Chapter 2
Automated Software Composition—A Top View

This chapter gives a brief introduction to automated software composition. Section 2.1 provides an overview of the general task of automated software composition. Then, Sect. 2.2 gives an overview over the features of composition problems. Third, Sect. 2.3 proposes *the first level* of a classification scheme, which is the basis for the technical discussions in the next chapters; i.e., it presents the two main classes of composition problems. The discussion of subclasses of the two main classes is part of Chaps. 3 and 4. A summary of the complete classification tree is depicted in Fig. 5.1 in Chap. 5.

2.1 Background

A nice vision statement of automated software composition was given by Koza end Rice in the context of automated programming [76]:

> The goal in automatic programming is to get a computer to perform a task by telling it what needs to be done, rather than by explicitly programming it.

While avoiding explicit programming often is desireable, automated software composition does not aim at replacing classical software development. As pointed out by Hoare, one of the most important properties of software is that "it carries out its intended function" [58]. To express this intended function, we will always need to rely on some kind of formal descriptions, and there is not the illusion of the cocktail party explanation of the vision of automated programming [131], where complex software can be derived from natural language requirement definitions expressed by somebody not even familiar with software engineering. On the other hand, the work of software developers can be *supported* by automation techniques.

The idea of automated software composition is to automate a small part of the code construction process. That is, we do not want to create huge software specifications, press the button, and wait for the signal that the desired software has been deployed on the machine. Instead, we want the machine to create rather simple programs *fast*.

F. Mohr, *Automated Software and Service Composition*,
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-34168-2_2

There seem to be two main cases where automated software composition is preferable over ordinary programming.

First, there may be occasions where we need to create software within the time frame of seconds where any human interaction would simply not be fast enough. For example, we want to run a script that solves an optimization algorithm based on simplex if the instance is rather small and with interior point if it is large, applying specific parameters to the respective algorithm depending on the input instance, which cannot be efficiently hard-coded in the script. We then would create a rough workflow of the general process and automatically refine it at runtime based on the concrete input.

Second, developers often only want to state conditions that should be true for an object instead of describing *how* this is achieved. For example, one would like to be able to write

```
y s.t. PriceOf(y,x) & EUR(y)
```

to say that y should be set to the price of x with respect to its current value in EUR instead of writing the following:

```
p := getBookPriceOf(x);
y = USD2EUR(p);
```

The declarative variant has many advantages. Not only is it closer to the actual intention of the developer, simpler (no temporary variable) and exhibits higher readability. It also decouples description from implementation, which means that the developer does not need to know the exact function that realizes the functionality; the name or location of the function may change without doing harm to the code. In particular, the developer does not even need to know how the property can be computed and whether or not the result is already in the right currency or whether it must be converted. Moreover, given the correctness of the functions used by the synthesizer, the generated code is correct by construction.

The largest subfield of automated software composition is called automated *service* composition. Services are self-contained and platform independent software components. Self-contained means that services do not visibly rely on other components or services, so they can be used right away without the necessity to specify components that should be used for some required interfaces. Platform independent means that two services can be used together independently from the language in which they have been implemented. The idea is that, instead of, say methods in an object oriented programming language, service operations do not communicate by exchanging object references over a commonly accessible memory but by *messages*. These properties allow for the definition of a simple composition model that assumes a set of operations that can be *combined* (platform independentness) ad hoc (self-containedness) into a new software artifact. These properties are naturally given within every programming language, so, by making these assumptions, auto-

mated service composition simply accounts for the fact that we now need to combine components implemented in possibly different programming languages.

2.2 Features of Software Composition Problems

Clearly, one cannot speak of *the* automated software composition problem. I identified 22 very heterogenous features that separate the service composition problems from each other. This greate variance makes many approaches distinct and sometimes even completely uncomparable to others. Since most of these features should be intuitive, I only give a brief overview rather than discussing them in detail. Table 2.1 shows an overview about the features.

Every feature specifies a characteristic that relates to the *algorithm inputs*, *algorithm outputs*, or its *behavior*. Composition problems impose conditions on the inputs, outputs, and even the implementation of composition algorithms (that address the respective problem). Hence, it is natural to see the set of different features that determine a composition problem as constraints made on one of these three aspects.

2.2.1 Input Features

The 14 input features describe characteristics about what is fed to a composition algorithm. Input features are as follows:

- *Presence of a control flow of the solution*.
  In this case, the desired piece of software is specified in form of a workflow, which needs to be concretized. The composition algorithm does not create a control flow but only refines the given one. Prominent representants where the control flow is given are [20, 138, 167], while prominent representants where no control flow is specified are [72, 97, 106].
- *Presence of a data flow of the solution*.
  In this case, we already know how data between the potential services will be communicated. If the control flow is given, usually also the data flow is given or even completely ignored (since not relevant for the composition problem). However, it is possible that the data flow is given but the control is not available. A prominant approach defined in this setting is the one of Bertoli et al. [23]. The task is then to find an admissible *order* of invocations of the services. However, most approaches that do not assume the control flow given also assume that no data flow is predefined [72, 97, 109].
- *Formalism to describe operation semantics*.
  Not all approaches define the semantics of operations [20, 167]. However, if these are specified, this is usually done using logic preconditions and postcon-

Table 2.1 An (uncomplete) overview of features of automated software composition problems and their possible characteristics.

	Feature	Domain (Possible Characteristics of the Respective Feature)
Inputs	Prespecification of Control Flow	not defined, partially or completely defined wrt. structure, semantics, or both
	Prespecification of Data Flow	not defined, partially defined, completely defined
	Operation Descriptions	not specified, explicitly (tags), preconditions & effects
	Language of Descriptions	none, (temporal) propositional logic, arbitrary variants of FOL
	QoS Requirements	none or combination of: hard constraints, soft constraints, objective function
	QoS Ranges	fix values, intervals, distributions
	Input Data	none, example data, target data
	Signature Complexity of Operations	none, at most one input and/or output, arbitrary inputs and outputs
	Deterministic Operation Behavior	no, yes
	Usage Constraints on Operations	no, yes
	Information Gathering versus World Altering	all operations are read-only versus operations also change the state of the world
	Expiration Time of Operation Outputs	invocation and reasonable persistence (IRP) vs. possible invalidation
	General Domain Knowledge	none, taxonomies, ontologies, arbitrary FOL rules
	Object Level Domain Knowledge	not available, available
Outputs	General Type of Output	output is a piece of software, output is the result obtained from *executing* the composition
	Composition Structure	sequences possibly with any combination of: alternatives, loops, flows
	Number of Solutions	one solution, set of solution candidates
	Atomicity of Composition	no, yes (composition can undo steps on failure)
Behavior	Composition & Execution Interleaved	no, yes
	Selection versus Planning	no planning involved, planning involved
	Maximum Usage per Operation	at most once, some upper bound, arbitrary use
	Precondition Satisfaction	preconditions must be satisfied by predecessor, preconditions arbitrarily satisfied

ditions/effects [72, 97, 106] or sometimes through keywords (tags) [99]. I do not consider the usage of (ontological) types as semantic descriptions.

- *Language of semantic descriptions*.
  If operations have descriptions in form of preconditions and effects, these may be specified in different ways. On one hand, they can be propositional, allowing for efficient composition but does not allow to express relations between inputs and outputs of operations [75, 135]. On the other hand, they can be (a subclass of) first order logic [72, 97, 106]. This is significantly more expressive at the natural cost of higher problem complexity.
- *Number of inputs and outputs of operations*.
  Some approaches ignore inputs and outputs completely, since data flow is not important for them [20]. Most other approaches do not impose limitations on the number of inputs and outputs, but it is imaginable to restrict them to only one output (as in Java).
- *How Quality of Service (QoS) is considered*.
  QoS is the common term to describe nonfunctional properties of services. Usually, these are properties like price, throughput, availability, trust, etc. [167]. If considered at all, QoS requirements can be posed as hard constraints [106], soft constraints with penalties [57], or be subject to optimization [167].
- *Description of QoS properties*.
  QoS properties are most of the time considered as scalar values. However, these could be more complex structures such as intervals (value is within a range), density functions (value is a random variable distributed in a particular way), or other functions (e.g., the price of a service depends on the number of invocations within a session).
- *Deterministic behavior of operations*.
  Is the response identical for every two equal invocations? This is the case if the implementation of the operation is stateless and does not contain random elements. Most approaches assume deterministic behavior of operations, but some also consider the more complex case [23].
- *Expiration time of operation outputs and effects*.
  The outputs of an operation invocation usually may become invalid after some time. For example, if the answer is the price of a flight, then the information is only valid within a short range of time. However, most approaches make the assumption of invocation and reasonable persistence (IRP). Under this assumption, the result of an operation invocation remains valid at least throughout the execution of the composition.
- *Information gathering or world altering*.
  If a property P holds after an operation, we must distinguish the case that P was *determined* to hold or whether it was *made* true. Settings that are purely information gathering are read-only settings. Together with the IRP assumption, this means that knowledge gathered by operations does never become false; hence, this constitutes a monotonic setting. In a world altering setting, however, information that was true at some point in the composition may be false later. Every approaches that does

not assume the control flow given implicitly makes this assumption, but, this point is rarely discussed.
- *Dependencies and conflicts among services*.
  Some approaches apply constraints on the common usage of services of the form: If service A is used, then B must not be used.
- *Presence of data to be fed to the solution*.
  Some approaches assume that the data passed to the search composition is already given in the query [98]. This allows for a composition mechanic that interleaves composition and execution. Most approaches, however, do not make this assumption.
- *General domain knowledge is specified*.
  Most approaches only assume a set of services given, but no background knowledge is used [72, 75, 135]. However, there are also some approaches that allow for domain knowledge in form of logic formulas [60,106].
- *Object level domain knowledge*.
  Domain knowledge can be given in form of general rules (previous point) but also in form of facts, e.g., ground literals that are known to be true in a particular domain, e.g., that FRA is an airport close to Frankfurt, Germany. This is particularly relevant in information integration-based settings like [8].

There are very few dependencies among the input features. That is, most combinations of input characteristics is theoretically imaginable. Of course, there are exceptions, e.g., if no QoS requirements are specified, then the ranges of QoS values is irrelevant; or data are only relevant if the operations are considered with inputs and outputs. Still, most features are independent, so I did not present them in form of a (unreasonably dense) feature diagram.

2.2.2 Output Features

I identify four features of the output of the composition algorithm:

- First, a composition algorithm may return a *piece of composed software* or the *result of the execution of a piece of software*. While in the first case the invoker is interested in a functionality that he can reuse arbitrarily often, the second case reflects some kind of database query whose outputs are the results of some more or less complex computation.
- Second, a composition algorithm may return extremely different *composition structures*, which can range from sequences of service operation calls to complex structures with alternative branches, loops, and concurrency.
- Third, a composition algorithm may return *different numbers of solutions*. Since many aspects that may be relevant for the requester cannot be efficiently formalized, the algorithm cannot necessarily take the final decision about the appropriateness of a solution; hence, it may return not only one but a hole set of solution candidates among which the requester can select.

- Finally, compositions may or may not be equipped with the transactional property of *atomicity*, which means that if their execution fails, potentially performed changes on the world are automatically rolled back.

In contrast to the input features, there are some dependencies among these output features. For example, if the general type of the composition output is the result performed by the execution of the identified composition, then the other features are irrelevant. Also, atomicity of compositions somehow requires that the composition is not purely sequential, because purely sequential compositions cannot react on possible execution failures of the invoked operations.

2.2.3 Behavior Features

Finally, I identified four features that describe high level characteristics about the behavior of a composition algorithm.

- First, a composition algorithm may or may not *interleave* the composition process and the software execution process.
- Second, a composition algorithm may be either a pure *selection* algorithm (selects operations for several placeholders of a given template) or a *planning* algorithm (also makes structural decisions on the control flow and data flow). Of course, planning is only relevant if the control flow and data flow are not already completely fixed in the input.
- Third, a composition algorithm may be limited in *how often* it may use (different instances) of every available services and their operations.
- Fourth, planning-based composition algorithms may be limited in *how the preconditions* of added service operations must be satisfied. For example, in [14], the precondition of an operation must be completely satisfied by the immediate predecessor in the control flow.

2.3 The Main Service Composition Problem Classes

In this section, I propose the presence of structural information about the solution as the main criterion for classification. That is, I use only one of the above feature (prespecification of control flow) as a classification criterion. This is a suitable criterion not only because it avoids hybrids that belong into both classes but also because it splits the field into two equally large subfields. This section discusses the goals, main research questions, use cases, and complexity of the two classes.

2.3.1 Class Identification

One way to separate the problems into two intuitive classes is to ask whether or not the structure of the desired composition is given. Here, the term *structure* refers to some form of definition of the control flow of the solution. Figure 2.1 shows this high-level classification scheme. If the structure is given, e.g., in form of a template with placeholders, then the composition problem is to *bind* the placeholders to concrete services. If the structure is not given, then the composition problem is to *find* it, i.e., to find the control and data flow of the desired service.

Even though other classification criteria are possible, this one is particularly convincing for three reasons. First, it defines a real partition on the field. Either some (possibly partial) structure is available for the input or it is not. For every approach, exactly one of the two assertions is true, so there are no hybrids. Second, the classification separates the field into two roughly equally large subfields, which can be seen in the following two chapters. Third, deciding the class for an approach is easy, because the question whether or not a structure is available can be answered immediately. Consequently, the question whether or not the structure of the solution is known is a good (maybe the best) criterion.

Another striking argument for this classification is that it distinguishes between two fundamentally different use cases. In the first case, the objective is to find a good *variant* of a known process. In the second case, the objective is to *design* a new process that satisfies a functional requirement specification.

As a consequence, the motivations and research questions pursued with approaches in the two classes is very different. In the following, I discuss these aspects the goals and the main research questions of the two classes in some more detail. Section 2.3.4 compares the two classes on a high level with each other.

Since the discussion of subclasses of these two classes is very exhaustive, I defer this discussion to the respective chapters. This is basically because the features used to form the subclasses are different for the two classes. Hence, Chap. 3 discusses the subclasses of the class of approaches that assume that the structure is known,

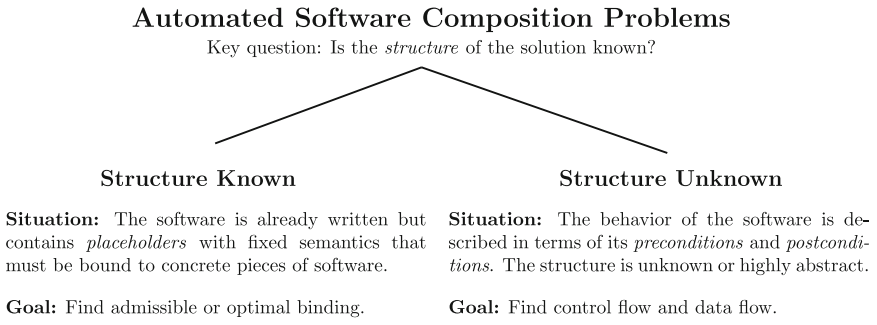

Fig. 2.1 The availability of a template is the best classification criterion

and Chap. 4 discusses the subclasses of the class of approaches that assume that the structure is not known. For the same reason, there is not one large decision tree.

2.3.2 Goals and Focus When the Structure is Known

The goal of approaches where the structure of the desired component is known is to construct a machine that refines the abstract description and binds its abstract parts to existing service operations. So the subject of automation is the selection of both an appropriate refinement and the concrete services and operations occuring in them.

The behavior of the desired component is described by a *template*. Figure 2.2 shows a brief sketch of a template and shows that it already specifies the control flow and the data flow (not visualized) of the desired component but leaves placeholders in it, which still must be bound to concrete services. So the eventual composition has already been defined on a more or less abstract level.

Several aspects may play a role in the instantiation process of the predefined template. First, we might consider nonfunctional properties such as price, execution time, etc., and find a solution that is (globally) optimal with respect to these properties. Second, we might consider functional constraints such as the behavior of candidate operations, exclusion constraints, invocation order constraints of used operations, etc. Third, we might be interested in solution that replace placeholders not only by atomic operation calls but by entired subcompositions.

There are several research questions relevant for approaches within this class.

1. How can nonfunctional aspects relevant for service composition be modeled as an optimization problem? Research is mostly concerned about how properties associated with individual service choices must be aggregated to the whole composition.

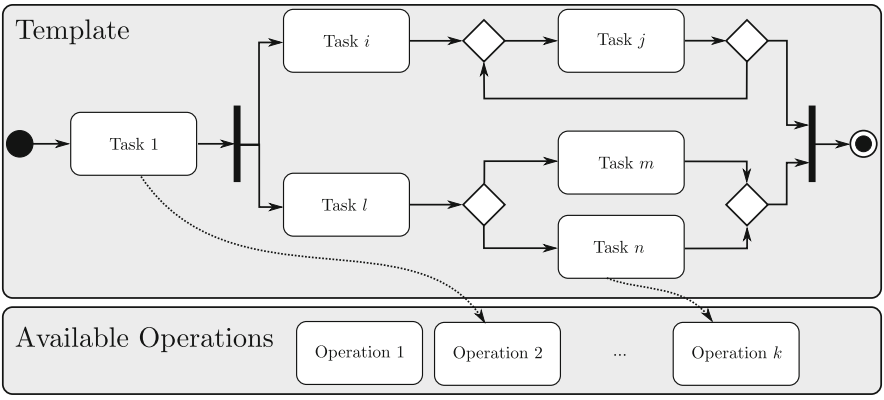

Fig. 2.2 Automated service composition with a given solution structure

2. How can templates be instantiated such that they satisfy functional constraints imposed by the user or the environment? For example, in the Roman model the question is how the placeholders can be replaced such that the communication protocols of used services are satisfied.
3. How can templates be instantiated if placeholders may be bound not only to atomic service operations but to entire compositions that must be created on the fly? Research is mostly concerned with the question how the search process for the instantiation can be designed such that a functionally valid solution is obtained.
4. How can this type of service composition be integrated into the software development workflow?

2.3.3 Goals and Focus When the Structure is Unknown

The goal of approaches that have no structure of the solution given is to construct a machine that computes outputs with the required properties given inputs with the promised properties. Figure 2.3 provides a sketch of this scenario. The intended behavior of the desired composition is specified in terms of *preconditions* that may be assumed to be true on execution and *postconditions* that are expected to hold after execution of the composition.

Approaches in this class devise a new paradigm of programming, which is declarative programming with translation into imperative code before compile time. Instead of writing the code of the desired algorithm itself, i.e., specify functions to be invoked, the developer only says what the algorithm can assume to hold at time of invocation and what should be true at the end of the desired code. Preconditions and postcondi-

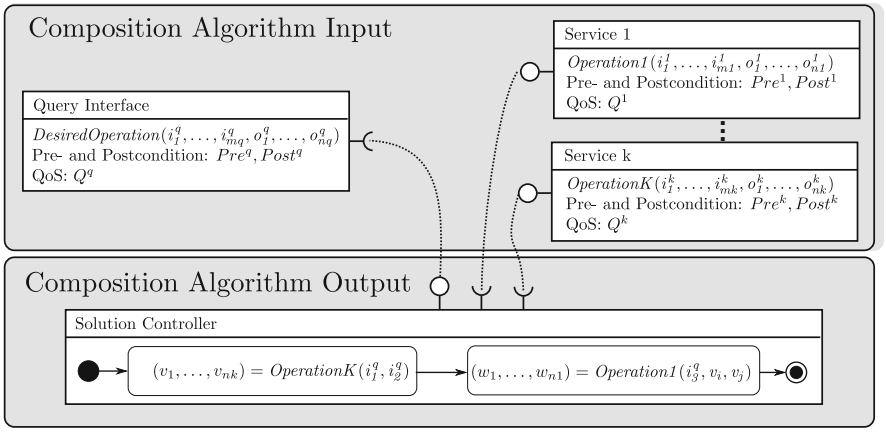

Fig. 2.3 Automated service composition without a solution structure

tions are formulated in terms of propositional or first order logic, sometimes through the notion of ontological concepts.

In other words, approaches in this class aim at exceeding the automation enabled already by compilers for high level languages. In fact, every compiler automatically creates software from a given formal description. However, compilers underly a deterministic translation process, so there is a direct correspondence between what the developer writes and the program that will be created later. Automated software composition goes a step further and allows the developer to use programming constructs that do not even have an executable implementation at design time but whose implementation is tried to be achieved automatically before the compiler is run. In contrast to the translation process of the compiler, which only fails on invalid inputs, the composition process may also fail because it cannot find an implementation with the requested properties.

Readers familiar with verification will notice the close relation between automated service composition and that area. In classical verification, a Hoare triple $\{P\}S\{Q\}$ is correct if we can show that, assuming that the (logical) precondition $\{P\}$ is true, the execution of the code statement S yields the condition $\{Q\}$. Verification assumes the program statement S being given as input for the verification process. In automated software composition, the composition algorithm must *create* a code statement S such that $\{P\}S\{Q\}$ is a correct Hoare triple. Hence, verification is a *subproblem* of this composition problem; usually, it is solved implicitly during search.

There are several research questions related to this composition type.

1. How expressive can (or should) requirements be specified? On one hand, propositional logic is usually not expressive enough, because it cannot relate the outputs produced by an operation to the inputs passed to it. On the other hand, complete first order logic may cause significant computational issues.
2. How can the search process for compositions be designed, and how can it be made efficient? This is simple for many propositional logical requirement definitions but hard for first order logical requirement definitions, (which are of particular interest because of the aforementioned reasons).
3. How can possibly competing nonfunctional properties be considered in this search process?
4. Fourth, how can the acceptance of this very formal approach be increased among developers and how can it be integrated into the development workflow at all?

This is only an (incomplete) overview over the high level questions; of course, each of them can be expanded into many subquestions.

2.3.4 Comparative Discussion of the Classes

The *conceptual* difference between the two classes is not the type of algorithm input but the actual problem that we want to solve. In particular, the two problems have different use cases, are unequally difficult to solve, and are faced with different objections.

Expected Use Cases

In the class of availability of the solution structure, the use case is that we want to find a (possibly optimal) *variant* of a general operation that best suits the individual context and requirements of a *user*. The template contains the algorithm that implements the desired functionality, leaving only some placeholders that may be replaced in different manners depending on the context. This type of composition may be required in situations where different clients want to use the same service but have different preferences regarding nonfunctional properties. For example, both want to use a service that determines the cheapest flights for some journey. While the first client wants the results within a second and accepts a higher price, the second client accepts only a low price but does not care about runtime. The decision how placeholders should be replaced depends on the actual preferences of the requesting client. So the user of the composition algorithm has potentially no or only very few knowledge in the area of software engineering.

The second class corresponds to the use case where a developer wants to have the implementation of an algorithm be generated automatically; the focus is the implementation of some desired *functionality*. This is sometimes called *automated programming*, but a better term would be *code deduction*. The desired functionality has not yet been fixed in a template but must be formulated by the user himself. This can only be done by someone with sufficient skills in software engineering. Hence, the context of this type of composition is a software development environment that combines imperative or functional programming with this type of code deduction.

Unequal Difficulty of the Problems

Intuitively, the hardness of automation seems to be different for the two classes. On one hand, it seems that, at least in theory, every composition that was obtained by the instantiation of a template can also be obtained when no structure is given at all. On the other hand, the absence of limitations on control flow and data flow leaves *much* more work to do for the composition algorithm. Instead of checking a (possibly large but) finite set of variants, the set of possible compositions when no structure is given is generally infinite. This discrepance obviously induces also a difference in the complexity.

However, the range of complexities within the respective classes is quite wide, so we cannot really compare the complexity between the classes but only between approaches. For both classes, there are cases constructible that have constant time complexity and others that are undecidable. Hence, we cannot draw an overall conclusion about the hardness of the classes; the approaches must be considered individually.

The only observation we can make is that, currently, all approaches that apply to the first class decide the problem to which they are applied while there are two approaches in the second class that do not have this property. More specifically, the approaches presented in [97,109] are not guaranteed to terminate if no solution exists.

Main Conceptual A Priori Objections

The main objection against automated software composition if a structure is available is the lack of variants that must be tried. Most papers are motivated by the "enormous and evergrowing number of services", but those legions of services seem to be hidden quite effectively from potential customers in the real world. In fact, none of the approaches within this class credibly reports the (potential of an) application to an at least somewhat real world setting. Do we really have hundreds or at least dozens of operations available for each task of the template such that automation is necessary? If this is not the case, then the number of variants is relatively small, and the composition task is often trivial.

A good answer to this objection could be that variants come from *parametrization* of operations, and that, even if the set of variants is relatively small, we must give an answer in very short time, e.g., because the composition must be found at runtime of the program that embeds it. So there are actually two arguments. First, the variants may not come from many different operations but many possible parametrizations of some few ones. Second, there may be the need to find compositions on-the-fly, which, even if the set of variants is small, makes it unacceptable to configure the solutions manually.

Approaches that create compositions from scratch based on preconditions and postconditions are often faced with the objection that we cannot assume formal preconditions and postconditions to be available in real software development. This is obviously an important issue, because most papers simply assume that operations have semantic preconditions and effects given, but semantic descriptions are rare in practice. Neither do the legions of publicly available (and semantically described) services exist, nor do developers annotate functions with preconditions and effects that would make them reusable by these automation techniques.

A good answer to this objection could be that developers actually *do* provide formal specifications anyway and that specifications in form of preconditions and effects can be hardly expected unless there are powerful tools that would give a benefit to the developer. So there are also two arguments in favor of this type of automation. First, software developers do nothing else but write formal specifications all the time, namely, the implementation of functions. So specifying preconditions and effects in addition to signatures is just to specify a little more than what is specified already anyway. This becomes even more true in environments where developers are forced to provide semantic documentation such as JavaDoc. Also, the core of descriptions are rather the postconditions; preconditions may often be empty. Moreover, semantic descriptions could, at least in parts, also be derived automatically, e.g., the description of getter methods of entity classes. Second, the current non-existence of preconditions and effects is absolutely no argument that this cannot change in the future if a benefit arises for the developer. We have already seen that developers are ready to specify tons of descriptions of classes and methods where they could write a much smaller algorithm with nice goto commands. Of course, unless there is a mature composition tool that makes this benefit available to the user, no semantic descriptions can be expected.