Semi-Automated Software Composition Through Generated Components

Felix Mohr  
Department of Computer Science  
University of Paderborn, Germany  
felix.mohr@upb.de

Hans Kleine Büning  
Department of Computer Science  
University of Paderborn, Germany  
kbcs1@upb.de

ABSTRACT

Software composition has been studied as a subject of state based planning for decades. Existing composition approaches that are efficient enough to be used in practice are limited to sequential arrangements of software components. This restriction dramatically reduces the number of composition problems that can be solved. However, there are many composition problems that could be solved by existing approaches if they had a possibility to combine components in very simple non-sequential ways.

To this end, we present an approach that arranges not only basic components but also composite components. Composite components enhance the structure of the composition by conditional control flows. Through algorithms that are written by experts, composite components are automatically generated before the composition process starts. Therefore, our approach is not a substitute for existing composition algorithms but complements them with a preprocessing step. We verified the validity of our approach through implementation of the presented algorithms.

Categories and Subject Descriptors

D.1.2 [Software Engineering]: Automatic Programming—Search-based software engineering; D.2.13 [Software Engineering]: Reusable Software—Reuse models

General Terms

Theory, Design

Keywords

software composition, generated components, reuse patterns

1. INTRODUCTION

This paper examines a possibility of augmenting the degree of automation in software composition. Software composition is the process of assembling a new software artifact using existing components. The basis for this process are a repository of components and a set of rules.

Definition 1. Let \( \mathcal{L} \) be a first-order logic language with finitely many predicates, variables, and constant symbols, but no function symbols.

A component is a tuple \((I, O, P, E)\), where \(I\) and \(O\) are sets of variable names, the preconditions \(P\) are a set of sets of literals of \(\mathcal{L}\) with variables of \(I\), and the effects \(E\) are a set of literals of \(\mathcal{L}\) that contain only variables of \(I\) and \(O\).

Let \(C\) be a set of components and \(K\) be a set of rules under \(\mathcal{L}\). Then \((C, K)\) is a software component system.

This kind of software components is often referred to as services. Hence, We use the terms synonymously.

A (software) composition problem is defined by a component system and a query, which is given in form of a component specification, hence a tuple \((I_q, O_q, P_q, E_q)\).

Despite first achievements, even rather easy composition problems cannot be solved by existing approaches. Current composition problems are mostly solved by plugging components together in a sequential way [14, 7, 15, 11, 8]. The behavior of the composition at runtime does not depend on the outputs of the invoked components. Two examples illustrate some of the shortcomings of that idea.

Example 1. A company wants to adjust the credit rating of some of its customers according to the information of a remote rating service. For customers that are not reliable according to that service, a warning flag shall be added to the corresponding entry in the database. The software components available in this scenario are getRemoteRating and insertWarning. getRemoteRating provides a boolean value that is true if the customer is reliable and false otherwise. insertWarning updates the dataset of a customer by inserting a flag that indicates possible trouble with payment.

Example 2. A user wants a list of nearby book stores that have a desired book on stock. She provides the author and the title of the book together with a maximum distance in which potential book stores should be located. The available components are the following: getCityForPosition, getStoresInCity, getISBN, hasISBN, getPositionOfStore, getDistance. The service getCityForPosition determines the name of the city for a given coordinate. getStoresInCity is a catalog that provides a list of book stores in a city. getISBN determines the ISBN of a book described by author and title. hasISBN provides the availability of a medium for a given store and ISBN. Finally, getPositionOfStore determines the coordinate of the provided store and getDistance calculates the euclidean distance between two coordinates.

Both tasks are unsolvable for most current composition approaches. First, maintenance of object lists is required,
whereas most of the available components deal with single objects. For example, getRemoteRating is a component that provides information for a single customer, while the input of the query is a list of customers. The obvious solution is to invoke getRemoteRating in a loop that iterates over the input list. Figure 1 shows the solution for Example 1. Second, the control flow of a solution necessarily depends on results obtained at runtime. For example, the amount of calls of the component hasISBN depends on the list size of book stores that are in range of the user. Since current composition systems do only consider loop-free control flows, they cannot solve problems with these characteristics.

Our idea to achieve compositions that solve problems like these is to combine not only component calls, but entire code blocks. Code blocks enrich the components by conditional statements and loops that allow to react on the outputs of component calls.

The composition algorithm can treat such code blocks as (composite) components with descriptions as given in Def. 1. Hence, the composition process itself does not change. Composite components are obtained semi-automatically in a preprocessing step through generators written by human experts. Due to complexity, we see no possibility to automatically create useful new components without human expertise. Therefore, experts define templates and generators that instantiate the templates with existing components. The resulting instantiations of the templates are called composite components. Flexibility is gained, because each template reflects a particular reuse idea that is domain independent. By the instantiation with domain specific components, the obtained composite components are domain specific and can be applied in the composition process.

Summarizing, this paper contributes through a novel approach to solve open composition tasks. We explain how new components can be obtained through a generator-based instantiation of templates. Our examples point out how current solutions are improved by the approach.

The rest of this paper is structured as follows: Section 2 explains how the degree of automation in software composition is augmented through generated composite components. Section 3 shows the generators that we used to solve the example problems and, by this, gives a proof of concept for the approach. Related work is discussed in Section 4.

2. ENHANCING COMPOSITION

2.1 Preliminaries

To achieve better compositions, we can choose between two main strategies. First, we may search not only for sequences of components but for more general structures, e.g., DAGs. Second, we may sequentially arrange code blocks instead of components. The code blocks would convey the required code complexity with loops and conditional statements, but compositions would still be sequences.

Due to practical reasons, we adopt the second strategy that arranges code blocks in a sequential fashion. The advantage is that sequentially arranging code blocks can be done straightforward with standard planning algorithms. In contrast, planning more sophisticated structures requires either a very advanced design of the search space or a proprietary planning algorithm.

2.2 Composing Instantiated Templates

```
subset = new List();
foreach (item : list) {
  (value) = getRemoteRating(item);
  if (value == 'false') {
    subset.add(item);
  }
}
foreach (item : subset) {
  insertWarning(item);
}
```

Figure 1: A sequence of two code blocks

The goal is to obtain new components with as few human activity as possible.

To this end, we derive code blocks from templates. Templates are domain independent code blocks with placeholders for component calls and boolean expressions.

The process of replacing the placeholders of a template with component calls and assertions is its instantiation. The instantiation of a template determines the input and output variables for the resulting code block as well as its precondition and effect. Section 2.3 explains how the instantiation of a template is done with generators.

The code block that results from the instantiation of a template is nothing else than a composite component. The result of a template instantiation is a set of code fragments with inputs, outputs, preconditions, and effects, which fits into Definition 1 of a component together with an implementation. To terminologically distinguish them from the components of the component system, we call the instantiated templates composite components.

Since templates are not domain specific, they may be instantiated to semantically very heterogeneous composite components. The structure of templates is fixed, yet the concrete semantics of a composite component depends on the instantiation of the template. Templates have generic preconditions and postconditions that are concretized during instantiation. Consequently, the composite components obtained by the instantiations of a template can be extremely different. For example, the subset calculating template can be used to compute only unreliable clients but also unpaid invoices or bookstores that have a particular book on stock. This is an essential difference to other template-based approaches, where templates are domain specific.

A composition is then a sequence of composite component calls. A component call binds the inputs and outputs of a component to concrete (program) variables in order to model the data flow of the composition. The formal structure of compositions of composite components is not different from the compositions of basic components. A solution for Example 1 is depicted in Figure 2.

We have two temporal possibilities to obtain composite components for the composition process. On one hand, templates could be instantiated at the time of composition (on-the-fly instantiation). The composer would then in each step choose a template and the component calls with which to instantiate it. On the other hand, templates could be instantiated before the composition process starts (pre-instantiation).

Both possibilities have a major advantage over the other. The advantage of on-the-fly instantiation is that we only instantiate templates that appear promising for the particular task. This entails the chance to keep the number of possible
instantiations for each template significantly smaller than in the case of pre-instantiation. However, we are not aware of any composition approach that is capable of this kind of run-time instantiation of templates. On the contrary, the clear advantage of pre-instantiation is that the composition process itself does not need to be changed. If the instantiation is done in a preprocessing step, the existing composition algorithm can treat the composite components as if they were basic components.

We focus on pre-instantiation. The task then comes back to sequentially arranging (composite) components.

We choose this approach, because at this point of research we concentrate on the proof of concept rather than on optimization issues. In the long term view it would be probably better to apply on-the-fly instantiation with heuristic support. However, without a sophisticated filtering technique, the benefits of on-the-fly instantiation are not exploited, and the accumulated calculation of instances would be in most cases even costlier than in the case of pre-instantiation.

Also, there are good arguments that relativize the problem of the great number of instantiations. First, pre-instantiation is a preprocessing step and not necessary for every query that enters the system. Second, the combinatorial explosion is less fatal than it appears at first sight. We may reasonably require that the number of placeholders \( p \) in the templates, the inputs \( i \), and the outputs \( o \) of the basic components can be bound to a small constant, say \( p, i, o \leq 5 \). This bounds the possible data flows in the composite component to a (possibly large) constant. For \( n \) basic components, the remaining complexity lies in trying \( O(n^p) \) instantiations (with \( p \leq 5 \)). This bounds the possible data flows in the composite component to a (possibly large) constant. For \( n \) basic components, the remaining complexity lies in trying \( O(n^p) \) instantiations (with \( p \leq 5 \)). This bounds the possible data flows in the composite component to a (possibly large) constant. For \( n \) basic components, the remaining complexity lies in trying \( O(n^p) \) instantiations (with \( p \leq 5 \)).

2.3 Template Instantiation with Generators

The templates that are used to derive composite components are written by experts. Contemplating application code, we conjecture that the reuse of solutions of a particular class of tasks can often be captured in patterns. For example, the ways in which methods in object oriented programming are reused can be captured by abstract patterns. Of course, some kinds of methods are very specific and not suitable for much reuse such that the search for a pattern does not make much sense. However, business program logic is first of all concerned with reading, creating, and changing entities. Intuitively, there are very common ways how those entities are processed in a program, independently from the concrete semantics of the objects. This insight led to frameworks that encourage or even force the developer to think on the level of creating, reading, updating and deleting business entities [12]. Identifying such patterns of reuse and capturing them in templates is the task of the expert.

Besides the pattern itself, the expert must provide semantic information with a template in order to enable a sound instantiation. First, the expert provides inputs, outputs, preconditions, and effects of the template on an abstract level. The concrete inputs, outputs, preconditions, and effects of the resulting composite components are derived from these specifications during the instantiation step. Moreover, the expert must have a proof of the correctness of the template; that is, that the effects can be followed from the preconditions and the program logic. Since the program logic is not finally fixed, the proof can only take place on an abstract level. The expert may define consistency rules that he assumes to be true in order to support his proof. The correctness of these consistency rules must then be checked during the instantiation step.

The pre-instantiation of a template may happen in two ways. The first possibility assumes a centralized instantiation mechanism on the side of the composition system. This requires a description language for templates in which the expert can express templates and rules that are relevant for their instantiation. The advantage is that there must be only one interpreter for such a template description language in order to obtain the set of composite components. However, such a language could become quite complex and by this both difficult to develop and to learn. The second possibility is to demand such an instantiation mechanism already from the expert. An expert would then provide an algorithm that produces a set of composite components on the basis of a component system as its input.

At the current stage of research, it makes sense to outsource not only the pattern creation but also the instantiation to the expert. The development of a pattern language is unreasonable until the presented technique has matured.

Consequently, we assume that composite components are obtained through generators. Experts provide a set of generators that take a software component system as input and produce composite components.

Definition 2. Let \( S \) be the space of software component systems under \( \mathcal{L} \) and \( C \) be the space of all composite components. We denote \( C_s \subseteq C \) as the set of all composite components that only invoke components from the component system \( s \). A generator is a function \( q: S \to 2^C \) such that \( q(s) \subseteq C_s \).

The set of composite components that are available for a component system \( s \) is the union of the outputs of the generators that have been instantiated with \( s \).

Although one may argue that imposing this work on the expert weakens the approach, we consider it as a significant improvement. Of course, every work shifted to the human weakens an approach of automation to a certain extent. However, the benefits that can be achieved by letting an expert develop a possibly general pattern of reuse seem to clearly overcompensate the costs of developing the patterns. The patterns written by the expert can be instantiated for domains and for solutions that the expert does not even
know. The better the abstraction encoded in the pattern, the more components can be derived from it. Our exemplary results strengthen this conjecture.

3. IMPLEMENTATION

We first give an overview over the generators that we created for our investigations. Then, we analyze the generators with respect to their potency to produce new components.

3.1 Our Generators

3.1.1 Fundamental Generators

The basic generators we implemented are ComponentIdentity (CI) and RuleIdentity (RI). CI creates one composite component for each basic component. This generator only copies the basic components; that is, the resulting composite components consist only of the component call. RI inserts an empty composite component for each rule in the component system. Since standard planners do not use rules in the planning process, there is the need to convey the rule knowledge in planning operations to make it be considered by the planner. We achieve this by encoding the rules as “empty” components. The preconditions and effects of the composite components are the premises and the conclusions of the rules respectively. In this way, the rule knowledge is integrated into the planning process.

3.1.2 LoopInvoker

LoopInvoker (LI) generates composite components in which a basic component is invoked for each item of a list A. The produced components guarantee particular properties on the items of A. For example, the composite component on the right of Figure 2 guarantees that all clients in the list are marked as unreliable in the database.

The precondition and effect of a composite component c∗ produced by LI depend only on the precondition and effect of the used basic component. For example, suppose that the precondition of insertWarning in Figure 1 is ¬R(a), where R(a) means that a is reliable, and its effect is W(a), where W(a) means that a warning has been inserted for a. The precondition of c∗ is then that ¬R(a) is true for all a ∈ A, and the effect is that W(a) holds for all a ∈ A.

The IOPE description of a resulting composite component c∗ using a basic component c is straightforward. The inputs of c∗ are a set A and the inputs of c others than the one where the items of A are passed to. The precondition end effect of c∗ are obtained by copying the ones of c and universally quantifying the particular input to the items of A. Composite components generated by LI have no output.

3.1.3 SubsetCalculator

SubsetCalculator (SC) generates composite components that compute a subset A′ of a given set A. For example, the component on the left of Figure 2 computes \( \text{subset} = \{ a \mid a \in \text{list} \land \neg R(a) \} \) (the unreliable clients). The property \( \neg R(a) \) is the conclusion of a rule of the component system. The premise of that rule is enabled by the postcondition of the call of getRemoteRating and a concrete evaluation of the result of that call in an if-statement. That is, a rule in the component system lets us infer that \( \neg R(a) \) holds if some x contains the reliability about a and if x is false.

The IOPE description of the resulting composite components can be computed straight forward. The inputs and preconditions are determined in the same ways as for LI. The only output of components generated by SC is the subset A′. Each of the generated components relies on a particular rule whose conclusion gives information about the item currently considered in the iteration. Suppose that this rule has the conclusion \( P(a) \) for the list item a; then the effect of the component is \( A' = \{ a \mid a \in A \land P(a) \} \).

3.2 Evaluation Of The Example

To obtain the following results, we defined the necessary rules to generate the presented composite components and proved for each generator that the descriptions of the produced components are correct. We omit the proof due to space limitations. The solutions were found with a standard PDDL planner.

Even in this small example, the generators significantly increase the number of available components. Starting with the 8 services from the introductory examples and 10 rules, LI produced 1 and SC produced 30 composite components. In other words, only two generators allowed us to automatically increase the size of the component repository almost by the factor of four.

3.2.1 The Rating Example

The solution to Example 1 is depicted in Figure 1. The first component is a result of SubsetCalculator; getRemoteRating is invoked with each client in the input list and the result is compared to false. The result is a list of which we know that it contains only unreliable clients. The second composite component is generated by LoopInvoker. For each client in the set of unreliable clients, a warning is inserted into the system through the component insertWarning.

3.2.2 The Bookstore Example

Example 2 is solved with five composite components; the solution is captured in Figure 3. The basic components getCityForPosition, getBookStoresInCity, and getIsbn are called to determine the bookstores in the city and the ISBN of the desired medium. These components, which are used in lines 1-3 are obtained through the CI generator. Then, the subset of bookstores that have the requested book on stock is calculated in lines 4-9 by the application of a code fragment produced by the SC generator. Next, the distance of the remaining shops to the current position is calculated in lines 10-16 with a composite component generated with SC. The component first determines the position of a shop, then uses this result to calculate the distance to the user’s position, and finally compares it to the maximum distance. Finally, the planner applies two rule components, which were created by the RI generator, in lines 17-18 to prove that the obtained set v5 satisfies the query.

4. RELATED WORK

Service composition has been subject to research for over a decade now, and excellent surveys are found in [2, 10, 1]. However, the great majority of approaches can only find sequential arrangements of component calls. For example, OWLS-XPLAN [7] and the application of SHOP2 in [14] are unable to solve the presented examples, because they cannot deal with sets of objects. The same holds for more recent approaches as to mention [15, 4, 3, 11, 8]. Still, our approach could be used as a preprocessing for these works in order to satisfy the exemplary problems.
The significant improvement is that the templates, which are used by the generators, are not bound to a specific domain. Hence, the work of the expert can be exploited in a much more flexible way.

The opportunities to extend the approach are manifold. Clearly, we could identify more patterns and extend the basis of composite components. Moreover, the model could be extended to support more complex component descriptions. For example, we could imagine stateful components or alternative postconditions. However, the next step should be to evaluate the approach with a solid repository of components to get more detailed results for the approach.

6. ACKNOWLEDGMENTS
This work was supported by the German Research Foundation (DFG) within the project On-The-Fly Computing (SFB 901).

7. REFERENCES