A Generative Configuration Framework for Service Process Composition[1]

Rajesh Thiagarajan and **Wolfgang Mayer** and **Markus Stumptner**[2]

Abstract. In our prior work we showed the benefits of formulating service composition as a Generative Constraint Satisfaction Problem (GCSP), where available services and composition problems are modeled in a generic manner and are instantiated on-the-fly during the solving process, in dynamic composition scenarios. In this paper, we (1) outline the salient features of our framework, (2) present extensions to our framework in the form of process-level invariants, and (3) share promising results of our experiments to evaluate the effectiveness of our framework in scenarios with varying difficulty.

1 Introduction

A vast number of proposals that exploit formal specifications of individual services to automatically select and compose individual services into executable service processes have been brought forward [17, 9, 7, 2, 11, 8]. While most frameworks can successfully address basic composition tasks, many are based on ad-hoc algorithms or lack a precise representation of a service's capabilities. Therefore, the problem of configuring and tailoring the software that implements a given service is often left aside.

Constraint satisfaction based configuration techniques have been proposed as an alternative to address these challenges in service composition scenarios, where both type and instance information of services and relevant data must be considered [6, 18, 5, 1]. Existing models compose services by treating services as components and assembling them. However, standard CSPs are insufficient to model configuration problems where the number of components to be configured is unknown. Existing CSP-based composition techniques handle this by pre-specifying the number or type of services to be composed. In general, such pre-specification is difficult to make since domain specific knowledge that is problem specific is not available apriori.

To address this gap, we present a generative consistency-based service composition approach that addresses these challenges [12]. We extend models that have been successfully applied to model and configure complex systems [19], to the software domain. The service composition problem is posed as a configuration task, where a set of service components and their interconnections are sought in order to satisfy a given goal. Our framework is based on a declarative constraint language to express user requirements, process constraints, and service profiles on a conceptual and also on the instance level. We show that our meta-level constraint specifications permit to concisely represent conceptual information about processes, services and data structures involved in the composition. One of the major benefits of our approach is the provision to define problems independent of the number and type of services required, thereby overcoming the disadvantage of other models.

In this paper, we briefly outline our formalism that offers uniform (generic) constructs to represent service capabilities and semantics, represent data- and control flow between services (Section 2). We also present our extensions to the original model to unambiguously correlate conversations involving service instances or different parts of a workflow in a composition (Section 3). We also present the results from our experiments in which we evaluated our framework's effectiveness to handle non-trivial pessimistic scenarios, where service compositions often fail (Section 4). Experimental results indicate that even in such pessimistic settings our framework's performance is quite competitive to other composition systems.

2 GCSP-based Service Composition

Constraint Satisfaction Problem (CSP)[3] is a successful problem solving technique applied in various domains including general planning [15, 10] and service composition [6, 18, 5, 1]. Generative CSPs (GCSPs) extend standard CSPs by lifting constraints and variables to a meta-level, where *generic constraints* are the primary modeling element. Generic constraints abstract the actual CSP variables into so-called *meta variables* that are instantiated into ordinary constraints over variables in a classical CSP during configuration. Generative configuration can be seen as the problem of incrementally extending a CSP network until all the generic constraints are satisfied. In our formalism, when a component (C) of a particular type is added to the configuration, related variables and constraints (instances of the generic constraints of C) are activated in the configuration process. This dynamic activation of CSP variables makes GCSPs suitable for dynamic situations where a priori prediction of the problem structure is difficult.

A GCSP is characterized by the set of available component types, their attributes and ports (to connect to other compo-

[1] This work was partially supported by the Australian Research Council (ARC) under grant DP0988961.

[2] Advanced Computing Research Centre, University of South Australia, {cisrkt,mayer,mst}@cs.unisa.edu.au

[3] A CSP consists of a finite set of variables, set of values that each variable can be assigned to (the variable's *domain*), and a set of constraints restricting the valid assignments of values to variables.

nents), and a set of generic constraints. In generic constraints, meta-variables act as placeholders for component variables. Generic constraints can be seen as prototypical constraints on a meta-level that are instantiated into *ordinary* constraints over variables in the CSP. For example, assume a generic constraint $X \sqsubseteq BudgetShipper \Rightarrow X.price < 1000$ is given, stating the invariant that the value of attribute *price* for any data object of type *BudgetShipper* must be less than 1000. A generic constraint over a meta-variable is said to be consistent if and only if the constraint is satisfied for all instances over active CSP variables.

In [12], we extend the generative formalism in [4, 3] with the following features.

- We introduce connection components that act as connectors between services. The explicit representation of connectors provides a uniform interface contract between services and also serves as a means to model the provider-consumer relationship between services.
- A connection component also holds a representation of the data values that may be passed along the connection.
- To capture their semantics, we treat complex data objects as components. This facilitates the uniform handling of service and data components in the configuration, and has the additional benefit that generic constraints can be used to impose invariants on data structures throughout a configuration.
- We introduce non-local process level constraints to model data flow, control flow, and structural invariants of service processes.

Figure 2 exemplifies the structure of a service model in our framework. Individual service component and connectors are depicted as rectangular boxes, while data elements are represented as ovals. In order to facilitate reasoning about data objects attached to connections between service components, the connections are also modeled as components. Component constraints are formulated in terms of the local service component, the connection components that are directly connected to its ports, and the data object associated with the connection. Thus, the specification of a service is decoupled from the specifications of the neighboring services. Each component can have a number of attributes of primitive or user-defined data type. Data objects are represented as a tree structure comprised of the object's attributes. Our formalism relies on the well-known "dot notation" to navigate from parent components and attributes to their children. Thus, the framework provides a uniform notation for modeling service components, connections, and complex data objects.

The problem domain is defined by an ontology of component and data types, where subtype relationships and attributes of each component and data type are described. Each type is associated with constraints that govern the permissible values of each attribute and port connection [3]. This object-oriented notation facilitates the concise representation of complex domain concepts and related integrity constraints for both service components and data types. The ontology can be formalized as constraints in the GCSP formalism. Figure 2b shows an example of a constraint that describes the *Shipper* service type (an abstract service type used in the case study described in Section 4). In the formalism, meta-variables denote placeholders for variables in a CSP representation of a (partially completed) composition problem. Here, the X may refer to a CSP variable that represents a service component; it is constrained to a variable that has been assigned an instance of a *Shipper* service (or a subtype thereof). The antecedent of the example generic constraint further introduces local names of the CSP variables that represent the components connected to the Shipper's *sin* and *sout* ports. The consequent of the generic constraint enforces that both connected components must represent a data connection (component type *DataConn*), whose object payload must be an instance of type *Product* (or a subtype thereof). Equality constraints between the input and output object's *type* attribute ensures that the product (type) remains unchanged by executing the service. Furthermore, the constraint relates the component's attributes *from* and *to* to the input and output object's *location* attribute in order to ensure that the Shipper service only applies to products in appropriate locations (Shipper services in our example are limited to ship to limited destinations). Similar generic constraints would be created for any other type in the ontology. The resulting GCSP model allows us to concisely specify problems whose size may vary based on the initial configuration and goal (hence incorporating aspects of *dynamic CSPs* [16]) and whose structure may depend on partial value assignments (hence incorporating aspects of *conditional CSPs* [16]).

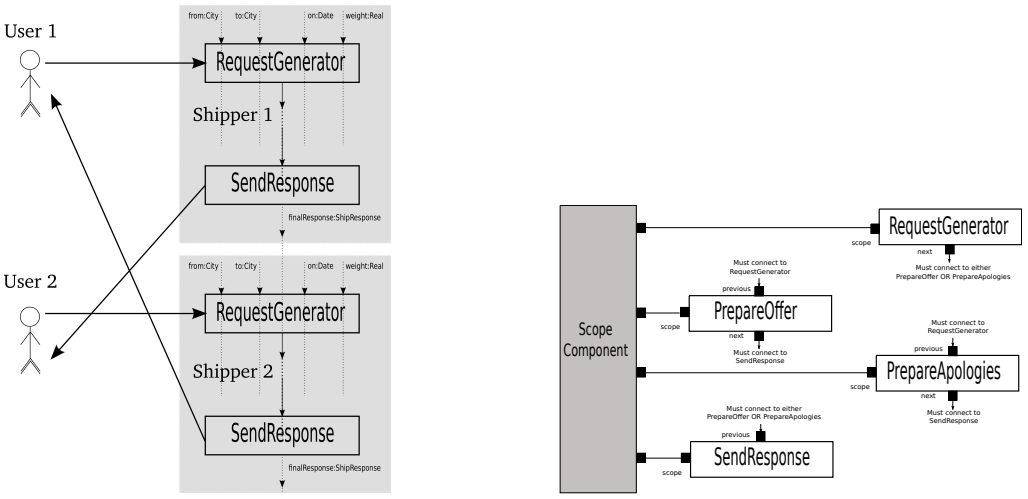
In our approach, a service composition problem is posed a as a configuration task, which is expressed by an initial set of components and constraints (denoted as R_1) that must be satisfied. During the configuration process, R_n is dynamically extended by adding new variables and constraints. After each extension, R_{n+1} represents a standard CSP constraint network (without generic constraints); therefore, standard algorithms can be applied to solve the CSP. In our approach, an iterative deepening strategy that limits the number of components that may be introduced in the configuration prohibits the network from expanding indefinitely. Once that limit has been reached, the algorithm backtracks and attempts to find an alternative solutions. If that fails, the limit is increased and the search is restarted. For a detailed explanation see [12].

3 Workflow Scope

The scope of a workflow or a sub-workflow defines the tasks it encapsulates in its behavioral process specification. For example, the Shipping process from the *Producer-Shipper* composition problem [14] encapsulates tasks *RequestGenerator* and *SendResponse* to process a request and acknowledge it, respectively. Existing specifications are insufficient if multiple instances of a sub-workflow exist within a composition. For example consider the process model in Figure 1a, where two users interact with two instances of the Shipping workflow[4]. The users would like to place a shipping order using the available shipping process, but are oblivious of each other. From a composition point-of-view, components in each subprocess are interchangeable. For example, the offer requested by *User 1* to *Shipper 1* may actually be sent to *User 2* (as in Figure 1a). Hence, means to ensure messages are directed to correct recipient must be provided.

To address this problem, we introduce explicit workflow *scope components* in our generic framework. Scope components, in addition to encapsulating a process specification, also

[4] We consider user interaction as an explicit part of the composition.



(a) Multiple Instances Problem

(b) Workflow Scope

Figure 1: Process Specifications

differentiate between multiple instances of the same workflow by maintaining a *session ID* that is unique for each occurrence of the workflow. Figure 1b shows the scope component that encapsulates the Shipping process in our example. Formally, scope components and their connections to their process elements are also defined using generic constraints like other components in our framework. Connections between scope component and the process elements are also defined as generic constraints. While each workflow instance is identified by a unique ID attribute. Additionally, each constituent component in a composition is further distinguished by its unique component ID.

4 Experimental Evaluation

We conducted an empirical investigation in order to assess the performance of our framework in different problem contexts. We investigated the time required to solve two parametric service composition problems. Our investigation were directed at assessing properties of the composition approach after potential candidates had been identified. To side-step the service matching problem, we generated a set of service instances for each service type prior to conducting our experiments. We varied the complexity of the search problem by varying the number of matching service candidates.

Our first evaluation was conducted on a generalized version of the well-known *Producer-Shipper* problem [13], where a product ordering and shipping process must be composed from individual producers and shipper services, considering the capabilities and restrictions of individual services. In this problem, a service process that consists of interactions between a producer service process and a shipper service process are to be synthesized to achieve the goal of delivering an item of particular size and cost to a given location. The producer and shipper processes themselves are composite service processes comprised of a number of tasks. Negotiation tasks between the user and both service processes must be taken into account, and the data and control flow representing the different steps and messages flowing between services must be synthesized to obtain an executable process.

Since the original problem description in [13] has only few components and can be solved almost instantaneously in our framework, we generalized the problem from a single shipping process instance to a parametrized problem that allows multiple concurrent instances. We also introduced additional requirements that ensure that only particular combinations of producer instances and shipper services may work together in a shipping process to obtain a more realistic scenario, where only few of a large number of alternative components will lead to a successful composition. These constraints link producers and shippers in different copies of the process, which can no longer be solved independently.

Our largest problem with 28 parallel producer-shipper processes (1400 services) can be solved in roughly 3 minutes. This result is quite competitive with other approaches.

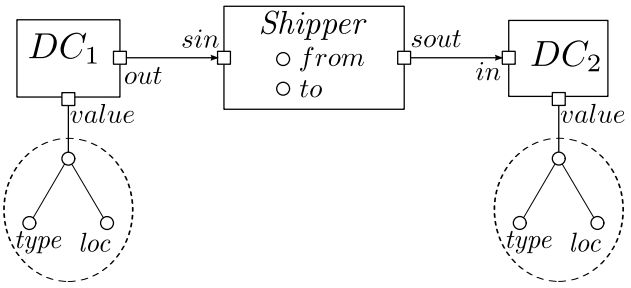
4.1 Supply Chain Coordination

The a Producer-Shipper process is quite atypical in that is does not include complex chains of services or non-trivial transformations of data exchanged between services. Hence, we conducted further experiments to assess the performance of our framework in a scenario where items must be processed by a sequence of services in order to meet the goal requirements.

Assume the supply chain of a pie factory (at a fixed location) is to be configured using services. The factory requires supplies of flour and sugar to produce pies. Our model includes wheat and sugarcane farming *services* located in various Indian cities. In addition to the factory and farmers, the problem domain includes flour and sugar mills located in a number of South-East Asian destinations. The model also includes shipping services that ship products between locations. Goods are modeled as a composite data structure containing name, quantity, and location. Each service has input and output ports that model the goods received and supplied by the service. Shipping services update the location property of goods to refer to the destination location. The aim is to compute a composition utilizing available services (farmers, mills, and shippers) in order to facilitate production operations in the pie factory while minimizing handling of the products (minimal number

of shippers).

Generic constraints permits us to model services precisely as shown in Figure 2.



(a) Shipper Service

$$X \sqsubseteq Shipper \wedge X.sin = DC_1 \wedge X.sout = DC_2 \Rightarrow$$
$$DC_1 \sqsubseteq DataConn \wedge DC_1 2 \sqsubseteq DataConn \wedge$$
$$DC_1.value \sqsubseteq Product \wedge DC_2.value \sqsubseteq Product \wedge$$
$$DC_1.value.type = DC_2.value.type \wedge$$
$$DC_1.value.loc = X.from \wedge$$
$$DC_2.value.loc = X.to.$$

(b) Shipper Generic Constraint

Figure 2: Shipping Service Specification

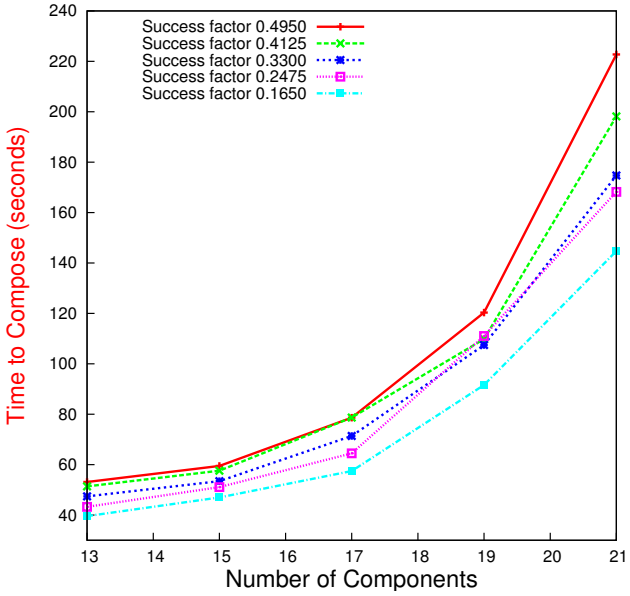
The problem structure in our scenario requires our solver to explore different alternative matching services, and for each alternative, longer chains of prerequisite services must be devised. That extensive exploration is required at each choice point is relevant to the web service composition context, where number of (similar) services can be quite high. The following experiment aims to quantify effects of the changed search behavior on our framework.

4.2 Experimental Setup and Results

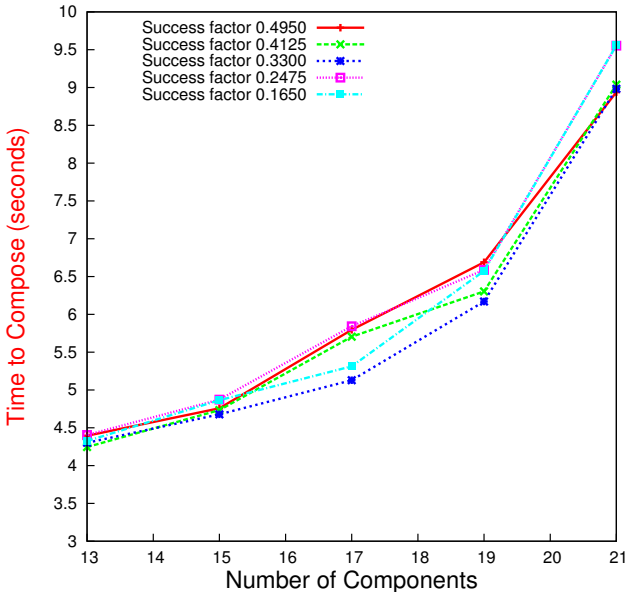
We conducted experiments on a number of variants of our model. All our models contain 1 pie factory (the goal), 2 mills, and 12 farming locations. The smallest model involves choosing among 25 shipping locations. We gradually increase the number of shipping locations in each of our models; the largest model consists of 45 locations. Our smallest variant results in a composition with 13 services and our largest variant requires 21 services. The smaller model was obtained by introducing more direct shippers between service locations, while the model size was increased by removing direct shippers and introducing additional in-direct shipping routes.

We introduce another parameter *success factor* to vary the complexity of the problem based on the availability of matching services. Consider the coordination process as a graph exploration task, where we seek to reach nodes representing farmers from the pie factory node. In such a setting, the success factor of a model is the average probability of reaching the farmer nodes through an appropriate mill node. That is, the higher the success factor is the more likely exploring an arbitrary branch will result in a successful match. A model with success factor 1 implies that exploring any out-going branch of the factory will definitely lead us to a matching services. The Producer-Shipper scenario model has a success factor close to 1 and hence exhibits near linear performance [12]. In this experiment we only consider models with success factors below 0.5 to evaluate the performance in cases that are more likely to fail. This setting reflects the service composition assumption that many candidates that provide similar services exist, but only a few of those will be suitable. We vary the success factors of our models by replacing suitable services with ones that share similar profiles, but do not offer the capabilities required for a composition goal.

The first set of experiments employed the iterative deepening strategy discussed in [12]. The limit on the number of services in each experiment was initialized to 1 and was incremented by 1 in each iterative deepening step.



(a) Composing by Iterative Deepening



(b) Composing by Estimation

Figure 3: Pie Factory Problem Results

The results are shown in Figure 3a. Our smallest problem with 13 services can be solved in roughly 45 seconds, and our largest problem with 21 services requires roughly 3 minutes. This result is comparable in terms of problem size and complexity, and is quite competitive to other composition systems [1, 13]. Note that the time required to find a solution is depends predominantly on the size of the problem. The success factor has only minor influence.

While the approach exhibits competitive performance and supports process-level compositions (unlike [7]), the results in Figure 3a seem counter-intuitive as our results indicate that the problem instance with smallest success factor is solved fastest. Intuitively one would expect the opposite, since the chance of making a wrong choice and backtracking is more frequent in models with low success factor. To analyze this observation, we conducted another set of experiments where the number of service required in a composition is approximated beforehand (no iterative deepening). The results are presented in Figure 3b. While the graphs in Figure 3 appear similar, the scale of the ordinate is different. It can be observed that the time taken to solve models with high success factor is lesser than that of model with low success factor. Therefore, backtracking caused by the iterative deepening strategy is the cause for the counter-intuitive observation in Figure 3a.

5 Conclusion

We outlined basic elements of our composition framework and showed that our formalism can represent component types, data types, and data flow constraints require in complex service composition scenarios. We presented process-specific constraints that enforce control and data flow properties within a restricted sub-scope of a larger process. We outlined an iterative approach to translate a composition problem formalized as Generic CSP problem into a series of classical CSP problems, and showed the practicality of our framework on two case studies. We obtained that our framework is competitive with other published service composition approaches and showed that our approach is robust with respect to the sparsity of solutions. We also showed that significant performance improvements can be achieved if additional information about the size of the solution is incorporated into the solving process.

We are currently exploring strategies to further improve scalability by avoiding to undo assignments in complete (successful) branches of the search space. Particularly, we are investigating the effect of replacing backtracking with a back jumping strategy that arbitrarily chooses a previous assignment to revise. Our current solving strategy is predominantly concerned with finding *a valid* solution comprised of a small number of components. Inferring the *optimal* solution for a configuration goal has played a subordinate role in our framework, since configurations comprising few components are often satisficing. Mechanisms to incorporate further guidance heuristics and optimization strategies into our process configuration framework are possible avenues for future research.

REFERENCES

[1] Patrick Albert, Laurent Henocque, and Mathias Kleiner, 'Configuration Based Workflow Composition', in *Proc. of ICWS 2005*, (2005).

[2] Matthias Born, Jörg Hoffmann, Tomasz Kaczmarek, Marek Kowalkiewicz, Ivan Markovic, James Scicluna, Ingo Weber, and Xuan Zhou, 'Semantic Annotation and Composition of Business Processes with Maestro', in *Proceedings of the European Semantic Web Conference (ESWC)*, (2008).

[3] Gerhard Fleischanderl et al., 'Configuring large-scale systems with generative constraint satisfaction', *IEEE Intelligent Systems*, **13**(4), (1998).

[4] Alois Haselböck and Markus Stumptner, 'An integrated approach for modelling complex configuration domains', in *Proc. of the 13th Int. Conf. on Expert Systems, AI, and Natural Language*, (1993).

[5] Ahlem Ben Hassine, Shigeo Matsubara, and Toru Ishida, 'A Constraint-Based Approach to Horizontal Web Service Composition.', in *Proc. ISWC*, (2006).

[6] E. Karakoc and P. Senkul, 'Composing semantic web services under constraints', *Expert Syst. Appl.*, **36**(8), 11021–11029, (2009).

[7] Freddy Lécué, Alexandre Delteil, and Alain Léger, 'Optimizing Causal Link Based Web Service Composition', in *ECAI*, (2008).

[8] Lei Li and Ian Horrocks, 'A software framework for matchmaking based on Semantic Web technology', in *Proc. of WWW*, pp. 331–339, Budapest, (2003).

[9] Zhen Liu, Anand Ranganathan, and Anton Riabov, 'A planning-based approach for the automated configuration of the enterprise service bus', in *Proc. ICSOC*, (2008).

[10] Adriana Lopez and Fahiem Bacchus, 'Generalizing GraphPlan by Formulating Planning as a CSP', in *Proc. IJCAI*, (2003).

[11] Annapaola Marconi, Marco Pistore, Piero Poccianti, and Paolo Traverso, 'Automated Web Service Composition at Work: the Amazon/MPS Case Study', in *Proc. of ICWS*, (2007).

[12] Wolfgang Mayer, Rajesh Thiagarajan, and Markus Stumptner, 'Service Composition As Generative Constraint Satisfaction', in *Proc. of ICWS*, (2009).

[13] Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso, 'Automated composition of web services by planning at the knowledge level.', in *Proc. IJCAI*, (2005).

[14] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and A. Marconi, 'Automated synthesis of composite BPEL4WS web services.', in *ICWS*, pp. 293–301, Orlando, (2005).

[15] Cédric Pralet and Gérard Verfaillie, 'Using constraint networks on timelines to model and solve planning and scheduling problems', in *Proc. ICAPS*, (2008).

[16] *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, Elsevier, 2006.

[17] Ignacio Silva-Lepe, Revathi Subramanian, Isabelle Rouvellou, Thomas A. Mikalsen, Judah Diament, and Arun Iyengar, 'Soalive service catalog: A simplified approach to describing, discovering and composing situational enterprise services', in *Proc. ICSOC*, (2008).

[18] Rajesh Thiagarajan and Markus Stumptner, 'Service Composition With Consistency-based Matchmaking: A CSP-based Approach', in *Proc. ECOWS*, (2007).

[19] Markus Zanker, Dietmar Jannach, Marius-Calin Silaghi, and Gerhard Friedrich, 'A distributed generative csp framework for multi-site product configuration', in *12th International Workshop on Cooperative Information Agents (CIA)*, (2008).