

Service Composition as Generative Constraint Satisfaction

Wolfgang Mayer Rajesh Thiagarajan Markus Stumptner
Advanced Computing Research Centre
University of South Australia
Adelaide, Australia
Email: {mayer,cisrkt,mst}@cs.unisa.edu.au

Abstract—The ability to build new (complex) services by composing existing services is one of the key benefits of the Service Oriented Architecture paradigm. Existing approaches to automate composition requires pre-planning or prediction of the number of required services, making them unsuitable in dynamic composition scenarios. To address this gap, we present a consistency-based service composition approach, where composition problems are modeled in a generative constraint-based formalism. We illustrate how the configuration of service processes differs from established constraint-based configuration techniques and develop an algorithm to synthesise valid service process compositions. We also show that our technique scales well to non-trivial problems.

I. INTRODUCTION

Semantic web technology has been widely employed in recent research attempting to automatically compose and validate complex interactions to address applications such as brokering, purchasing, supply chain integration, and assembly of scientific experiments and work flows. In the majority of cases, no single service is available that satisfies a specific requirement, and a complex execution must be constructed from individual services. Since service availability and goals may vary, automated discovery and composition of services for a given goal is desired.

Numerous approaches for automated service discovery and composition have been developed. While early proposals were mainly concerned with syntactic properties of service descriptions [1], more recently rich conceptual abstractions of static [2] and operational properties of individual services [3] and service processes [4], [5] have been utilized. However, attempts to combine complementary techniques into a uniform framework have so far been hampered by incompatible knowledge representation paradigms and ad-hoc inference algorithms.

Configuration formalisms based on Constraint Satisfaction Problems (CSPs) have been proposed in the context of service composition to address some of these challenges [4], [6], [7]. Existing models perform service composition 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 Constraint Satisfaction Problem(CSP)-based composition techniques handle this by pre-specifying

the number/type of services to be composed. But, in general, such pre-specification is infeasible because the number/type of services required to meet a requested is problem specific.

In this paper, we present a consistency-based service composition approach that serves as unified platform that is rich enough to capture a diverse set of service composition principles. Our framework therefore has the ability to serve as unifying platform in which multiple techniques can be integrated seamlessly.

We extend models that have been successfully applied for complex physical systems [8], [9] to the service processes. In our framework, 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 level and also on the instance level. 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 pre-specification disadvantage of the existing models.

By formalizing a service composition task as a constraint satisfaction problem, well-established heuristics and algorithms can be exploited to find a suitable composition (see [8] for a survey). Since our framework is largely independent of any concrete underlying representation language, compositions can be synthesized into a form suitable for execution, such as BPEL and OWL-S.

Our contributions can be summarized as follows:

- We present a declarative constraint-based mechanism for dynamic service process composition that simultaneously exploits essential properties of individual service and service processes in a single framework,
- we show how heterogeneous composition problems can be formalized as generative constraint satisfaction problems,
- we develop a domain-independent algorithm to efficiently find valid compositions, and
- we show that our framework scales to non-trivial problems while retaining the flexibility of generic configuration mechanisms.

This paper is organized as follows: We give a brief description of the GCSP framework for configuration in Section II, and

show how it can be extended to service composition in Section III. Our algorithm to find valid service compositions is presented in Section IV and evaluated empirically in Section V. We then compare our approach with related work (Section VI) before summarizing our contributions and current and further work.

II. GCSPs

As a problem solving technique in technical systems, the term *configuration* refers to the assembly of systems from smaller components. To automate this process, the available components and possible interconnections via ports must be formalized. It has been shown that this knowledge about a domain can be elegantly expressed as Constraint Satisfaction Problem [9]. We assume that the set of available components is known initially, but the size and structure of a particular solution is not.

A *Constraint Satisfaction Problem (CSP)* consists of a finite set of variables V and a finite set of constraints C . A constraint is a relation over a set of variables $V' \subseteq V$ that specifies the valid combinations of value assignments to these variables. The set of values which may be assigned to a variable is called the *domain* of that variable. Solving a constraint satisfaction problem means finding an assignment to all the variables in the network such that all constraints are satisfied. The main obstacle in using CSPs for configuration is the fact that the set of variables and constraints is static and thus cannot be used to describe problems where the structure or size of the solution is unknown.

Generic Constraint Satisfaction Problems (GCSPs) overcome this limitation by lifting constraints and variables to a meta-level, where *generic constraints* describe the valid CSPs instances. Generic constraints can be seen as constraint schema that drive the solving process by introducing fresh CSP variables and constraints as new components are added to the configuration. Therefore GCSPs can adapt the problem size and structure during the solving process based on the information contained in the partial solution.

A GCSP is characterized by the set of available component types, their attributes and ports, and a set of generic constraints: Let \mathcal{T} be the set of available component types and let A be the set of port and attribute names.

Definition 1 (GCSP) A GCSP is a tuple $\langle \mathcal{X}, \Gamma, \mathcal{C}, \mathcal{P} \rangle$, where \mathcal{X} is a set of meta variables, Γ is a set of generic constraints over \mathcal{X} ; \mathcal{C} is an infinite set of constraint variables representing component instances, and $\mathcal{P} = \{C.a \mid C \in \mathcal{C}, a \in A\}$ is an infinite set of constraint variables representing the components' attributes and ports. Each component variable $C \in \mathcal{C}$ may be assigned a value from \mathcal{T} , whereas attribute/port variables $P \in \mathcal{P}$ may choose their value from \mathcal{C} (if P represents a port connecting to another component), or may take on a numeric or a string value (if P represents an attribute). \square

Each variable in $C \in \mathcal{C}$ may be *active* or *inactive*; it is active if C corresponds to a component that is part of the solution of

a configuration problem, and it is inactive otherwise. Similarly for variables in \mathcal{P} .

A generic constraint is a logical implication of the form $\Phi \Rightarrow \Psi$ over variables in \mathcal{X} , where Φ represents the precondition of the constraint and Ψ specifies the variables and constraints that are introduced into the CSP if Φ holds for some instantiation of \mathcal{X} in \mathcal{C} . Typically, Φ is used to express that constraint Ψ is applicable only for particular component types, while Ψ would enforce the requirements on attribute values and port connections.

A configuration problem confines a GCSP to those CSPs that satisfy the configuration goal, which is expressed by an initial set of components and constraints that must be met.

Definition 2 (Configuration Problem) A Configuration Problem is a CSP $\langle V, \Delta \rangle$ where $V \subseteq \mathcal{C} \cup \mathcal{P}$ represents the initially active variables, and Δ contains a set of constraints over V that express the desired configuration goal. \square

A solution to a configuration problem R is given by an assignment of values to all CSP variables in R such that all constraints in R are satisfied and all possible instances of generic constraints over variables in R are satisfied.

Example 1 Consider a simple example where a flashlight is to be assembled from its parts. Assume that the component types are $\mathcal{T} = \{Bulb, Case, Battery, SmallBattery, Switch\}$, and let $A = \{V, P1, P2, S, B\}$. Here V is an attribute that represents the battery's voltage, while the remaining attribute names refer to ports of component type *Case*. The GCSP representing this domain contains an infinite number of component variables c_i , each of which may be assigned a component type from \mathcal{T} . Similarly, the set of property variables is given by $\mathcal{P} = \{c_1.V, c_1.P1, \dots, c_2.V, \dots\}$, each with domain $\{c_1, \dots\}$.

Generic constraints describe the relationships and restrictions between different component, port, and attribute variables. For example, to assert that each *Case* component must be connected to a *Bulb* component via its *B* port, a generic constraint

$$X = Case \wedge X.B = Y \Rightarrow Y = Bulb$$

is asserted in Γ . Here, X and Y denote meta variables that are instantiated with variables in \mathcal{C} . Note that the domain of $X.B$ is the set of component variables; therefore, the generic constraint establishes the structure of the instance-level CSP dynamically. Once instantiated, the generic constraint will manifest as a CSP-level constraint that asserts that, for some CSP variable c_i , the property variable $c_i.B$ must refer to another component variable c_j whose value is *Bulb*. Assuming that similar generic constraints are defined for the *Bulb* and the remaining component types, further generic constraint instances will be created that restrict the values of c_j 's property variables and neighbors. A complete configuration is then given by a set of component and port variables together with instantiated generic constraints such that all instantiations of generic constraints have been formed and are consistent. \square

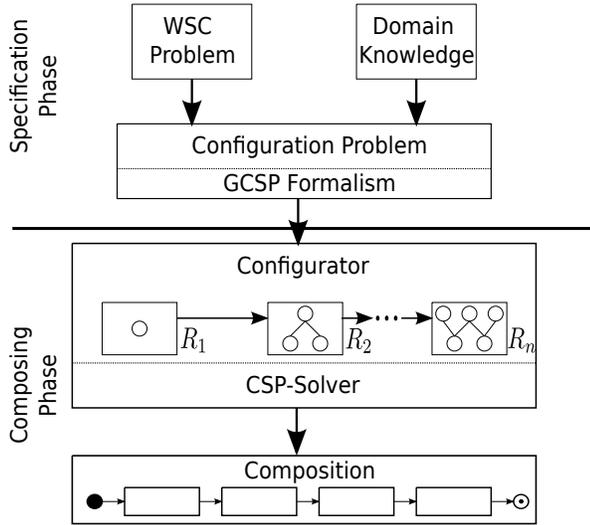


Fig. 1. GCSP Service Composition

III. SERVICE CONFIGURATION AS GCSP

To apply GCSPs for service composition, the service profiles, process constraints, and composition goal must be translated into a configuration problem. The overall architecture of our approach is shown in Figure 1.

A configuration problem is created from a given service composition problem and associated domain knowledge, where the initial variables and constraints are chosen to reflect the composition goal. Inputs that have been made available by the user and particular service instances that should be used in the composition are also created in the CSP. The relevant data types and available service component types, their properties and requirements are translated into generic constraints.

The GCSP representation of the configuration problem is subsequently solved in an incremental process, where the initial constraint problem R_1 is extended to incorporate additional service components until a consistent solution R_n to the CSP has been found that also satisfies all generic constraints. Here, techniques that have been developed to incrementally maintain and solve the CSPs can be leveraged to find suitable assignments efficiently.

The set of active variables and their value assignment in the final solution of the GCSP encode the service components and their interconnections that comprise a satisfying service composition. This solution is extracted and subsequently converted into other representations, such as BPEL.

Example 2 We use the *Producer-Shipper* composition scenario proposed in [10] to present the features of our framework. This scenario poses interesting challenges because the composition is at the process level (involving message correlation between the producer/shipper processes and user interactions) rather than just assembly of atomic activities.

In this problem, a user places an order for a product that is to be produced and shipped. The producer process in this scenario consists of several steps such as receive request, send

quotation to user, receive user approval, manufacture product, and offer product. The shipper process also consists of similar tasks. The composition goal is to combine the two processes with user interaction (modeled as a service). Note that this is not merely a sequential invocation (producer and ship) task, instead the composition involves correlation of messages between the two processes and the user. For example, the quotations from the shipper and producer must be composed together and offered to the user for approval. User approval is decomposed again to carry out tasks in individual processes. For more details see [10]. \square

In the following we discuss essential characteristics of our modeling approach, and illustrate constraints and requirements that do not usually exist in traditional configuration problems but must be adhered-to in the process- and web-based service configuration domain.

A. Modeling individual service components.

Similar to GCSP-based product configuration, where properties of individual parts and possible interconnections are modeled as generic constraints, models of individual service types are also expressed as generic constraints. For each service type, generic constraints express the properties and input and output ports the service needs and offers, respectively. Additional constraints ensure that only ports of other services that are compatible are considered as possible candidates in the synthesis process.

Consider component *ReceiveQuote* in Figure 2. To express that port q only accepts data objects of type *Quote*, the following constraints would be asserted:

$$X = \text{ReceiveQuote} \wedge X.q = DC \wedge DC.value = D \Rightarrow D = \text{Quote}.$$

$$X = \text{ReceiveQuote} \wedge X.q = DC \Rightarrow DC.out = X.$$

The first generic constraint states that the data value held in the *DataConn* component that is connected via the q port must be of type *Quote*. The second constraint asserts that q is indeed connected to the *out* port. This constraint is necessary, since GCSP constraints generally refer to component variables rather than ports, and therefore connections to different ports of *DataConn* cannot be distinguished without resolving the reverse reference, too.

While the service composition problem shares the component-oriented modeling paradigm with traditional applications of constraint-based product configuration, significant differences between the two domains must be overcome. In the following, we briefly outline the central issues and our extensions to the basic GCSP model.

B. Non-uniform component interfaces.

While traditional product configuration typically relies on a carefully crafted library of components and generic constraints to describe hierarchies of similar components, the models of individual components and data types involved in service configuration often stem from heterogeneous sources.

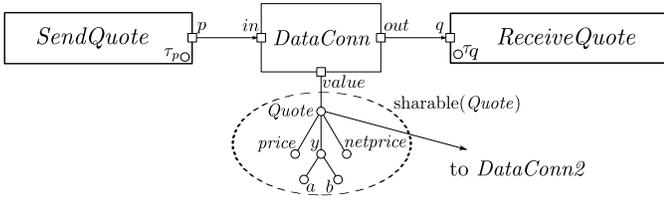


Fig. 2. Data Connection Component

Therefore, components may be heterogeneous in structure and may expose some of the service implementation details that would otherwise be hidden behind generic constraints referring to attributes and ports of abstract component types.

For example, different providers may choose to offer the same product but their service implementations may have different attributes and ports. A modeling approach that expresses connections between components based on the component's type and attribute names, explicit disjunctions are required in generic constraints to compensate for such differences. Since the number of potential service providers may be large, this modeling approach is inefficient in a heterogeneous domain.

We address this problem by introducing artificial abstraction boundaries in the form of abstract data types into the modeling process. Rather than basing generic constraints on component types, service components are linked via the data that they require and provide. For services that communicate predominantly via message passing, the structure and content of messages form natural abstraction boundaries. (Similar to popular Semantic Web frameworks [11], we assume that a suitable library of mediators to translate between different data representations is available.) For this purpose, we extend the plain GCSP formalism with additional CSP variables that represent the concrete data type(s) that are provided or expected by a component port. Figure 2 illustrates part of our model. In addition to the port variable p , component *SendQuote* also has an attribute variable τ_p ; its domain is the set of concrete data types that port p may emit. Similarly, τ_q holds the possible types that *ReceiveQuote* expects as input via q . While superficially similar to I/O type based service composition frameworks [12], our approach offers more expressive power, since constraints can enforce arbitrary relations between CSP variables and may refer to concrete values.

The implementation of service matching by constraint satisfaction also has implications on the quality of the matches. While purely conceptual matching frameworks such as OWL-S often rely on the open world assumption to accommodate for partial knowledge, our framework permits to interpret matches using the closed world assumption, when concrete instance level information is available in the GCSP. As a consequence the effect of undesirable matches can be reduced in many cases [13].

C. Composite data types and invariants.

While component models for product configuration are often centered around components and their attributes, models

of complex data structures that are transformed by a service are even more important in service configuration.

Our framework is powerful enough to represent data types as components, which alleviated the need for separate structure attributes. We introduce explicit component types representing data models into the framework, where attributes and invariants of structured data types are captured as generic constraints. Treating data as components seamlessly extends generative configuration to data objects, such that concrete values passed between components can be synthesized as part of the configuration process.

Figure 2 shows a data component that is connected to the *value* port of a *DataConn* component. The root node *Quote* of the data component is a CSP component variable; attributes of a data type as well as data type specific invariants can be formulated as generic constraints. For example, to assert that attribute *price* of type *Quote* must always be less than *netprice*, the following generic constraint can be stated:

$$X = Quote \Rightarrow X.price = A \wedge X.netprice = B \wedge A < B.$$

D. Control and data flow.

While product configuration is predominantly concerned with assembling components based on whole-part relationships, process configuration must explicitly account for control- and data flow between components.

We extend our model to include explicit control and data flow links, where the directionality of each link is expressed through a dedicated link component. For data connections, this component also carries a value part, represented as a data component in the CSP. Figure 2 shows a data connection component with ports *in*, *out*, and *value*. Directionality is encoded by convention on the usage of component port, with *in* and *out* being reserved for incoming and outgoing connections, respectively. This can be enforced through the generic constraint associated with component type c_1 :

$$X = SendQuote \wedge X.p = DC \Rightarrow DC.in = X$$

Data propagation between components is handled naturally in our models, since the GCSP framework is based on a unidirectional connection model where ports refer to the connecting components. Therefore, multiple ports may share the same CSP variable. Figure 2 shows an example, where a data component is shared by two *DataConn* component instances.

While data sharing is convenient for many data types, additional precautions must be taken to prohibit duplication for certain data objects. While data components representing information usually can be duplicated as needed, objects representing physical artifacts may not. For example, using the same credit card details twice to purchase two pair of shoes is usually admissible, whereas pretending that a single pair of shoes could simply be worn by different persons at the same time is usually not.

Our solution to this problem is to extend the constraint model with an additional predicate *SHARABLE* that distinguishes sharable data types from those that are not. For each non-sharable type T , an additional generic constraint is imposed that ensures that each component variables of type T is references by at most one other CSP variable.

E. Flow constraints.

Since variables in a GCSP are considered independently, additional effort is required to ensure that data flow follows the appropriate control flow paths. In particular, non-local constraints describing process-related constraints must be introduced.

For example, to ensure that a quote for some product indeed stems from the same vendor the request was sent to, constraints restricting the origin of a received data item are necessary. For this purpose, we introduce non-local process constraints over control and data flow paths that prescribe conditions that a valid path must satisfy. In contrast to generic constraints that are local to a component, path conditions impose constraints on components and ports that are reachable via some path. We formalize path constraints as LTL-like formulas to be able to express complex structural and temporal properties of processes.

For example, to assert that a quotation should be prepared prior to being assessed for suitability, the *ReceiveQuote* service may assert the generic constraint

$$X = \text{ReceiveQuote} \Rightarrow G[X.q \rightarrow F[\text{compType}(\text{SendQuote})]]$$

Here, the proposition $X.q$ is true only for the CSP variables that represents this particular port, and $\text{compType}(T)$ represents a proposition is true iff a component along the path is of type T . We interpret the temporal operator F as going in opposite direction of data flow.

The same principle can be utilized to synthesize complex workflow patterns, where interaction with services must proceed according to given protocol specifications. By associating individual service instances with a unique identity, complex flow interaction protocols can be enforced using flow constraints over sequences of service identifier.

For this purpose, we introduce a distinguished attribute ID which holds unique identifiers for each component instance. The identifiers are automatically assigned when a component variable is activated. We use this attribute in LTL constraints to formalize non-local process-related properties.

For example, a protocol that requires that payment information is submitted to the same vendor from which a quote for the product was obtained can be easily expressed as LTL formula relating the origin of the received quote data object and the destination of the data connection representing the submission of payment details.

F. Structural constraints.

Structural constraints ensure that all control and data flow connections are well formed: cycles and multiple incoming or

outgoing connections for a port are prohibited to ensure each data control flow path is deterministic and each data path has a uniquely determined value. For example, to assert that each *in* port of a *DataConnection* component is supplied by at most one input connection, the following generic constraint can be used:

$$X = \text{DataConnection} \wedge X.in = Y \wedge X.in = Z \Rightarrow X = Z$$

G. Model cardinality.

In classical product configuration, the space of possible configurations is typically bounded by the maximum number of components that can be plugged into its containing structure. For services processes, additional measures have to be established to ensure a finite service composition will be found if one exists. Our generic approach to computing service configurations incrementally is described in the next section.

IV. AN ALGORITHM FOR PROCESS COMPOSITION

In this section we develop an incremental algorithm that computes solutions for a GCSP. While the declarative characterization of complete GCSP solutions given in Section II is convenient to illustrate the principles, incremental computation of solutions must account for partial configurations that may arise during the solving process.

It is easy to see that in a complete assignment to variables in a GCSP, each generic constraint evaluates to either true or false. For partial assignments, however, the possibility arises that a constraint cannot be evaluated completely if a component required by the constraint has not yet been introduced into the GCSP. Hence, we evaluate constraints over three-valued logic, where constraints that cannot be evaluated completely resolve to the “unknown” value and are deferred until the required components have been introduced in the configuration.

While the declarative characterization of a GCSP solution does not impose a particular order on the evaluation of constraints, evaluation ordering may have significant impact on performance in practice. Since structural process integrity is enforced through constraints, we must ensure that these constraints are applied before other non-local constraints are considered. Otherwise, invalid compositions may arise or the evaluation of constraints may not terminate. We impose a partial ordering on constraints to ensure that structural integrity constraints are applied first.

Once partial configurations are extended incrementally, it is possible that certain configurations may arise that can be extended infinitely often, but cannot be completed to a valid configuration due to an incorrect choice earlier in the search. To ensure our search for compositions terminates, we employ an iterative deepening strategy that limits the number of component instances that may be introduced in the configuration. 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.

Algorithm 1 computes a solution of a configuration problem $\langle V, \Delta \rangle$ w.r.t. a GCSP $\langle \mathcal{X}, \Gamma, \mathcal{C}, \mathcal{P} \rangle$. For simplicity, the parameters of the GCSP and the global parameter θ that limits the size of the composition are kept implicit in the algorithm. The pair $\langle V, \Delta \rangle$ initially holds the active variables and constraints present in the initial configuration problem, and will subsequently be expanded as the CSP is extended by application of generic constraints. The overall algorithm commences with checking the consistency of the given CSP (line 1). If an inconsistency is detected, the current CSP cannot be extended to a complete solution and backtracking occurs. Otherwise, an unassigned variable is selected and a value is assigned by calling the ASSIGN function. As a result, a revised CSP reflecting the updated composition is obtained and subsequently solved. The ASSIGN function aims to select a suitable value for the selected variable v , based on the remaining domain values for v in Δ . If the domain has become empty, backtracking occurs. Otherwise, the constraint store is extended to reflect the new assignment, and generic constraints are checked to assess if previously deferred constraints have become applicable (line 20). If an instance $(\phi \Rightarrow \psi)[\sigma]$ of a generic constraint is applicable, it may activate additional variables $\text{VAR}(\psi[\sigma])$ that are added to the CSP. We use σ to denote the substitution of meta variables with CSP variables such that $\phi[\sigma]$ is satisfied in the constraint store Δ . The test in line 22 ensures that the number of component instances present in the CSP does not exceed the limit imposed by our iterative strategy. If the limit is reached, the algorithm backtracks and attempts to find a solution with fewer components. If that fails, SOLVE yields \perp and θ is incremented before retrying (not shown).

Algorithm 1 can be seen as an incremental solving process, where generic constraints drive the expansion of the constraint system. Here, ISCONSISTENT encapsulates a classic CSP solver that may employ well-known CSP heuristics to efficiently filter variable domains and check for inconsistencies. As described earlier, we require that the order in which constraints are checked can be controlled, for example by associating priorities with constraints (a mechanism that is supported by many CSP implementations). Similarly, in SELECT-UNASSIGNED and NEXT-VALUE established domain-independent CSP variable and value selection heuristics may be applied, complemented by domain-specific heuristics (if available).

The meta-level expansion is driven by NEXT-VALUE in conjunction with EXPAND-GCSP and is independent of particular CSP solving and checking implementations. To ensure the GCSP solution remains small, NEXT-VALUE employs particular value ordering, where port variables are first left unconnected, followed by connections to existing components; only if no earlier assignment has led to a consistent partial assignment will CSP variables corresponding to newly created components be considered. If a new component variable is activated in the CSP, the relevant generic constraints are subsequently instantiated in EXPAND-GCSP.

While the computational complexity of the algorithm is

Algorithm 1 GCSP Solving Algorithm

```

function SOLVE( $V, \Delta$ )
1  if  $\neg$ ISCONSISTENT( $V, \Delta$ ) then
2    return  $\perp$  ▷ Backtrack
3  end if
4   $v \leftarrow$  SELECT-UNASSIGNED( $V, \Delta$ )
5  if no such  $v$  exists then
6    return  $\Delta$  ▷ Solution found
7  end if
8  repeat
9     $\langle V', \Delta' \rangle \leftarrow$  ASSIGN( $V, \Delta, v$ )
10    $\Delta'' \leftarrow$  SOLVE( $V', \Delta'$ )
11  until  $\Delta' = \perp \vee \Delta'' \neq \perp$ 
12  return  $\Delta''$ 
end function
function ASSIGN( $V, \Delta, v$ )
13  repeat
14    $u \leftarrow$  NEXT-VALUE( $v, \Delta$ )
15   if no such  $u$  exists then
16     return  $\langle \perp, \perp \rangle$  ▷ Backtrack
17   end if
18    $\Delta'' \leftarrow \Delta \cup \{v = u\}$ 
19    $\langle V', \Delta' \rangle \leftarrow$  EXTEND-GCSP( $V, \Delta''$ )
20  until ISCONSISTENT( $V', \Delta'$ )
21  return  $\langle V', \Delta' \rangle$ 
end function
function EXTEND-GCSP( $V, \Delta$ )
22  for each  $(\phi \Rightarrow \psi) \in \Gamma$  do
23   if  $\exists \sigma \Delta \models \phi[\sigma] \wedge \Delta \not\models \psi[\sigma]$  then
24      $V \leftarrow V \cup \text{VAR}(\psi[\sigma])$ 
25     if COMPS( $V$ )  $> \theta$  then
26       return  $\langle \perp, \perp \rangle$ 
27     end if
28      $\Delta \leftarrow \Delta \wedge \{\psi[\sigma]\}$ 
29   end if
30  end for
31  return  $\langle V, \Delta \rangle$ 
end function

```

PSPACE complete in the worst case, it is usually efficient in practice (as will be shown in the following section). This can be explained by the GCSP expansion strategy that helps to keep the CSP size small by introducing variables as needed. Moreover, since the constraint network typically grows from areas where variables are heavily constrained outwards, many of the newly activated variables are immediately bound to a specific value. Therefore, they can be treated as constants and be removed from the remaining CSP, again leading to a reduction in problem size.

V. EVALUATION

To assess the performance of our GCSP framework, experiments on the well-known *Producer-Shipper* composition problem [10] have been conducted. 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 Example 2) 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.

We conducted experiments for problem variations of 2–28 concurrent instances of the shipping process. For each problem, the composition goal of having a successful shipment at the end was posed, and the sufficient number of producer and shipper service tasks was created in the initial configuration. (Since the solution to each problem was known beforehand, the numbers could be chosen appropriately. Preliminary experiments with slightly more instances did not show significant differences to the results reported here, suggesting that the GCSP mechanism is effective in filtering out irrelevant components.)

Each configuration problem was repeated ten times and averages for the number of components in the final configuration and the CPU time required to arrive at the final configuration were recorded. The results are shown in Figure 3. It can be seen that the problems consisting of four or fewer process instances can be solved almost instantaneously. For larger problems, the time required to solve a problem increases exponentially, but remains within reasonable bounds even for large problems. Our largest problem with 28 producer-shipper processes can be solved roughly three minutes – a result that is very competitive in contrast with other approaches.

Our empirical results confirm that the efficiency of the GCSP formalism stems from a combination of early variable binding and slow growth of the search frontier. Typically, the number of inputs for a service is low, and most service processes do not contain long chains of service invocations where *every* service requires many *unique* inputs simultaneously. Therefore, as services are added to the composition, the number of unassigned variables grows almost linearly with the number of components rather than exponentially.

Our iterative strategy to increase the bounds on the number of components in a solution helps to control possible inefficiencies due to backtracking, since a wrong choice of a component early on often results in considerable process

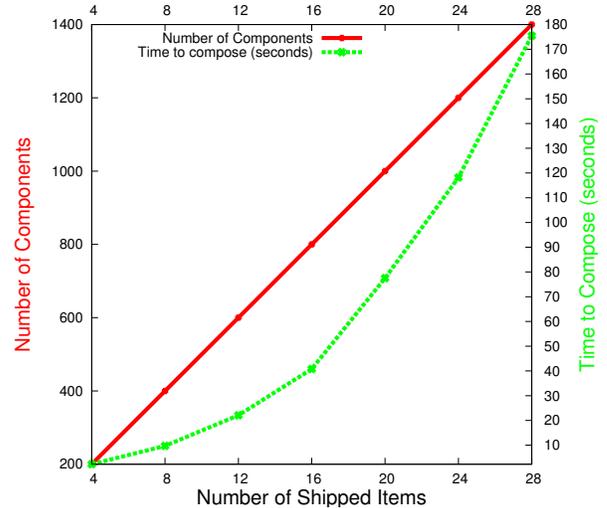


Fig. 3. Results: P-S Composition

duplication that grows quickly towards the threshold, resulting in quick pruning of the branch.

VI. RELATED WORK

CSPs have already been applied for service composition. In [7], a given abstract workflow is modeled as a CSP in order to find suitable concrete services to instantiate an executable process. Likewise, in [14] multiple CSPs were used to solve constraints at different levels of abstractions. In [15], hierarchical decomposed solving of CSPs were proposed in the context of web service composition. Constraint optimization techniques have been proposed to find the a set of services optimizing a given cost function [6]. All these works require the number of service components to be fixed, whereas our GCSP formalism introduces additional services as required. The fixed scenario can be handled by imposing constraints on the number of services. Cost-based optimization can be integrated in our GCSP framework by exploiting preference constraints and constraint optimization techniques; however, this is beyond the scope of this article.

Configuration techniques based on meta-models have been developed to synthesize a composite service interaction process from a given set of (abstract) workflow instances [4]. While the set of service components is fixed in each composition scenario, the authors propose iterative abstraction refinement when more dynamic composition is required. In contrast, we do not require a library of workflow instances, but synthesize compositions in a goal-directed manner. We are currently extending our framework to exploit available workflow libraries.

Compared to planning-based composition approaches [16], [3], GCSPs allow to exploit established CSP solving heuristics to detect infeasible sub-compositions early. Furthermore, constraints naturally permit to interleave plan refinement, service execution and composition, where services are queried to obtain concrete information to complement the abstract service

specifications and better guide the solver. Our framework is also more flexible in that a given (partial) composition can be revised by removing or changing components to meet revised requirements. This is essential for robust software systems, where faults and intermittent failures must be compensated for.

Our constraint-based formalism complements scenarios where non-functional properties of service processes serve as an additional filter to enforce further restrictions on service selection and composition [17], [18].

VII. CONCLUSION

We have shown that the service composition problem can be translated into a dynamic constraint satisfaction problem, where the scope and structure are determined automatically based on a meta-level constraint framework. We have introduced the basic principles of component-oriented generative constraint problems, and have shown how these ideas can be extended to intrinsic properties of complex service process composition scenarios. By mapping service composition problems to generative constraint satisfaction problems, domain-independent solving heuristics can be applied. Our empirical studies confirm that our technique is efficient in practice and scales to non-trivial problems while retaining rich modeling capabilities. The expressive power of the framework has been shown to cover the Semantic Web Description Logic standards [19] and is compatible with OWL-S descriptions.

We are currently extending our GCSP framework with mechanisms for hierarchical configuration of work flows, where complex interaction protocols spanning multiple service components and sub-processes must be followed to achieve a desired output. In the future, we plan to investigate the integration of GCSP composition with the execution of concrete services. This combined approach promises to improve composition results significantly, since precise information obtained by querying a service can complement service matching based on conceptual models. Both paradigms integrate naturally in our constraint framework.

REFERENCES

- [1] H. R. M. Nezhad, B. Benatallah, F. Casati, and F. Toumani, "Web Services Interoperability Specifications," *IEEE Computer*, vol. 39, no. 5, pp. 24–32, 2006.
- [2] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing Semantics to Web Services: The OWL-S Approach," in *Proc. of SWSWPC*, ser. LNCS 3387, 2004, pp. 26–42.
- [3] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau, "HTN planning for Web Service composition using SHOP2," *Journal of Web Semantics*, vol. 1, no. 4, pp. 377–396, 2004.
- [4] P. Albert, L. Henocque, and M. Kleiner, "Configuration Based Workflow Composition," in *Proc. of the IEEE International Conference on Web Services (ICWS 2005)*, Orlando, Florida, USA, Jul. 2005.
- [5] S. Kona, A. Bansal, M. B. Blake, and G. Gupta, "Generalized semantics-based service composition," in *Proc. of the IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, 2008.
- [6] A. B. Hassine, S. Matsubara, and T. Ishida, "A Constraint-Based Approach to Horizontal Web Service Composition," in *Proceedings of the International Semantic Web Conference (ISWC)*, Athens, GA, USA, Nov. 2006.
- [7] R. Thiagarajan and M. Stumptner, "Service Composition With Consistency-based Matchmaking: A CSP-based Approach," in *Proceedings of the European Conference on Web Services (ECOWS)*, Halle, Germany, Nov. 2007.
- [8] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, "Configuring large-scale systems with generative constraint satisfaction," *IEEE Intelligent Systems*, vol. 13, no. 4, 1998.
- [9] M. Stumptner, G. Friedrich, and A. Haselböck, "Generative constraint-based configuration of large technical systems," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 12, no. 4, pp. 307–320, Dec. 1998.
- [10] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi, "Automated synthesis of executable web service compositions from BPEL4WS processes," in *WWW*, 2005, pp. 1186–1187.
- [11] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, "Web Service Modeling Ontology," *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.
- [12] R. Grønmo and M. C. Jaeger, "Model-Driven Semantic Web Service Composition," in *APSEC05*, Dec. 2005, pp. 79–86.
- [13] R. Thiagarajan, M. Stumptner, and W. Mayer, "Semantic Web Service Composition by Consistency-based Model Refinement," in *Proc. of IEEE Asia-Pacific Service Computing Conference (APSCC)*, Tsukuba Science City, Japan, Dec. 2007.
- [14] Q. A. Liang, S. Miller, and J.-Y. Chung, "Service mining for web service composition," in *Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration, IRI-2005*, D. Zhang, T. M. Khoshgoftaar, and M.-L. Shyu, Eds. IEEE Systems, Man, and Cybernetics Society, 2005, pp. 470–475.
- [15] D. Maruyama, I. Paik, and M. Shinozawa, "A flexible and dynamic csp solver for web service composition in the semantic web environment," in *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*. Washington, DC, USA: IEEE Computer Society, 2006, p. 43.
- [16] M. Carman, L. Serafini, and P. Traverso, "Web Service Composition as Planning," in *ICAPS Workshop on Planning for Web Services*, Trento, Italy, Jun. 2003.
- [17] M. Lin, J. Xie, H. Guo, and H. Wang, "Solving qos-driven web service dynamic composition as fuzzy constraint satisfaction," in *The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, 2005. EEE '05.*, Los Alamitos, CA, USA, Mar. 2005, pp. 9–14.
- [18] S. Padmanabhuni, B. Majumdar, M. Chawla, and U. Mysore, "A constraint satisfaction approach to non-functional requirements in adaptive web services," in *NWESP '06: Proceedings of the International Conference on Next Generation Web Services Practices*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 109–116.
- [19] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker, "Configuration knowledge representation for semantic web applications," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 17, no. 1, pp. 31–50, Jan. 2003, special issue on Configuration.