Support for Concurrent Adaptation of Multiple Web Service Compositions to Maximize Business Metrics

Qinghua Lu 1,2, Vladimir Tosic 1,2

1 School of Computer Science & Engineering, University of New South Wales, Sydney, Australia
2 Business Adaptation and Interoperation Project, Managing Complexity Theme, NICTA 1, Sydney, Australia

Abstract—Runtime adaptation of Web service compositions can often be done in several ways, so it is necessary to decide which adaptation approach to take. While many research projects studied runtime adaptation of Web service compositions or business processes, this paper presents our unique solutions that maximize business metrics, in cases when several Web service composition instances should be adapted at the same time. We specify all necessary information about possible adaptions and their business metrics as policies in our WS-Policy4MASC language and model the optimization problem in the powerful constraint programming language MiniZinc. Into our MiniZnMASC middleware we integrated new algorithms that determine how to adapt each Web service composition instance so the total business value is maximized, while satisfying all given constraints (e.g., about resource limitations). Experiments with the MiniZnMASC prototype showed that our solutions are feasible, functionally correct, business beneficial, with low performance overhead, and with linear scalability.

Keywords—business-driven IT management; runtime adaptation; decision making; constraint programming; Web service management; business process management; autonomic computing

I. INTRODUCTION

Service-oriented architecture (SOA) implemented with Web service technologies [1] is currently the most popular approach to build information technology (IT) systems for business processes. In complex, long-running business processes changes happen often, both at the business level (e.g., business goal changes) and the technical level (e.g., service performance changes). When such changes occur during runtime, the affected Web service compositions should be adapted. This adaptation can be usually done in several ways and advanced decision making is needed to determine how to proceed. A particular layer of complexity is that often the change affects not only one Web service composition instance, but multiple Web service composition instances and possibly different Web service compositions at the same time. The notion of an “instance” is a runtime invocation of a Web service composition. For example, when a Web service composition is invoked 2 times, 2 different instances are created. Since each Web service composition can be viewed as an implementation of a particular business process type, a Web service composition instance is also a business process instance.

Since the affected concurrent Web service composition instances can have different characteristics, it is often necessary to adapt them in different ways. When a change occurs, instances at different runtime positions in the business process model might require different adaptation (e.g., instances that already passed the changed part of the business process usually require different adaptation from instances within the changed part of the business process). Adapting all affected instances in the same way can lead to sub-optimal solutions (e.g., due to limited underlying resources). Therefore, it is beneficial to examine how to concurrently adapt all affected instances individually to find the globally optimal adaptation.

The traditional IT system and service management approaches mainly focus on functional correctness and optimization of technical metrics, such as availability and response time. However, business metrics, such as profit and market share, are more important to business users of IT systems/services than technical metrics. For example, when response time of a service changes from 10ms to 30ms, a business user is usually not very interested in the response time change, but wants to know the business impact of this change, e.g., how much it will cost her/his business. The business-driven IT management (BDIM) [2] research area studies models, algorithms, and architectures for mapping business and technical IT metrics and making IT management decisions best from the business viewpoint. Decisions in BDIM are made based on such mappings and are performed to maximize total business value for the enterprise.

Another aspect is that it is hard for humans to manage Web service compositions implementing complex, long-running business processes (and many other complex IT systems) because it is difficult for them to understand multifaceted interdependencies between many diverse components and metrics at various levels of abstraction. There is a need that software performs most of the management activities automatically, with only minimal human supervision. Autonomic computing [3] is an approach towards reducing complexity in IT system/service management, where IT systems self-manage themselves using configurable policies, with minimal human intervention. Autonomic BDIM is the intersection area of autonomic computing and BDIM – it adds processing of business metrics to decision making components of autonomic systems. Moura et al. [4] concluded that autonomic BDIM is a promising research area, with many unsolved challenges and [5] noted that many of these challenges are relevant for management of Web services and Web service compositions.

While there has been a lot of research in the area of adaptation of Web service compositions and business processes, there has been relatively little research on advanced support for business-driven decision making for such adaptation, particularly when several Web service composition instances should be adapted at the same time, with minimal human in-
tervention (i.e., autonomically). In this paper, we present our MiniZnMASC middleware for decision making in autonomic business-driven adaptations of Web service compositions. It is an extension of our earlier middleware MiniMASC [6][7]. All necessary information about possible adaptations and their business metrics is specified through policies in the WS-Policy4MASC language [5][8], which is our extension of the WS-Policy industry standard. Both MiniZnMASC and MiniMASC contain advanced business-driven decision making algorithms for autonomic adaptation [5]. However, MiniZnMASC much better handles situations when multiple running Web service composition instances have to be adapted at the same time. In such situations, MiniMASC examines adaptation of all instances in the same way (which can lead to suboptimal solutions) and has less powerful support for considering various constraints (e.g., resource limitations). On the other hand, constraint programming can help in finding optimal decisions in complex circumstances where many constraints exist. MiniZinc [9] is a powerful constraint programming specification language, which has a free efficient constraint solving software. Therefore, MiniZnMASC integrates the MiniZinc solver into MiniMASC and implements new, powerful decision-making algorithms that can concurrently make different adaptation decisions for different classes of Web service composition instances in a way that achieves maximal total business value while satisfying all given constraints. Here, a “class of instance” is a group of Web service composition instances that share a combination of characteristics that warrants adaptation in the same way. The most important among these characteristics are: the implemented business process type, the executed Web service composition, the current position/state within the running Web service composition, the class of consumer (e.g., promised class of service), and other contractual obligations to the consumer. Note that in our current work all instances from the same class are adapted in the same way. Also note that our work deals not only with concurrent adaptation of instances of the same Web service composition, but also of instances of multiple different Web service compositions. However, since instances of different Web service compositions almost always belong to different classes of instance, we simply speak about concurrent adaptation of multiple classes of instance.

This paper is organized in the following way. Section II introduces a motivating example for this work. Section III examines the past related work. Section IV introduces the past results built upon in this work. Section V presents our constraint programming model for business-driven decision making for Web service composition adaptation. Section VI gives an overview of the architecture of MiniZnMASC. Section VII discusses our evaluation of MiniZnMASC. Section VIII concludes the paper.

II. A MOTIVATING EXAMPLE

In this section, we provide a motivating example to illustrate the need for concurrent making of different adaptation decisions for different classes of Web service composition instances. The example is a simplification of real-life business processes our broader research group worked with in the past several years. A loan broker company provides a loan brokering Web service composition shown in Fig. 1. This company classifies its consumers into three classes according to their previous loan records and credit history in the company: gold, silver, and bronze. The company provides different classes of consumer with different technical quality of service (QoS) guarantees, prices per year, penalties if the guarantees are not met, and historical probabilities to meet the guarantees.

![Figure 1. A loan brokering Web service composition.](image)

The credit check service is a third-party service provided by an external credit check agency. During runtime, it suddenly becomes unavailable for some reason. The adaptation system finds two alternatives, service E and service F for the credit check service. Each alternative provides 2 different classes of service, with various guaranteed QoS, prices, penalties, and historical probabilities to meet the guarantees. It takes various times to setup these services.

One same adaptation action for all instances is not appropriate from the business viewpoint. In this example, gold and silver consumers are “big” consumers who may bring high profit to the company. If the adaptation system lets gold and silver consumers wait until the new credit check becomes available, there is a possibility of loss of valuable consumers. Gold and silver consumers have previous loan records and good credit history in the company. It is better to let them skip this external credit check service or go to an internal credit check service. So, different adaptations are needed for different Web service composition instances. In this case, classes of instance are determined by their current position in the Web service composition and their class of consumer. The number of currently running Web service composition instances in each class also affects which adaptations are selected. It changes during runtime as instances are started or stopped, so a class of instance can be adapted in a different way when the number of instances (in any class) changes.

| Table I. THE LIST OF ADAPTATION OPTIONS IN THIS EXAMPLE |
|---|---|---|---|---|
| Adaptation option | Adaptation description |
| APA1 | Use E-Business after E-Business is set up |
| APA2 | Use E-Economy after E-Economy is set up |
| APA3 | Use F-Business after F-Business is set up |
| APA4 | Use F-Economy after F-Economy is set up |
| APA5 | Skip the credit check service and go to bank selection service |
| APA6 | Go to an internal credit check service |
| APA7 | Roll back to the application lodgement service and start the process again using E-Business after E-Business is set up |
| APA8 | Roll back to the application lodgement service and start the process again using E-Economy after E-Economy is set up |
| APA9 | Roll back to the application lodgement service and start the process again using F-Business after F-Business is set up |
| APA10 | Roll back to the application lodgement service and start the process again using F-Economy after F-Economy is set up |
| APA11 | Do no adaptation |

Table I shows an example of the list of adaptation options and their descriptions. Note that not every adaptation can be used by every class of instance. In this context, various business tradeoffs are involved in the adaptation decision making.
To determine how to adapt, business value and cost of each adaptation option have to be calculated according to some policies. The selected adaptation options have to satisfy various constraints including cost limit constraints (in this example: $100 total cost) and other constraints. For example, availability of the chosen credit check replacement must not be less than the guaranteed availability for a particular class of service and response time of the chosen credit check replacement must be at least 5ms lower than the overall response time.

**TABLE II. AN EXAMPLE ARRAY OF SELECTED ADAPATIONS**

<table>
<thead>
<tr>
<th>Class of instance</th>
<th>Position in the Web service composition</th>
<th>Class of consumer</th>
<th>No. of instances</th>
<th>Selected adaptation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI1-G</td>
<td>1 - not yet come to the credit check service (incl. those that have not yet started execution)</td>
<td>Gold</td>
<td>1</td>
<td>APA1</td>
</tr>
<tr>
<td>CI2-G</td>
<td>2 - in the credit check service</td>
<td>Gold</td>
<td>2</td>
<td>APA5</td>
</tr>
<tr>
<td>CI3-G</td>
<td>3 - passed the credit check service</td>
<td>Gold</td>
<td>1</td>
<td>APA11</td>
</tr>
<tr>
<td>CI1-S</td>
<td>1 - not yet come to the credit check service (incl. those that have not yet started execution)</td>
<td>Silver</td>
<td>1</td>
<td>APA2</td>
</tr>
<tr>
<td>CI2-S</td>
<td>2 - in the credit check service</td>
<td>Silver</td>
<td>1</td>
<td>APA6</td>
</tr>
<tr>
<td>CI3-S</td>
<td>3 - passed the credit check service</td>
<td>Silver</td>
<td>2</td>
<td>APA11</td>
</tr>
<tr>
<td>CI1-B</td>
<td>1 - not yet come to the credit check service (incl. those that have not yet started execution)</td>
<td>Bronze</td>
<td>2</td>
<td>APA4</td>
</tr>
<tr>
<td>CI2-B</td>
<td>2 - in the credit check service</td>
<td>Bronze</td>
<td>2</td>
<td>APA10</td>
</tr>
<tr>
<td>CI3-B</td>
<td>3 - passed the credit check service</td>
<td>Bronze</td>
<td>2</td>
<td>APA11</td>
</tr>
</tbody>
</table>

Table II shows the globally optimal array of adaptations for different classes of instance in one example scenario. The goal of our research was to develop software that can automatically find such globally optimal solutions, in various operational circumstances. We used this example and many others to test and evaluate our work (cf. Section VII).

III. RELATED WORK

Since there have been many publications on particular types of adaptation of Web service compositions and business processes, in this section we will only point out the most relevant issues and position some representative related works. We provided a classification and a relatively detailed analysis of many additional related works in [6].

While in industry practice adaptation decision making is still mainly done by human administrators, the vast majority of research focuses on adaptation with minimal help from humans. However, the past adaptation decision making algorithms predominantly choose adaptation that maximizes technical metrics (e.g., [10][11]), while maximization of business metrics is still an open research area [5][6]. While [12] is not directly on adaptation of Web service compositions, some of its solutions could be reused in our context. It presented a system for maximization of business metrics that schedules the triggered management policies by minimizing the penalty specified in service level agreements (SLAs), but it did not examine resolution for conflicting policies, which is a critical problem in policy-based management.

Different adaptation triggers often require different monitoring, decision making, and adaptation execution. For example, if a management system does not monitor business-related events, it will not be able to recognize the adaptation needs they cause, so it will not be able to react and adapt appropriately. However, the vast majority of past works researching adaptation of Web service compositions (e.g., [10][13]) focused on adaptations triggered by technical reasons, while only a few research projects (e.g., [14][15]) examined the impact of business causes (but without addressing them completely). In particular, [14] identified three types of adaptation causes: exceptions, business policy changes, and business model changes. However, there are many other additional causes, such as infrastructure updates, business strategy changes, and business customer changes.

Adaptation of a Web service composition instance could change various aspects of system configuration and execution: structure of the implemented business process type (e.g., replacing one block of activities with another for all instances) [10], Web services used for the implementation (e.g., replacing faulty service) [16], execution of the instance (e.g., rolling back to a checkpoint) [17], or contracts with customers (e.g., changing SLAs) [18]. There is a lot of research literature presenting support for individual types of change, but a comprehensive support for all these types of change was first provided in the MASC middleware [19], which is the predecessor of the research presented in this paper.

Our work is different from traditional optimization problem in service composition. We focus on the decision making support for concurrent runtime adaptation of multiple Web service composition instances to maximize business value.

IV. PAST RESULTS BUILT UPON IN THIS WORK

A. WS-Policy4MASC

WS-Policy4MASC [5][8], our extension of the WS-Policy industry standard, is a policy language that can describe various adaptations and all information necessary for decision making. WS-Policy4MASC defines 5 types of WS-Policy policy assertions: 1) goal policy assertions (GPAs) prescribe conditions to be met (e.g., desired response time), 2) action policy assertions (APAs) list adaptation actions, 3) utility policy assertions (UPAs) contain business metrics for particular situations, 4) probability policy assertions (PPAs) specify probabilities of occurrence, and 5) meta-policy assertions (MPAs) describe which values are important for adaptation decisions.

WS-Policy4MASC enables specification of both financial and non-financial business value metrics (BVMs) in UPAs. In practice it can be difficult to monetize contributions of individual non-financial aspects. Therefore, WS-Policy4MASC defines BVM categories that are easier to estimate. There are 3 mutually orthogonal dimensions of properties that classify these BVM categories: i) financial or non-financial, ii) agreed or non-agreed (i.e., only possible), iii) benefit or cost. A BVM category is a combination of 1 characteristic from each of these 3 dimensions, so there are 8 BVM categories. For example, “non-financial non-agreed benefit” characterizes customer satisfaction, while “financial non-agreed benefit” characterizes estimates of future sales. A number of UPAs can correspond to consequences of executing a particular APA. WS-Policy4MASC enables dealing with uncertainty through PPAs.
It is also possible to assign probabilities that different estimates of the same business value metric will be correct. Each adaptation approach is modeled as an APA. When several APAs can be applied in the same situation, but should not be applied at the same time, there is a policy conflict. WS-Policy4MASC enables specification of business strategies in MPAs, to guide deciding which among alternative adaptation approaches to take in case of policy conflict.

B. MiniMASC and Business-Driven Adaptation Decision Algorithms It Implements

Our MiniMASC [6][7] middleware is a comprehensive framework for autonomic adaptation of Web service compositions. It is the next generation of the MASC middleware [19]. We designed, implemented, and tested MiniMASC as a relatively simple, light weight, modular and extensible middleware framework for diverse autonomic adaptations of Web service compositions. Note that although the focus of this paper is on the autonomic BDIM support, MiniMASC can be also used for traditional decision making that maximizes technical metrics. This is primarily because WS-Policy4MASC can describe various adaptations and all information necessary for decision making of all types of adaptations for Web service compositions. Our evaluation using several prototype implementations showed that the proposed MiniMASC architecture and autonomic BDIM decision-making algorithms are feasible and easy to modify/extend. Furthermore, our performance and scalability tests showed that MiniMASC does not introduce unforeseen performance or scalability problems.

The triggering of decision making algorithm in MiniMASC is caused by events (e.g., not meeting a GPA - -- his relates technical and business metrics). The algorithm first finds all the APAs triggered by this event. If the number of triggered events is more than 1, the Policy Conflict Resolution module of MiniMASC decides APA selection. In the Policy Conflict Resolution module, our basic BDIM conflict resolution algorithm first loops through all conflicting APAs to check whether each of them satisfies all constraints and, if yes, to calculate sum of all BVMs for this APAs. APAs that satisfy all constraints are added to the list, along with their summary BVM. If none of the conflicting APAs satisfies the constraints, the resulting list is empty and an exception is thrown. The APAs in the list are ordered based in the decreasing value of their summary BVM, so the first APA in the list has the highest summary BVM while satisfying all constraints. This APA is returned as the adaptation action to be executed.

For the BVM summation algorithm, the first step is to match all UPAs in the Policy Repository to find those that are relevant for the currently examined APA (which could contain several sub-actions). The BVMs in WS-Policy4MASC UPAs are given in the form of arithmetic-with-currency-unit expressions (possibly with variables), so the next step of the algorithm calculates (for all BVMs in all relevant UPAs) the monetary amounts resulting from these expressions and converts them into a common currency (using an external currency conversion service). Then, the algorithm checks whether use of probabilities is enabled. If it is, each of the BVMs (in all relevant UPAs) is multiplied with 2 probabilities: the likelihood that the given values are correct estimates (here, probabilities for all compatible UPAs must add to 100%). At the end, the algorithm determines the summary result that is returned. However, this is not a simple sum, but a data structure with 8 values, each summing BVMs of the same category. This is because in different circumstances (e.g., business strategies), different business value metric categories are relevant for decision making. This was explained in detail in [5][8].

C. Constraint Programming and MiniZinc

In constraint programming (CP) [20], a human modeler identifies constraints for the problem variables and the solving software finds feasible and optimal solution(s) satisfying all these constraints. The solving software first finds a feasible solution and adds a constraint that the future solution must be better than the current feasible solution. It then loops until it cannot find a better solution. The optimal solution will be the last solution found.

We selected constraint programming as the optimization method to find optimal concurrent adaptations of multiple Web service compositions due to 3 reasons. First, this problem can be defined as a (not too complex) constraint satisfaction optimization problem. Different adaptation decisions can be defined as problem variables. Available adaptation alternatives can be defined as possible values for problem variables in domains. Business strategies, the current service composition situation, the running instances that require adaptation, and resource limitations can be all defined as constraints. Finding an adaptation alternative providing highest business value is the objective function for constraint programming. Second, constraint programming represents the problem much more directly and flexibly than mathematical programming. Compared to mathematical programming, the problem models in constraint programming are relatively easy to understand. Third, constraint programming supports different kinds of constraints [20] important in modeling our problem domain.

MiniZinc [9] is a powerful constraint programming specification language, which has a free efficient constraint solving software. It has been used to solve industry optimization problems. Nethercote et al. [7] claimed and our team checked [9] that, compared to the other constraint programming languages and solvers, the main advantages of MiniZinc are: efficient and reliable solver support for the language, appropriate scope of the language and solver for our domain, separation of model and data, easily formatted input, clear and concise output, good documentation, and growing user base. MiniZinc solver supports a variety of solving methods and chooses the most appropriate one [7]. We trust that the MiniZinc solver always finds globally optimal solutions and additional extensive evaluation of this assumption is outside of scope of our research.

V. Modeling Adaptation Decision Making in Constraint Programming

In this section, we represent our decision-making model in constraint programming. This model is applied in the optimization algorithm of the Policy Conflict Resolution module of MiniZinc, discussed in the next section. When changes happen, the algorithm will concurrently choose 1 adaption
option (WS-Policy4MASC APA) for each Web service composition instance, from a set of applicable adaptation options.

We first introduce our notation. \( N \) specifies the number of classes of instance, across all supported business process types (and, thus, across different Web service compositions). \( X_n \) \((n=1, \ldots, N)\) represents the current number of running instances in class \( n \). \( M_n \) \((n=1, \ldots, N)\) represents the number of adaptation options for an instance in class \( n \). \( A_{n,i} \) \((n=1, \ldots, N; i=1, \ldots, M_n)\) represents the \( i \)-th adaptation option for class \( n \). \( K \) represents the number of business value metric categories that can be reasoned about (e.g., WS-Policy4MASC has 8 built-in BVM categories, but enables definition of additional ones). \( V_{k,n,i} \) \((k=1, \ldots, K; n=1, \ldots, N; i=1, \ldots, M_n)\) represents the summary business value of all used BVM categories for adaptation option \( A_{n,i} \). It is calculated by our algorithm for calculations of business metrics, described in our previous publications [5][6] and summarized in the previous section. The algorithm is implemented in MiniZinc as Java code, so in this constraint programming model we just use its result \( V_{k,n,i} \). (We derived a precise mathematical formula describing how \( V_{k,n,i} \) is calculated, but there is no need to show it here). \( W_k \) \((k=1, \ldots, K)\) represents the weight of BVM category \( k \). Weights are usually in the interval \([-1,1]\) (negative values are for costs, but can have other values. For example, a weight could be -1.2 to emphasize a particularly important category of costs. Weights are specified in the applied meta-policy (WS-Policy4MASC MPA).

\[
B_{n,i} = \sum_{k \in \text{UsedBVTs}} (W_{k,n,i}) \quad (n=1, \ldots, N; i=1, \ldots, M_n),
\]

represents the summary business value of all used BVM categories for adaptation option \( A_{n,i} \). The set \( \text{UsedBVTs} \) contains all BVM categories that represent relevant costs. It is specified in the applied meta-policy, which enables that it can differ across change situations (e.g., while the policy conflict resolution algorithm in MiniMASC [5][6] supported the concept of tiebreaking, there is no tiebreaking in the presented constraint programming model. It is possible to express tiebreaking using advanced constraint programming, but the resulting model is very complex and hard to understand. Thus, we decided not to add tiebreaking. The problem of finding the globally optimal array of adaptations can be represented in constraint programming as the task to find the array of \( N \) adaptations \( j_n \) \((n=1, \ldots, N)\), \( j_n \in \{ A_{n,1}, \ldots, A_{n,M_n} \} \) (i.e., 1 adaptation for each class of instance), to maximize \( S = \sum_{i=1}^{N} (X_i B_{n,i}) \) and satisfy additional constraints. These additional constraints can be of different types (MiniZinc provides various operators and WS-Policy4MASC can provide various needed information in policy assertions), but we emphasize cost limits and resource limits, as probably the most important in practice. Cost limits describe common business situations when an adaptation option (although potentially beneficial in the long run) should be discarded because its short-term costs are higher than available funds. They are modeled similarly to: \( \sum_{n=1}^{N} (X_n \sum_{i \in \text{UsedBVTs}} W_{i,n,i}) \leq \text{CostLimit} \). Here, the set \( \text{CostBVTs} \) contains all BVM categories that represent relevant costs. It is specified in the applied meta-policy. Another important group of constraints are resource limits. Resources (e.g., memory, processor time, bandwidth, energy, etc.) are often limited and this is one of the main reasons why adaptations decisions of concurrent instances should be considered together instead of separately. The newest version of WS-Policy4MASC enables definition of non-financial business value types that represent resource usage and use appropriate units. For example, it is possible to define that all non-financial (agreed or possible) costs with attribute \( \text{Cause} = \text{"Memory"} \) represent memory usage and have unit “GB”. Then, a memory limit can be modeled similarly to: \( \sum_{n=1}^{N} (X_n \sum_{i \in \text{MemoryBVTs}} W_{i,n,i}) \leq \text{MemoryLimit} \). The set \( \text{MemoryBVTs} \) contains all BVM categories that represent memory usage.

While the policy conflict resolution algorithm in MiniMASC [5][6] supported the concept of tiebreaking, there is no tiebreaking in the presented constraint programming model. It is possible to express tiebreaking using advanced constraint programming, but the resulting model is very complex and hard to understand. Thus, we decided not to add tiebreaking.

VI. ARCHITECTURE OF MINIZNMSC

Fig. 2 shows the architecture of our autonomic management system for Web service compositions. There are 3 groups of components and artifacts: file creation components,

![Figure 2. Architecture of our management system for Web service compositions](image-url)
MiniZnMASC, and the extended systems (incl. the managed system). This means that MiniZnMASC is a part of a broader management system that contains runtime monitoring, runtime execution of control (adaptation) actions, as well as design-time activities such as modeling and re-design.

The colored center of Fig. 2 represents MiniZnMASC, the main part of our autonomic management system for Web service compositions. It decides adaptation actions from the business viewpoint, based on monitoring data about execution of the managed Web service compositions and according to policies in WS-Policy4MASC [5][8].

The most important part of MiniZnMASC is the Policy Conflict Resolution module, which implements the adaptation decision algorithm that decides whether the triggered policy assertions conflict and, if they do, which of them should be executed. Our business-driven decision making model in constraint programming (cf. Section V) is implemented in the MiniZinc language and applied in the Policy Conflict Resolution module. The MiniZinc solver (which we used as a black box) located in the Selection among Alternative sub-module computes the results and makes runtime adaptation decisions. To select the globally optimal array of adaptations, the MiniZinc solver populates the given MiniZinc model with 2 types of runtime data: 1) total business values (incl. non-financial values modeling resource constraints) and total costs for particular adaptation options, 2) the current number of running instances of each class of instance. The first type of runtime data is sent by the Calculation of Business Metrics sub-module, which uses the algorithm outlined in Section IV.B and policy assertions stored in the Policy Repository. The second type of runtime data is stored in the Database of Monitored Data, based on notifications from external monitoring modules about starting/stopping of instances.

We conducted performance and scalability tests to roughly compare execution times in various combinations of MiniMASC and MiniZinc to decide how to integrate MiniMASC with MiniZinc. Since MiniZinc is much slower than Java in calculations of business value metrics, we have adopted the architecture in which calculation of business value metrics is done by Java code, while MiniZinc models deal with complex selections. This is because Java code gives speed, while MiniZinc models give flexibility for handling complex decision cases for which Java code in MiniMASC becomes much more complicated and harder to maintain. Another important benefit of such integration is that it significantly simplifies MiniZinc models, which now depend on the content of WS-Policy4MASC files only to a limited extent because policy conflict situations have common characteristics captured in a generic model we developed. These simpler MiniZinc models (based on the generic model) require less MiniZinc expertise from software engineers who develop them. Further, the impact of future changes to MiniZinc is minimized in this way.

In MiniZnMASC, the information guiding adaptation decisions is specified as policies in WS-Policy4MASC. WS-PolicyAttachment defines how to associate a policy with subjects to which it applies. WS-Policy4MASC policies are passed by the Policy Parsing module and then stored in the Policy Repository module. Database of Monitored Data stores runtime data about monitored technical metrics (e.g., measured response time, calculated availability), business metrics (e.g., paid prices and penalties), and events that come from the external monitoring modules, after this data is handled by the Processing of Monitored Data module. Based on the recent monitored data (particularly events) and historical information stored in the Database of Monitored Data, the Determining Triggered Policies module determines which WS-Policy4MASC policy assertions should be executed.

On the left-hand side of Fig. 2 are optional components and artifacts in our system. Their main purpose is to help users in creation of WS-Policy4MASC files, WS-PolicyAttachment files, and MiniZinc model files. Suleiman has previously developed software [21] that semi-automatically creates WS-Policy4MASC files from UML (Unified Modeling Language) models annotated with original stereotypes compatible with WS-Policy4MASC policies. The same work also enabled annotation of monitored data on such extended UML models, to facilitate decisions about design-time evolution of the managed systems. Models in MiniZinc are currently developed manually by software engineers who know MiniZinc, but in the future we will add modules for their (semi-)automatic generation from extended UML models.

On the right-hand side are components that MiniZnMASC system extends and integrates with, without re-implementing them: the executing managed system, third-party modules for monitoring the managed system, and third-party modules for executing chosen adaptation actions. The managed system can be a Web service or Web service composition system. MiniZnMASC receives events and monitored data from the external monitoring modules and sends adaptation decisions to the external adaptation execution modules. We assumed the existence of third-party modules for monitoring managed systems and for execution of chosen adaptation actions. Many related works (including MASC [19]) provided support for some monitoring and adaptation of Web service compositions, so we decided not to duplicate this functionality.

VII. EVALUATION
We evaluated MiniZnMASC and the developed algorithms for business-driven decision making in 5 aspects: feasibility, functional correctness, business benefits, performance overhead, and scalability. We evaluated feasibility through implementation of several proof-of-concept prototypes. Apart from the MiniZnMASC prototypes in Java and using the Postgres SQL database, we re-implemented some modules in C# to check that feasibility does not depend on Java features. We found no problems with feasibility. We implemented the motivating example from Section II and evaluated the functional correctness of MiniZnMASC by comparing the results calculated by MiniZnMASC and by hand. We also developed several other examples for this evaluation. The results show that MiniZnMASC was built correctly.

As mentioned previously, we trust that the MiniZinc solver always finds globally optimal solutions. If this assumption is correct, the solutions found by MiniZnMASC are always better or equal (from the business viewpoint), compared to the other possible solutions. To verify this, we extended the tests of functional correctness to calculate the total business value
For the performance and scalability tests, we used a Hewlett-Packard laptop model HP EliteBook6930p with Intel Core Duo CPU T900 2.53GHz processor and 4.0 GB of RAM memory, running 32-bit Windows Vista operating system. We first measured the performance with increasing number of conflicting APAs and BVMs to be summed. We started with 2 APAs, then increased to 10, and then to 100. We also increased from 2 BVMs, over 32 BVMs, to 64 BVMs. Using the Java “System.currentTimeMillis()” call, we measured the execution time of decision making in MiniMASC and MiniZnMASC. We repeated 100s of tests at different times of day and averaged their results. Table IV shows the distribution ranges (min, max) and averages of measured execution time. In the performed tests, MiniZnMASC was 0.4%-10% slower than pure MiniMASC. The overall execution time of decision making in MiniMASC and MiniZnMASC increases with the number of conflicting APAs and BVMs because the execution time of the summation of BVMs for each conflicting APA increases. The last test case (100 conflicting APAs with 64 BVMs each) is not realistic, so 4.5 sec is not an issue. It is important to note that in realistic application scenarios of MiniZnMASC the number of conflicting APAs will be low (typically 2, maybe a few more), so we did not perform detailed additional tests with much higher number of conflicting APAs. On the other hand, the number of additional non-conflicting APAs in Policy Repository can be huge. Therefore, we also checked that the number of non-conflicting APAs (in 100s) has practically no effect on performance. Additional test results for MiniMASC were given in [6].

Fig. 3 show the performance measurement results for MiniZnMASC with increasing number of classes of instance (up to unrealistic 100). Execution time of both the policy conflict resolution algorithm and the overall decision making increased almost linearly, showing good scalability. We also scaled the number of conflicting APAs (1 to 7) and the number of running instances in a class (1 to 10), but the impact on performance was small compared to the impact of scaling the number of classes of instance. This also shows that most of the execution time of decision making in MiniZnMASC is spent on deciding triggered APAs and finding matching UPA’s and PPA’s for them, while the policy conflict resolution is quick.
development and exploration of various decision-making algorithms and corresponding middleware are among the most important open topics related to autonomic BDIM.

This paper presented our business-driven decision making algorithms and other solutions built into the MiniZnMASC middleware, with the focus on addressing situations when multiple running instances of Web service compositions should be adapted at the same time. These solutions build upon our previous work on the WS-Policy4MASC language for specification of necessary information, the adaptation decision making algorithms finding optimal adaptation for each Web service composition instance separately, and the MiniMASC middleware implementing these algorithms. Our new decision making algorithms can concurrently make different runtime adaptation decisions for different classes of instance in a way that achieves maximal total business value while satisfying all given constraints. The new algorithms use descriptions in the MiniZinc constraint programming language and their implementation uses the MiniZinc solver as a part of the Policy Conflict Resolution module of MiniZnMASC. This enabled efficient handling of complex situations that were not addressed satisfactorily by past research. To maximize performance, we adopted the architecture in which calculation of business value metrics is done by Java code previously built into MiniMASC, while MiniZinc models deals with complex selections. Our MiniZnMASC prototype was implemented using Java, the PostgreSQL database, and the MiniZinc solver. We found no problems with feasibility of implementing our solutions. Our functional correctness and business benefit test results showed that our MiniZinc models were built correctly. According to our performance test results, the extensions built into MiniZnMASC have relatively low overhead and satisfactory (linear) scalability within the range of foreseen uses.

Although the focus of this paper was on the support for business-driven decision making with minimal human intervention, MiniZnMASC can be also used for traditional decision making that maximizes technical metrics. This is because WS-Policy4MASC can describe various adaptations and all information necessary for decision making of all types of adaptations for Web service compositions. While we designed MiniZnMASC middleware primarily for adaptation of Web service compositions, it can be also used (sometimes with minor extensions) in adaptation decision making for other types of IT systems. Thus, our ongoing work focuses on application of MiniZnMASC solutions to several other types of complex distributed systems. We are also exploring whether there are realistic usage scenarios for allowing that Web service composition instances from the same class can be adapted in different ways. We already know how to support this, if it is needed.

ACKNOWLEDGMENT
We thank Gareth Edwards and Vern Yao Tee for their contributions to implementation of the MiniMASC prototype. We also thank Dr Jacky Keung, Xiwei Xu, and Dr Juan Li for their helpful suggestions and valuable comments on this paper.

REFERENCES