implemented in the MASC middleware are discussed in [14].

Process instance structural adaptation is supported with a number of actions: 1) Removal of a specified (e.g., using XPath) block of activities (or one activity) from the base (adapted) process. 2) Addition of an external known process at a specified point of the base process. 3) Addition of a single call to a known external Web service at a specified point of the base process. 4) Searching a specified Web service directory with specified parameters for an external process or Web service operation that, when found, is added at a specified point in the base process. 5) Replacement of a specified block of activities with an external known process. 6) Replacement of a specified block of activities with a single call to a known external Web service. 7) Searching a specified Web service directory with specified parameters for an external process or Web service operation that, when found, replaces a block of activities in the base process. While replacement could be relatively easily modeled as a removal plus an addition, we decided to model it explicitly because it is a common case and because its meaning is hidden when it is modeled with two separate constructs. 8) Activity instance execution adaptation is done with actions for process termination, suspension, and resumption.

Activity instance execution adaptation is also supported with many actions: 1) Cancellation of the currently executing activity. 2) Skipping the next activity that was supposed to be executed. 3) Skipping a block of activities from the next activity to a specified activity. 4) Rescheduling of the next activity for some (specified) later point in the process execution. 5) Compensation of the last executed activity (assuming that its compensation operation is known). 6) Process-level retrieval of the last activity (e.g., if it was not completed). 7) Suspension of the currently executing activity (this suspends its thread, but not the other threads in the process). 8) Resumption of a specified previously suspended activity.

Policy assertion adaptation is performed with actions for deactivation and activation of individual WS-Policy4MASC policy assertions. We plan to add priorities to policy assertions (to be used when there are no meta-policies), so we will also add an action to change these priorities during run-time.

3.4. Support for Business-Driven Management

A particular novelty of WS-Policy4MASC are utility policy assertions and meta-policy assertions, so we will describe them in more detail here. (While [10] outlined our early ideas, we now provide details and updates.) Detailed illustrative examples are given in [9].

Utility policy assertions enable providing monetary amounts to <When> constructs that contain information about allowed states, trigger events, and optional filtering conditions. For example, it is possible to specify amounts paid when goal policy assertions are met (or not met) or when action policy assertions are executed. Since all real monetary transactions require two parties, two attributes enable specification of the beneficiary party and the paying party. A positive monetary amount means that the beneficiary party receives payment from the paying party, while a negative amount denotes that the beneficiary party has to pay the paying party. In the former case, the beneficiary party has a benefit, while in the latter it has a cost. This is inverse for the paying party. The sum of (positive) benefits and (negative) costs is a profit. While it is mandatory to specify a beneficiary party, specification of a paying party is optional. If it is missing, this means that the specified monetary amount is not a real scheduled payment, but an estimate of some possible future business value from one or several paying parties. An additional Boolean attribute specifies whether the given amount is tangible (i.e., a real monetary amount paid) or intangible (i.e., a monetary estimate of some other business value, such as customer satisfaction). In a future version of WS-Policy4MASC, it will be possible to also specify schedules of future payments (currently, only one payment is specified per utility policy assertion), probability that an estimated future business value will be realized, and confidence in the precision of monetary estimates of intangible business values. It is also possible that a separation between various intangible business values will also be supported later. Note that all these utility policy assertions show absolute value of some state (situation). We have already developed, but not yet fully implemented in WS-Policy4MASC, additional specification of relative utility policy assertions that show difference in value (along some set of dimensions) between two states (situations). A meta-policy assertion lists several conflicting action policy assertions and a conflict resolution strategy used when some of them are triggered simultaneously. At this time, we have focused on strategies that choose only the best (as defined under some criteria) among the listed alternatives. In the future, we will also research strategies that allow conditional execution of more than one alternative. The WS-Policy4MASC language is extensible, so one can add new strategies. We have defined several strategies using various maximizations of business values, designed MASC middleware support (e.g., algorithms and data structures) for these strategies, and started implementing them in our MASC prototype (the MASCPolicyDecisionMaker module). The strategies are classified along 3 mutually orthogonal dimensions:
1. 'Both agreed payments and estimated future business values' vs. 'only agreed payments': One group of strategies adds all agreed payments and estimated future business values for the compared alternatives. The other group of strategies adds only agreed payments, but if difference between 2 (or more) such sums is less or equal some specified limit, then it separately adds estimated future values as a tie-breaker. In the future, we will develop strategies taking into consideration probability of estimated future business values.

2. 'Both tangible and intangible' vs. 'only tangible' vs. 'only intangible': One group of strategies adds all tangible and intangible business values for the compared alternatives. Another group of strategies adds only tangible business values, but if difference between 2 (or more) such sums is less or equal some specified limit, then it separately adds intangible business values as a tie-breaker. Conversely, yet another group of strategies adds only intangible business values, but if difference between 2 or more such sums is less or equal some specified amount, then it separately adds tangible business values as a tie-breaker. The latter strategies are used when intangible business metrics (e.g., market share, customer retention, customer satisfaction) are more important than immediate profit. In the future, we will develop strategies taking into consideration confidence in the precision of monetary estimates of intangible business values.

3. ‘Benefits and costs’ vs. ‘cost limit’: One group of strategies adds together all benefits and costs for the beneficiary party for the compared alternatives. The other group of strategies adds together all benefits and costs, but also (in parallel) separately adds only all costs. Then, it discards alternatives where the costs equal or exceed some specified limit (i.e., threshold). The latter strategies are used when too high costs are not acceptable (e.g., due to a lack of current funds), even if they bring higher long-term profit.

Since these 3 dimensions are mutually orthogonal, a strategy specifies behavior along each dimension, e.g., ‘only agreed payments’ (dimension 1), ‘only tangible’ (dimension 2), and ‘benefits and costs’ (dimension 3). This produces $2^3 \times 2 = 12$ combinations. For the 4 combinations with ‘only agreed payments’ along dimension 1 and either ‘only tangible’ or ‘only intangible’ along dimension 2, it is also necessary to specify which dimension is used as the first tiebreaker, which produces 2 variations per combination. Thus, the total number of strategies that we currently support is $12 \times 4 = 16$.

We also considered support for several additional dimensions, but decided to postpone their prototype implementation for future work. For example, [10] describes a possible differentiation between strategies that consider ‘only immediate’ impact of actions in compared policy assertions vs. those that calculate (using simulations) ‘long term’ impact.

4. Use of WS-Policy4MASC in MASC

Due to the space limits, we are not able to show and explain in this paper the complete architecture of the MASC middleware. Presentation and analysis of the current MASC version is available in [14], while [11, 12] contain additional details about some important aspects (mostly reused in the current version). Since support for MASC is the main goal of WS-Policy4MASC, this section briefly summarizes how we have evaluated feasibility and usefulness of our solutions.

We have evaluated feasibility of the WS-Policy4MASC solutions by implementing a policy repository and other modules in MASC. Within the MASC middleware, WS-Policy4MASC policy assertions are stored in an in-memory policy repository. It is a collection of instances of policy classes generated automatically from the WS-Policy4MASC schema, using an XML-schema-to-classes generator. (In our .NET 3.0 and C#-based MASC prototype, we used the XSD tool from .NET 3.0.) When MASC starts, our MCSProcessParser within it imports WS-Policy4MASC files, creates instances of corresponding policy classes, and stores these instances in the policy repository. The SOAPMessageLoggingService and the QoSMeasurementService monitor messages exchange between the composed Web services and collect monitoring information, such as values of message parts and measured QoS metrics. This information is stored in the Monitoring Database. Using this information, the MASCMonitoringService evaluates goal monitoring policies to detect adaptation triggers and events of interest. When they happen, the MASCMonitoringService generates an event (with all necessary information, e.g., process instance ID and monitored data values) to the MASCInvocation/DecisionPoint. The MASCInvocation/DecisionPoint determines adaptation action policy assertions to be applied and submits them to the MASCInvocationCoordinator for execution. If there are several alternatives, it chooses the one to execute, based on WS-Policy4MASC meta-policy and utility policy assertions and the strategies for maximization of business value.

Further, we have examined expressiveness, effectiveness, and usefulness of our WS-Policy4MASC solutions on a set of realistic stock trading scenarios [12, 14]. We have written WS-Policy4MASC files for some of the scenarios to check whether and how various adaptation needs can be expressed in our language. In addition, we have implemented and studied some of the scenarios with our prototype of the MASC middleware. For example, in some experiments [12] MASC dynamically adapted a base business process for national stock trading with support for international stock trading. In other experiments [14], we periodically injected random exception events at various stages of the stock trading process to study behavior of the system (with and without MASC) in response to faults or QoS changes of constituent services. In some experiments [12], we measured the overhead introduced by MASC on overall response time. The conducted experiments were completed successfully and demonstrated feasibility and usefulness of the MASC approach in adding dynamic adaptation capabilities to existing Web services compositions, guided by declarative policies specified in WS-Policy4MASC. MASC has provided a solution for policy-driven static and dynamic adaptation without any changes to the base process definition, implementations of the used Web services, or the implementation of .NET 3.0 technologies. All that is needed for adaptation (in addition to MASC software) is a WS-Policy4MASC document listing...
5. Conclusions and future work

WS-Policy is a promising Web service technology popular in the security domain. We argue that an extended WS-Policy can play a key unifying role in annotating WSDL and WS-BPEL Web service descriptions with various rules and support for Web service management (monitoring and control), such as service customization/versioning, fault management, and QoS management. Apart from WS-CoL, the past WS-Policy extensions were not developed to address the needs in these areas. Our WS-Policy extensions for the specification of goal policy assertions, action policy assertions, and utility policy assertions address the same needs as WSLA, WSOL, WS-Agreement, and WS-CoL. Thus, our work enables that an XML Web service composition can be comprehensively described using only WSDL, WS-BPEL, and WS-Policy4MASC. It provides many solutions that are not present in related works (including WS-CoL), such as specification of information for various types of adaptation of Web service compositions and for selection between alternative control actions based on various strategies for maximization of diverse (tangible or intangible, agreed or estimated) business values. We have defined and started implementing 16 such strategies. We have evaluated feasibility of the WS-Policy4MASC solutions by implementing a policy repository and other modules in MASC. This complements our examination of usefulness of these constructs, which we have performed on a set of realistic scenarios.

We have completed definition of the main XML schemas for WS-Policy4MASC. Our initial focus was on supporting monitoring and dynamic adaptation through WS-Policy4MASC goal policy assertions, action policy assertions, and related constructs (e.g., describing when something occurs). Once this was completed, our focus shifted to supporting business-driven management of Web services through using WS-Policy4MASC utility policy assertions and meta-policy assertions to select between alternative action policy assertions. As outlined in this paper, WS-Policy4MASC can be further improved in several ways, such as achieving complete compatibility with the current WS-Policy standardization draft and addition of specification of more formal semantics to aid in policy analysis. However, the main item for our ongoing work is further development of the proof-of-concept prototype implementation of the MASC middleware that uses WS-Policy4MASC policy assertions. While we already have a working MASC prototype [12, 14], we use an iterative development process to add new features into it (and, sometimes, the MASC architecture) and evaluate them on case studies (e.g., stock trading). In some cases, changes to the MASC architecture require changes to the WS-Policy4MASC schemas (language grammar). Further, we work on a new UML profile for design-time specification of information relevant for run-time Web service management activities using WS-Policy4MASC [21]. Having all this in mind, our language will continue to evolve.

References